Borland®
A MICRO FOCUS COMPANY

Silk Test 15.5

Silk Test Classic
Classic Agent Help

Micro Focus
575 Anton Blvd., Suite 510
Costa Mesa, CA 92626

Copyright © Micro Focus 2014. All rights reserved. Portions Copyright © 1992-2009 Borland Software Corporation (a Micro Focus company).

MICRO FOCUS, the Micro Focus logo, and Micro Focus product names are trademarks or registered trademarks of Micro Focus IP Development Limited or its subsidiaries or affiliated companies in the United States, United Kingdom, and other countries.

BORLAND, the Borland logo, and Borland product names are trademarks or registered trademarks of Borland Software Corporation or its subsidiaries or affiliated companies in the United States, United Kingdom, and other countries.

All other marks are the property of their respective owners.

2014-06-06

Contents

Licensing Information 17
Getting Started 18
Silk Test Product Suite 18
Contacting Micro Focus 19
Information Needed by Micro Focus SupportLine 19
Product Notification Service 19
Classic Agent 20
How Silk Test Classic Assigns an Agent to a Window Declaration 20
Agent Options 20
Setting the Default Agent 38
Setting the Default Agent Using the Runtime Options Dialog Box 39
Setting the Default Agent Using the Toolbar Icons 39
Connecting to the Default Agent 39
Creating a Script that Uses Both Agents 39
Overview of Record Functionality Available for the Silk Test Agents 40
Setting the Window Timeout Value to Prevent Window Not Found Exceptions 41
Manually Setting the Window Timeout Value 41
Setting the Window Timeout Value in the Agent Options Dialog Box 41
Basic Workflow for the Classic Agent 43
Creating a New Project 43
Enabling Extensions Automatically Using the Basic Workflow 43
Setting the Recovery System for the Classic Agent 44
Recording a Test Case With the Classic Agent 45
Running a Test Case 46
Viewing Test Results 46
Troubleshooting Basic Workflow Issues 47
Migrating from the Classic Agent to the Open Agent 48
Differences for Agent Options Between the Classic Agent and the Open Agent 48
Differences in Object Recognition Between the Classic Agent and the Open Agent 49
Differences in the Classes Supported by the Open Agent and the Classic Agent 51
Differences in the Parameters Supported by the Open Agent and the Classic Agent 55
Overview of the Methods Supported by the Silk Test Classic Agents 56
SYS Functions Supported by the Open Agent and the Classic Agent 56
Silk Test Classic Projects 58
Storing Project Information 58
Accessing Files Within Your Project 59
Sharing a Project Among a Group 59
Project Explorer 60
Creating a New Project 61
Opening an Existing Project 62
Converting Existing Tests to a Project 62
Using Option Sets in Your Project 62
Editing an Options Set 63
Silk Test Classic File Types 63
Organizing Projects 64
Adding Existing Files to a Project 64
Renaming Your Project 65
Working with Folders in a Project 65
Moving Files Between Projects 67
Removing Files from a Project 67

Turning the Project Explorer View On and Off 67
Viewing Resources Within a Project 68
Packaging a Silk Test Classic Project 68
Emailing a Packaged Project 70
Exporting a Project 71
Troubleshooting Projects 71
Files Not Found When Opening Project 71
Silk Test Classic Cannot Load My Project File 72
Silk Test Classic Cannot Save Files to My Project 72
Silk Test Classic Does Not Run 72
My Files No Longer Display In the Recent Files List 73
Cannot Find Items In Classic 4Test 73
Editing the Project Files 73
Enabling Extensions for Applications Under Test 74
Extensions that Silk Test Classic can Automatically Configure 74
Extensions that Must be Set Manually 75
Extensions on Host and Target Machines 75
Enabling Extensions Automatically Using the Basic Workflow 76
Enabling Extensions on a Host Machine Manually 76
Manually Enabling Extensions on a Target Machine 77
Enabling Extensions for Embedded Browser Applications that Use the Classic Agent 78
Enabling Extensions for HTML Applications (HTAs) 78
Adding a Test Application to the Extension Dialog Boxes 79
Verifying Extension Settings 80
Why Applications do not have Standard Names 80
Duplicating the Settings of a Test Application in Another Test Application 80
Deleting an Application from the Extension Enabler or Extensions Dialog Box 81
Disabling Browser Extensions 81
Comparison of the Extensions Dialog Box and the Extension Enabler Dialog Box 81
Configuring the Browser 82
Setting Agent Options for Web Testing 83
Specifying a Browser for Silk Test Classic to Use in Testing a Web Application 83
Specifying your Default Browser 84
Understanding the Recovery System for the Classic Agent 85
Setting the Recovery System for the Classic Agent 86
Base State 87
DefaultBaseState Function 87
Adding Tests that Use the Classic Agent to the DefaultBaseState 88
DefaultBaseState and wMainWindow 88
Flow of Control 89
The Non-Web Recovery Systems Flow of Control 89
Web Applications and the Recovery System 89
How the Non-Web Recovery System Closes Windows 90
How the Non-Web Recovery System Starts the Application 90
Modifying the Default Recovery System 91
Overriding the Default Recovery System 91
Handling Login Windows 92
Handling Browser Pop-up Windows in Tests that Use the Classic Agent 94
Specifying Windows to be Left Open for Tests that Use the Classic Agent 95
Specifying New Window Closing Procedures 95
Specifying Buttons, Keys, and Menus that Close Windows 96
Recording a Close Method for Tests that Use the Classic Agent 96
Test Plans 98
Structure of a Test Plan 98
Overview of Test Plan Templates 99

Example Outline for Word Search Feature 99
Converting a Results File to a Test Plan 101
Working with Test Plans 101
Creating a New Test Plan 101
Indent and Change Levels in an Outline 102
Adding Comments to Test Plan Results 102
Documenting Manual Tests in the Test Plan 103
Describing the State of a Manual Test 103
Inserting a Template 103
Changing Colors in a Test Plan 104
Linking the Test Plan to Scripts and Test Cases 104
Working with Large Test Plans 105
Determining Where Values are Defined in a Large Test Plan 105
Dividing a Test Plan into a Master Plan and Sub-Plans 105
Creating a Sub-Plan 106
Copying a Sub-Plan 106
Opening a Sub-Plan 106
Connecting a Sub-Plan with a Master Plan 106
Refreshing a Local Copy of a Sub-Plan 106
Sharing a Test Plan Initialization File 106
Saving Changes 107
Overview of Locks 107
Acquiring and Releasing a Lock 107
Generating a Test Plan Completion Report 107
Adding Data to a Test Plan 108
Specifying Unique and Shared Data 108
Adding Comments in the Test Plan Editor 108
Testplan Editor Statements 108
The # Operator in the Testplan Editor 108
Using the Testplan Detail Dialog Box to Enter the testdata Statement 109
Entering the testdata Statement Manually 109
Linking Test Plans 109
Linking a Description to a Script or Test Case using the Testplan Detail Dialog Box 109
Linking a Test Plan to a Data-Driven Test Case 110
Linking to a Test Plan Manually 110
Linking a Test Case or Script to a Test Plan using the Testplan Detail Dialog Box 110
Linking the Test Plan to Scripts and Test Cases 110
Example of Linking a Test Plan to a Test Case 111
Categorizing and Marking Test Plans 111
Marking a Test Plan 112
How the Marking Commands Interact 112
Marking One or More Tests 112
Printing Marked Tests 112
Using Symbols 113
Overview of Symbols 113
Symbol Definition Statements in the Test Plan Editor 114
Defining Symbols in the Testplan Detail Dialog box 115
Assigning a Value to a Symbol 115
Specifying Symbols as Arguments when Entering a testcase Statement 115
Attributes and Values 116
Overview of Attributes and Values 116
Predefined Attributes 116
User Defined Attributes 116
Adding or Removing Members of a Set Attribute 117

Rules for Using + and - 117
Defining an Attribute and its Values 117
Assigning Attributes and Values to a Test Plan 118
Assigning an Attribute from the Testplan Detail Dialog Box 118
Modifying the Definition of an Attribute 119
Queries 119
Overview of Test Plan Queries 119
Overview of Combining Queries to Create a New Query 120
Guidelines for Including Symbols in a Query 120
The Differences between Query and Named Query Commands 121
Create a New Query 121
Edit a Query 122
Delete a Query 122
Combining Queries 122
Designing and Recording Test Cases with the Classic Agent 123
Hierarchical Object Recognition 123
Highlighting Objects During Recording 124
Setting Recording Preferences for the Classic Agent 124
Test Cases 125
Overview of Test Cases 125
Anatomy of a Basic Test Case 126
Types of Test Cases 126
Test Case Design 126
Constructing a Test Case 127
Data in Test Cases 128
Saving Test Cases 128
Recording Without Window Declarations 129
Overview of Application States 129
Behavior of an Application State Based on NONE 130
Example: A Feature of a Word Processor 130
Recording Test Cases with the Classic Agent 131
Overview of Recording the Stages of a Test Case 131
Overview of Recording 4Test Components 132
Recording a Test Case With the Classic Agent 133
Verifying a Test Case 133
Recording the Cleanup Stage and Pasting the Recording 134
Testing the Ability of the Recovery System to Close the Dialog Boxes of Your Application 135
Linking to a Script and Test Case by Recording a Test Case 135
Saving a Script File 136
Recording an Application State 136
Testing an Application State 137
Recording Actions 137
Recording the Location of an Object 137
Recording Window Identifiers 138
Verification 138
Verifying Object Properties 138
Verifying Object Attributes 140
Overview of Verifying Bitmaps 141
Overview of Verifying an Objects State 141
Fuzzy Verification 142
Verifying that a Window or Control is No Longer Displayed 144
Data-Driven Test Cases 144
Data-Driven Workflow 145
Working with Data-Driven Test Cases 146
Code Automatically Generated by Silk Test Classic 146

Tips And Tricks for Data-Driven Test Cases 147
Testing an Application with Invalid Data 148
Enabling and Disabling Workflow Bars 149
Data Source for Data-Driven Test Cases 149
Creating the Data-Driven Test Case 151
Property Sets 156
Verifying Properties as Sets 156
Creating a New Property Set 157
Combining Property Sets 157
Deleting a Property Set 157
Editing an Existing Property Set 157
Specifying a Class-Property Pair 158
Predefined Property Sets 158
Characters Excluded from Recording and Replaying 159
Testing in Your Environment with the Classic Agent 160
Distributed Testing with the Classic Agent 160
Configuring Your Test Environment 160
Running Test Cases in Parallel 167
Testing Multiple Machines 175
Testing Multiple Applications 182
Troubleshooting Distributed Testing 194
Testing ActiveX/Visual Basic Controls 195
Overview of ActiveX/Visual Basic Support 195
Enabling ActiveX/Visual Basic Support 196
Predefined Classes for ActiveX/Visual Basic Controls 196
Predefined Class Definition File for Visual Basic 196
List of Predefined ActiveX/Visual Basic Controls 196
Access to VBOptionButton Control Methods 198
0-Based Arrays 198
Dependent Objects and Collection Objects 199
Working with Dynamically Windowed Controls 199
Window Timeout 199
Conversion of BOOLEAN Values 200
Testing Controls: 4Test Versus ActiveX Methods 200
Control Access is Similar to Visual Basic 200
Prerequisites for Testing ActiveX/Visual Basic Controls 201
ActiveX/Visual Basic Exception Values 201
Recording New Classes for ActiveX/Visual Basic Controls 201
Loading Class Definition Files 202
Disabling ActiveX/Visual Basic Support 202
Ignoring an ActiveX/Visual Basic Class 202
Setting ActiveX/Visual Basic Extension Options 203
Setup for Testing ActiveX Controls or Java Applets in the Browser 204
Client/Server Application Support 204
Client/Server Testing Challenges 204
Verifying Tables in ClientServer Applications 205
Evolving a Testing Strategy 205
Incremental Functional Test Design 206
Network Testing Types 206
How 4Test Handles Script Deadlock 207
Troubleshooting Configuration Test Failures 208
Testing .NET Applications with the Classic Agent 208
Enabling .NET Support 208
Tips for Working with .NET 209
Windows Forms Applications 209
Testing Java AWT/Swing Applications with the Classic Agent 217

Testing Standard Java Objects and Custom Controls 218
Recording and Playing Back JFC Menus 218
Recording and Playing Back Java AWT Menus 219
Object Recognition for Java AWT/Swing Applications 219
Supported Controls for Java AWT/Swing Applications 219
Java AWT Classes for the Classic Agent 219
Supported Java Virtual Machines 219
Supported Browsers for Testing Java Applets 219
Overview of JavaScript Support 220
Support for JavaBeans 220
Classes in Object-Oriented Programming Languages 221
Configuring Silk Test Classic to Test Java 221
Testing Java Applications and Applets 226
Frequently Asked Questions About Testing Java Applications 255
Testing Java SWT and Eclipse Applications with the Classic Agent 256
Suppressing Controls (Classic Agent) 256
Java SWT Classes for the Classic Agent 257
Testing Web Applications with the Classic Agent 257
Supported Controls for Web Applications 257
Sample Web Applications 257
API Click Versus Agent Click 257
Testing Dynamic HTML (DHTML) Popup Menus 258
Web Application Setup Steps 259
Recording the Test Frame for a Web Application 259
Recording Window Declarations for a Web Application 259
Streamlining HTML Frame Declarations 260
Test Frames 260
User Options 262
Testing Methodology for Web Applications 266
VO Automation 267
Testing Objects in a Web Page 268
General Web Application Classes 279
Testing Windows API-Based Applications 279
Overview of Windows API-Based Application Support 279
Locator Attributes for Windows API-Based Applications 280
Suppressing Controls (Classic Agent) 280
Suppressing Controls (Open Agent) 280
Configuring Standard Applications 281
Determining the priorLabel in the Win32 Technology Domain 282
Testing Applications with the SilkBean 282
Preparing Test Scripts to Run with SilkBean 282
Configuring SilkBean Support on the Target (UNIX) Machine 283
Configuring SilkBean Support on the Host Machine when Testing Multiple Applications 284
Correcting Problems when Using the SilkBean 284
Using Advanced Techniques with the Classic Agent 286
Starting from the Command Line 286
Starting Silk Test Classic from the Command Line 286
Starting the Classic Agent from the Command Line 288
Recording a Test Frame 289
Overview of Object Files 289
Declarations 291
Window Declarations 295
Identifiers and Tags 299
Save the Test Frame 301
Specifying How a Dialog Box is Invoked 302

Class Attributes 302
Improving Object Recognition with Microsoft Accessibility 304
Enabling Accessibility 305
Adding Accessibility Classes 305
Improving Object Recognition with Accessibility 305
Removing Accessibility Classes 306
Calling Windows DLLs from 4Test 306
Aliasing a DLL Name 307
Calling a DLL from within a 4Test Script 307
Passing Arguments to DLL Functions 308
Using DLL Support Files Installed with Silk Test Classic 309
Extending the Class Hierarchy 310
Classes 310
Verifying Attributes and Properties 315
Defining Methods and Custom Properties 317
Examples 319
Porting Tests to Other GUIs 321
Handling Differences Among GUIs 321
About GUI Specifiers 327
Supporting GUI-Specific Objects 330
Supporting Custom Controls 331
Why Silk Test Classic Sees Controls as Custom Controls 331
Reasons Why Silk Test Classic Sees the Control as a Custom Control 332
Supporting Graphical Controls 332
Custom Controls (Classic Agent) 332
Using Clipboard Methods 336
Filtering Custom Classes 337
OCR Support 340
Supporting Internationalized Objects 344
Overview of Silk Test Classic Support of Unicode Content 344
Using DB Tester with Unicode Content 345
Issues Displaying Double-Byte Characters 345
Learning More About Internationalization 345
Silk Test Classic File Formats 345
Working with Bi-Directional Languages 348
Recording Identifiers for International Applications 348
Configuring Your Environment 348
Troubleshooting Unicode Content 351
Using Autocomplete 354
Overview of AutoComplete 354
Customizing your MemberList 354
Frequently Asked Questions about AutoComplete 355
Turning AutoComplete Options Off 356
Using AppStateList 357
Using DataTypeList 357
Using FunctionTip 357
Using MemberList 358
Overview of the Library Browser 358
Library Browser Source File 359
Adding Information to the Library Browser 359
Add User-Defined Files to the Library Browser with Silk Test Classic 360
Viewing Functions in the Library Browser 360
Viewing Methods for a Class in the Library Browser 360
Examples of Documenting User-Defined Methods 361
Web Classes Not Displayed in Library Browser 361
Text Recognition Support 362

Running Tests and Interpreting Results 364
Running Tests 364
Creating a suite 364
Passing Arguments To a Script 364
Running a Test Case 365
Running a testplan 366
Running the currently active script or suite 366
Stopping a Running Testcase Before it Completes 366
Setting a Test Case to Use Animation Mode 367
Interpreting Results 367
Overview of the Results File 367
Viewing Test Results 368
Difference Viewer Overview 368
Errors And the Results File 368
Testplan Pass/Fail Report and Chart 370
Merging testplan results overview 370
Analyzing Results with the Silk TrueLog Explorer 370
TrueLog Explorer 371
TrueLog Limitations and Prerequisites 371
Opening the TrueLog Options Dialog Box 372
Setting TrueLog Options 372
Toggle TrueLog at Runtime Using a Script 373
Viewing Results Using the TrueLog Explorer 373
Modifying Your Script to Resolve Window Not Found Exceptions When Using TrueLog 374
Analyzing Bitmaps 374
Overview of the Bitmap Tool 374
When to use the Bitmap Tool 375
Capturing Bitmaps with the Bitmap Tool 375
Comparing Bitmaps 377
Rules for Using Comparison Commands 378
Bitmap Functions 378
Baseline and Result Bitmaps 378
Zooming the Baseline Bitmap, Result Bitmap, and Differences Window 379
Looking at Statistics 379
Exiting from Scan Mode 379
Starting the Bitmap Tool 380
Using Masks 380
Analyzing Bitmaps for Differences 383
Working with Result Files 384
Attaching a comment to a result set 384
Comparing Result Files 384
Customizing results 385
Deleting a results set 385
Change the default number of results sets 385
Changing the Colors of Elements In the Results File 385
Fix incorrect values in a script 386
Marking Failed Testcases 386
Merging results 386
Navigating to errors 386
Viewing an individual summary 387
Storing and Exporting Results 387
Storing results 387
Exporting Results to a Structured File for Further Manipulation 387
Removing the unused space in a results file 388
Sending Results Directly to Issue Manager 388

Logging Elapsed Time Thread and Machine Information 388
Presenting Results 388
Fully customize a chart 388
Generate a Pass/Fail Report on the Active Test Plan Results File 389
Producing a Pass/Fail Chart 389
Displaying a different set of results 390
Debugging Test Scripts 391
Designing and testing with debugging in mind 391
Overview of the Debugger 391
Executing a script in the debugger 391
Starting the debugger 392
Debugger menus 392
Stepping into and over functions 392
Working with scripts 393
Exiting the debugger 393
Breakpoints 393
Setting Breakpoints 393
Viewing Breakpoints 394
Deleting Breakpoints 394
Variables 394
Viewing variables 394
Changing the value of variables 395
Expressions 395
Overview of Expressions 395
Evaluate expressions 395
Enabling View Trace Listing 395
Viewing a list of modules 396
View the debugging transcripts 396
Debugging Tips 396
Checking the precedence of operators 396
Code that never executes 396
Global and local variables with the same name 396
Global variables with unexpected values 396
Incorrect use of break statements 397
Incorrect values for loop variables 397
Infinite loops 397
Typographical errors 397
Uninitialized variables 397
Troubleshooting the Classic Agent 398
ActiveX and Visual Basic Applications 398
What Happens When You Enable ActiveX/Visual Basic? 398
Silk Test Classic Does Not Display the Appropriate Visual Basic Properties 398
Silk Test Classic Does Not Recognize ActiveX Controls in a Web Application 398
Silk Test Classic Displays an Error When Playing Back a Click on a Sheridan Command Button 398
Silk Test Classic Displays Native Visual Basic Objects as Custom Windows 399
Record Class Finds no Properties or Methods for a Visual Basic Object 399
Inconsistent Recognition of ActiveX Controls 399
Test Failures During Visual Basic Application Configuration 400
Application Environment 400
Dr. Watson when Running from Batch File 400
I Cannot Get Silk Test Classic to Work With JBuilder or Oracle JDeveloper 400
Silk Test Classic does not Launch my Java Web Start Application 400
Which JAR File do I Use? 401
Sample Declarations and Script for Testing JFC Popup Menus 401
Java Extension Loses Injection when Using Virtual Network Computing (VNC) 403

Troubleshooting Basic Workflow Issues 403
Browsers 404
I Am not Testing Applets but Browser is Launched During Playback 404
Playback is Slow when I Test Applications Launched from a Browser 404
Library Browser does Not Display Web Browser Classes 404
Error Messages 405
Agent not responding 405
BrowserChild MainWindow Not Found When Using Internet Explorer 7.x 405
Cannot find file agent.exe 406
Control is not responding 406
Functionality Not Supported on the Open Agent 406
Unable to Connect to Agent 407
Unable to Delete File 407
Unable to Start Internet Explorer 407
Variable Browser not defined 408
Window Browser does not define a tag 408
Window is not active 408
Window is not enabled 409
Window is not exposed 409
Window not found 410
Functions and Methods 410
Class not Loaded Error 410
Exists Method Returns False when Object Exists 411
How can I Determine the Exact Class of a java.lang.Object Returned by a Method 411
How to Define lwLeaveOpen 412
Defining TestCaseEnter and TestCaseExit Methods 412
How to Write the Invoke Method 413
I cannot Verify $Name Property during Playback 414
Errors when calling nested methods 414
Methods Return Incorrect Indexed Values in My Scripts 415
Handling Exceptions 415
Default Error Handling 415
Custom Error Handling 415
Trapping the exception number 417
Defining your own exceptions 417
Using do...except statements to trap and handle exceptions 418
Programmatically Logging an Error 418
Performing More than One Verification in a Test Case 419
Writing an Error-Handling Function 421
Exception Values 422
Troubleshooting Java Applications 426
Why Is My Java Application Not Ready To Test? 426
Why Can I Not Test a Java Application Which Is Started Through a Command Prompt? 426
What Can I Do If My Java Application Not Contain Any Controls Below JavaMainWin? 427
How Can I Enable a Java Plug-In? 427
What Can I Do If the Java Plug-In Check Box Is Not Checked? 427
What Can I Do When I Am Testing an Applet That Does Not Use a Plug-In, But the Browser Has a 427
What Can I Do If the Silk Test Java File Is Not Included in a Plug-In? 428
What Can I Do If Java Controls In an Applet Are Not Recognized? 428
Multiple Machines Testing 428
Remote Testing and Default Browser 428
Setting Up the Recovery System for Multiple Local Applications 428

two_apps.t 429
two_apps.inc 430
Objects 435
Does Silk Test Classic Support Oracle Forms? 435
Mouse Clicks Fail on Certain JFC and Visual Café Objects 436
My Sub-Menus of a Java Menu are being Recorded as JavaDialogBoxes 436
Other Problems 436
Adding a Property to the Recorder 436
Application Hangs When Playing Back a Menu Item Pick 437
Cannot Access Some of the Silk Test Classic Menu Commands 437
Cannot Double-Click a Silk Test Classic File and Open Silk Test Classic 437
Cannot Extend AnyWin, Control, or MoveableWin Classes 438
Cannot Find the Quick Start Wizard 438
Cannot open results file 438
Cannot Play Back Picks of Cascaded Sub-Menus for an AWT Application 439
Cannot Record Second Window 439
Common DLL Problems 439
Common Scripting Problems 440
Conflict with Virus Detectors 441
Displaying the Euro Symbol 441
Do I Need Administrator Privileges to Run Silk Test Classic? 441
General Protection Faults 442
Running Global Variables from a Test Plan Versus Running Them from a Script 442
Ignoring a Java Class 443
Include File or Script Compiles but Changes are Not Picked Up 443
Library Browser Not Displaying User-Defined Methods 444
Maximum Size of Silk Test Classic Files 444
Playing Back Mouse Actions 444
Recorder Does Not Capture All Actions 445
Recording two SetText () Statements 445
Relationship between Exceptions Defined in 4test.inc and Messages Sent To the Result File 446
The 4Test Editor Does Not Display Enough Characters 446
Silk Test Classic Support of Delphi Applications 446
Stopping a Test Plan 448
A Text Field Is Not Allowing Input 448
Using a Property Instead of a Data Member 448
Using File Functions to Add Information to the Beginning of a File 449
Why Does the Str Function Not Round Correctly? 449
Troubleshooting Projects 449
Files Not Found When Opening Project 450
Silk Test Classic Cannot Load My Project File 450
Silk Test Classic Cannot Save Files to My Project 450
Silk Test Classic Does Not Run 451
My Files No Longer Display In the Recent Files List 451
Cannot Find Items In Classic 4Test 451
Editing the Project Files 451
Recognition Issues 452
How Can the Application Developers Make Applications Ready for Automated Testing? 452
I Cannot See all Objects in my Application even after Enabling Show All Classes 452
java.lang.UnsatisfiedLinkError 452
JavaMainWin is Not Recognized 452
None of My Java Controls are Recognized 453

Only JavaMainWin is Recognized 453
Only Applet Seen 453
Silk Test Classic Does not Record Click() Actions Against Custom Controls in Java Applets 454
Silk Test Classic Does not Recognize a Popup Dialog Box caused by an AWT Applet in a Browser 454
Silk Test Classic is Not Recognizing Updates on Internet Explorer Page Containing JavaScript 454
Java Controls are Not Recognized 455
Verify Properties does not Capture Window Properties 455
Tips 455
Owner-Draw List Boxes and Combo Boxes 455
Options for Legacy Scripts 456
Declaring an Object for which the Class can Vary 458
Drag and Drop Operations 458
Example Test Cases for the Find Dialog Box 459
Declaring an Object for which the Class can Vary 460
When to use the Bitmap Tool 461
Troubleshooting Web Applications 461
Why Is My Web Application Not Ready To Test? 461
What Can I Do If the Page I Have Selected Is Empty? 461
Why Do I Get an Error Message When I Set the Accessibility Extension? 461
HtmlPopupList Causes the Browser to Crash when Using IE DOM Extension 462
Silk Test Classic Does Not Recognize Links 462
Mouse Coordinate (x, y) is Off the Screen 462
Recording a Declaration for a Browser Page Containing Many Child Objects 463
Recording VerifyProperties() Detects BrowserPage Properties and Children 463
Silk Test Classic Cannot See Any Children in My Browser Page 463
Silk Test Classic Cannot Verify Browser Extension Settings 464
Silk Test Classic Cannot Find the Web Page of the Application 464
Silk Test Classic Cannot Recognize Web Objects 465
Silk Test Classic Recognizes Static HTML Text But Does Not Recognize Text ..465
A Test Frame Which Contains HTML Frame Declarations Does Not Compile 466
Web Property Sets Are Not Displayed During Verification 466
Why Does the Recorder Generate so Many MoveMouse() Calls? 467
Using the Runtime Version of Silk Test Classic 468
Installing the Runtime Version 468
Starting the Runtime Version 468
Comparing Silk Test Classic and Silk Test Classic Runtime Menus and Commands 468
Glossary 479
4Test Classes 479
4Test-Compatible Information or Methods 479
Abstract Windowing Toolkit 479
accented character 479
agent 479
applet 480
application state 480
attributes 480
Band (.NET) 480
base state 480
bidirectional text 480
Bytecode 480
call stack 481
child object 481
class 481
class library 481

class mapping 481
Classic 4Test 481
client area 481
custom object 481
data-driven test case 482
data member 482
declarations 482
DefaultBaseState 482
diacritic 482
Difference Viewer 482
double-byte character set (DBCS) 482
dynamic instantiation 482
dynamic link library (DLL) 483
enabling 483
exception 483
frame file 483
fully qualified object name 483
group description 483
handles 484
hierarchy of GUI objects 484
host machine 484
hotkey 484
Hungarian notation 488
identifier 489
include file 489
internationalization or globalization 489
Java Database Connectivity (JDBC) 489
Java Development Kit (JDK) 489
Java Foundation Classes (JFC) 489
Java Runtime Environment (JRE) 489
Java Virtual Machine (JVM) 489
JavaBeans 490
Latin script 490
layout 490
levels of localization 490
load testing 490
localization 490
localize an application 490
locator 490
logical hierarchy 491
manual test 491
mark 491
master plan 491
message box 491
method 491
minus (-) sign 491
modal 492
modeless 492
Multibyte Character Set (MBCS) 492
Multiple Application Domains (.NET) 492
negative testing 492
nested declarations 492
No-Touch (.NET) 492
object 492
outline 493
Overloaded method 493

parent object 493
performance testing 493
physical hierarchy (.NET) 493
plus (+) sign 493
polymorphism 493
project 494
properties 494
query 494
recovery system 494
regression testing 494
results file 494
script 494
script file 495
side-by-side (.NET) 495
Simplified Chinese 495
Single-Byte Character Set (SBCS) 495
smoke test 495
Standard Widget Toolkit (SWT) 495
statement 495
status line 496
stress testing 496
subplan 496
suite 496
Swing 496
symbols 496
tag 496
target machine 497
template 497
test description 497
test frame file 497
test case 497
test plan 498
TotalMemory parameter 498
Traditional Chinese 498
variable 498
verification statement 498
Visual 4Test 498
window declarations 498
window part 499
XPath 499

Licensing Information

Unless you are using a trial version, Silk Test requires a license.

The licensing model is based on the client that you are using and the applications that you want to be able to test. The available licensing modes support the following application types:

Licensing Mode	Application Type

Full	• Web applications, including the following:
	  • Apache Flex
	  • Java-Applets
	• Mobile Web applications.
	  • Android
	  • iOS
	• Apache Flex
	• Java AWT/Swing
	• Java SWT and Eclipse RCP
	• .NET, including Windows Forms and Windows Presentation Foundation (WPF)
	• Rumba
	• Windows API-Based
	 Note: To upgrade your license to a Full license, visit *www.borland.com*.

Premium	All application types that are supported with a *Full* license, plus SAP applications. **Note:** To upgrade your license to a Premium license, visit *www.borland.com*.

Getting Started

Silk Test Classic is the traditional Silk Test client. With Silk Test Classic you can develop tests using the 4Test language, an object-oriented fourth-generation language (4GL), which is designed specifically for QA professionals. Silk Test Classic guides you through the entire process of creating test cases, running the tests, and interpreting the results of your test runs.

Silk Test Classic supports the testing of a broad set of application technologies.

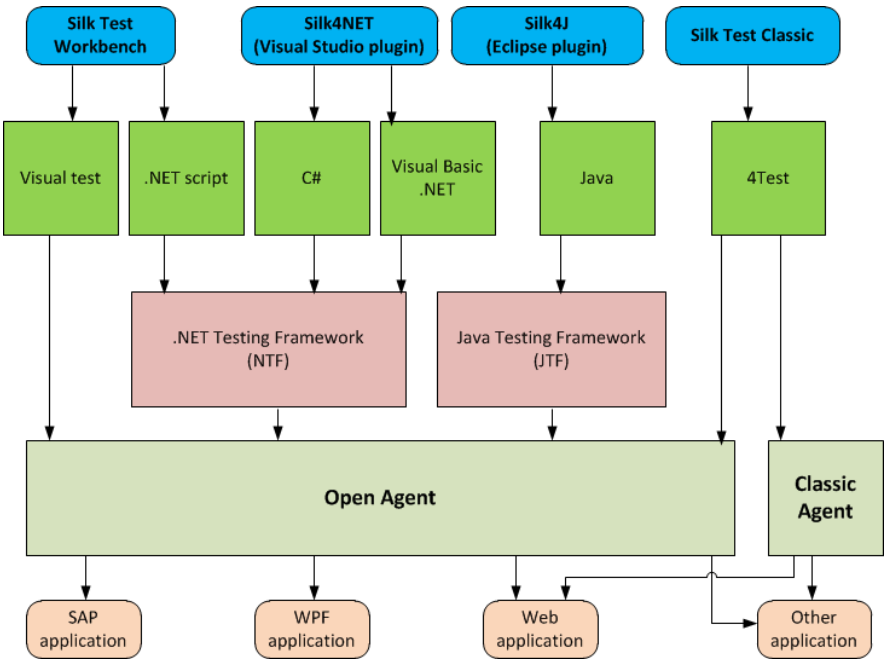
This section provides information to get you up and running with Silk Test Classic.

Silk Test Product Suite

Silk Test is an automated testing tool for fast and reliable functional and regression testing. Silk Test helps development teams, quality teams, and business analysts to deliver software faster, and with high quality. With Silk Test you can record and replay tests across multiple platforms and devices to ensure that your applications work exactly as intended.

The Silk Test product suite includes the following components:

- Silk Test Workbench – Silk Test Workbench is the quality testing environment that offers .NET scripting for power users and easy to use visual tests to make testing more accessible to a broader audience.
- Silk4NET – The Silk4NET Visual Studio plug-in enables you to create Visual Basic or C# test scripts directly in Visual Studio.
- Silk4J – The Silk4J Eclipse plug-in enables you to create Java-based test scripts directly in your Eclipse environment.
- Silk Test Classic – Silk Test Classic is the traditional, 4Test Silk Test product.
- Silk Test Agents – The Silk Test Agent is the software process that translates the commands in your tests into GUI-specific commands. In other words, the Agent drives and monitors the application you are testing. One Agent can run locally on the host machine. In a networked environment, any number of Agents can run on remote machines.



The product suite that you install determines which components are available. To install all components, choose the complete install option. To install all components with the exception of Silk Test Classic, choose the standard install option.

Contacting Micro Focus

Micro Focus is committed to providing world-class technical support and consulting services. Micro Focus provides worldwide support, delivering timely, reliable service to ensure every customer's business success.

All customers who are under a maintenance and support contract, as well as prospective customers who are evaluating products, are eligible for customer support. Our highly trained staff respond to your requests as quickly and professionally as possible.

Visit *http://supportline.microfocus.com/assistedservices.asp* to communicate directly with Micro Focus SupportLine to resolve your issues, or email supportline@microfocus.com.

Visit Micro Focus SupportLine at *http://supportline.microfocus.com* for up-to-date support news and access to other support information. First time users may be required to register to the site.

Information Needed by Micro Focus SupportLine

When contacting Micro Focus SupportLine, please include the following information if possible. The more information you can give, the better Micro Focus SupportLine can help you.

- The name and version number of all products that you think might be causing an issue.
- Your computer make and model.
- System information such as operating system name and version, processors, and memory details.
- Any detailed description of the issue, including steps to reproduce the issue.
- Exact wording of any error messages involved.
- Your serial number.

To find out these numbers, look in the subject line and body of your Electronic Product Delivery Notice email that you received from Micro Focus.

Product Notification Service

The product notification service is an application that runs in your system tray and allows you to find out if updates are available for Silk Test. It also provides a link for you to click to navigate to the updates.

Running the Service

In the system tray, click the update notification icon and the Product Notification Service application opens.

Installed Version	Provides the version number of the currently installed Silk Test application.
Update Version	Provides a link and the version number of the next minor update, if one is available.
New Version	Provides a link and the version number of the next full release, if one is available.
Settings	Click the **Settings** button to open the **Settings** window. Select if and how often you want the notification service to check for updates.

Classic Agent

The Silk Test agent is the software process that translates the commands in your test scripts into GUI-specific commands. In other words, the agent drives and monitors the application you are testing. One agent can run locally on the host machine. In a networked environment, any number of agents can run on remote machines.

Silk Test Classic provides two types of agents, the Open Agent and the Classic Agent. The agent that you assign to your project or script depends on the type of application that you are testing.

When you create a new project, Silk Test Classic automatically uses the agent that supports the type of application that you are testing. For instance, if you create an Apache Flex or Windows API-based client/server project, Silk Test Classic uses the Open Agent. When you open a project or script that was developed with the Classic Agent, Silk Test Classic automatically uses the Classic Agent. For information about the supported technology domains for each agent, refer to *Testing in Your Environment*.

The Classic Agent uses hierarchical object recognition to record and replay test cases that use window declarations to find and identify objects. With the Classic Agent, one Agent process can run locally on the host machine, but in a networked environment, the host machine can connect to any number of remote Agents simultaneously or sequentially. You can record and replay tests remotely using the Classic Agent.

For information about new features, supported platforms and versions, known issues, and work-arounds, refer to the *Release Notes*.

How Silk Test Classic Assigns an Agent to a Window Declaration

When you record a test with the Open Agent set as the default agent, Silk Test Classic includes a locator to identify the top-most window of the test application. For instance, this window declaration for a Notepad application that uses the Open Agent includes the following locator:

```
window MainWin UntitledNotepad
locator "/MainWin[@caption='Untitled - Notepad']"
```

Silk Test Classic determines which Agent to use by detecting whether a locator or `Find` or `FindAll` command is used. If no locator or `Find` or `FindAll` command is present, Silk Test Classic uses the Classic Agent.

In earlier releases, the `TAG_IS_OPEN_AGENT` tag was defined on the root window declaration of a control hierarchy to identify that the Open Agent should be used. This is no longer necessary. When Silk Test Classic detects a locator on the top-most window or detects a `Find` or `FindAll` command, the Open Agent is automatically used. When a window declaration contains both locators and tags and either could be used for resolving the window, check or uncheck the **Prefer Locator** check box in the **General Options** dialog box to determine which method is used.

Agent Options

The following table lists the `AgentClass` options that can be manipulated with the `GetOption` method and `SetOption` method. Only options that can be manipulated by the user are listed here; other options are for internal use only.

Agent Option	Agent Supported	Description
OPT_AGENT_CLICKS_ONLY	Classic Agent	BOOLEAN FALSE to use the API-based clicks; TRUE to use agent-based clicks. The default is FALSE. This option applies to clicks on specific HTML options only. For additional information, see *API Click Versus Agent Click*. This option can be set through the **Compatibility** tab on the **Agent Options** dialog box, Agent.`SetOption`, or `BindAgentOption`(), and may be retrieved through Agent.`GetOption`().
OPT_ALTERNATE_RECORD_BREAK	Classic Agent Open Agent	BOOLEAN TRUE pauses recording when **Ctrl +Shift** is pressed. Otherwise, **Ctrl+Alt** is used. By default, this is FALSE.
OPT_APPREADY_RETRY	Classic Agent Open Agent	NUMBER The number of seconds that the agent waits between attempts to verify that an application is ready. The agent continues trying to test the application for readiness if it is not ready until the time specified with OPT_APPREADY_TIMEOUT is reached.
OPT_APPREADY_TIMEOUT	Classic Agent Open Agent	NUMBER The number of seconds that the agent waits for an application to become ready. If the application is not ready within the specified timeout, Silk Test Classic raises an exception. To require the agent to check the ready state of an application, set OPT_VERIFY_APPREADY. This option applies only if the application or extension knows how to communicate to the agent that it is ready. To find out whether the extension has this capability, see the documentation that comes with the extension.
OPT_BITMAP_MATCH_COUNT	Classic Agent Open Agent	INTEGER The number of consecutive snapshots that must be the same for the bitmap to be considered stable. Snapshots

Agent Option	Agent Supported	Description
		are taken up to the number of seconds specified by OPT_BITMAP_MATCH_TIMEOUT, with a pause specified by OPT_BITMAP_MATCH_INTERVAL occurring between each snapshot. Related methods: • `CaptureBitmap` • `GetBitmapCRC` • `SYS_CompareBitmap` • `VerifyBitmap` • `WaitBitmap`
OPT_BITMAP_MATCH_INTERVAL	Classic Agent Open Agent	INTEGER The time interval between snapshots to use for ensuring the stability of the bitmap image. The snapshots are taken up to the time specified by OPT_BITMAP_MATCH_TIMEOUT. Related methods: • `CaptureBitmap` • `GetBitmapCRC` • `SYS_CompareBitmap` • `VerifyBitmap` • `WaitBitmap`
OPT_BITMAP_MATCH_TIMEOUT	Classic Agent Open Agent	NUMBER The total time allowed for a bitmap image to become stable. During the time period, Silk Test Classic takes multiple snapshots of the image, waiting the number of seconds specified with OPT_BITMAP_MATCH_TIMEOUT between snapshots. If the value returned by OPT_BITMAP_MATCH_TIMEOUT is reached before the number of bitmaps specified by OPT_BITMAP_MATCH_COUNT match, Silk Test Classic stops taking snapshots and raises the exception E_BITMAP_NOT_STABLE. Related methods: • `CaptureBitmap` • `GetBitmapCRC` • `VerifyBitmap`

Agent Option	Agent Supported	Description
		• `WaitBitmap`
OPT_BITMAP_PIXEL_TOLERANCE	Classic Agent	INTEGER
	Open Agent	The number of pixels of difference below which two bitmaps are considered to match. If the number of pixels that are different is smaller than the number specified with this option, the bitmaps are considered identical. The maximum tolerance is 32767 pixels. Related methods: • `SYS_CompareBitmap` • `VerifyBitmap` • `WaitBitmap`
OPT_CLASS_MAP	Classic Agent	LIST OF STRING
	Open Agent	The class mapping table for custom objects, with each entry in the list in the form `custom_class = standard_class`.
OPT_CLOSE_CONFIRM_BUTTONS	Classic Agent	LIST OF STRING
	Open Agent	The list of buttons used to close confirmation dialog boxes, which are dialog boxes that display when closing windows with the methods `Close`, `CloseWindows`, and `Exit`.
OPT_CLOSE_DIALOG_KEYS	Classic Agent	LIST OF STRING
	Open Agent	The keystroke sequence used to close dialog boxes with the methods `Close`, `CloseWindows`, and `Exit`.
OPT_CLOSE_MENU_NAME	Classic Agent	STRING A list of strings representing the list of menu items on the system menu used to close windows with the methods `Close`, `CloseWindows`, and `Exit`. Default is Close.
OPT_CLOSE_WINDOW_BUTTONS	Classic Agent	LIST OF STRING
	Open Agent	The list of buttons used to close windows with the methods `Close`, `CloseWindows`, and `Exit`.

Agent Option	Agent Supported	Description
OPT_CLOSE_WINDOW_MENUS	Classic Agent Open Agent	LIST OF STRING The list of menu items used to close windows with the methods `Close`, `CloseWindows`, and `Exit`.
OPT_CLOSE_WINDOW_TIMEOUT	Classic Agent Open Agent	NUMBER The number of seconds that Silk Test Classic waits before it tries a different close strategy for the `Close` method when the respective window does not close. Close strategies include **Alt+F4** or sending the keys specified by OPT_CLOSE_DIALOG_KEYS. By default, this is 2.
OPT_COMPATIBLE_TAGS	Classic Agent	BOOLEAN TRUE to generate and operate on tags compatible with releases earlier than Release 2; FALSE to use the current algorithm. The current algorithm affects tags that use index numbers and some tags that use captions. In general, the current tags are more portable, while the earlier algorithm generates more platform-dependent tags.
OPT_COMPATIBILITY	Open Agent	STRING Enables you to use the behavior of the specified Silk Test Classic version for specific features, when the behavior of these features has changed in a later version. Example strings: • `12` • `11.1` • `13.0.1` By default, this option is not set.
OPT_COMPRESS_WHITESPACE	Classic Agent	BOOLEAN TRUE to replace all multiple consecutive white spaces with a single space for comparison of tags. FALSE (the default) to avoid replacing blank characters in this manner. This is intended to provide a way to match tags where the only difference is the number of white spaces between words.

Agent Option	Agent Supported	Description
		If at all possible, use "wildcard " instead of this option. This option can increase test time because of the increased time it takes for compressing of white spaces in both source and target tags. If Silk Test Classic processes an object that has many children, this option may result in increased testing times. The tag comparison is performed in two parts. The first part is a simple comparison; if there is a match, no further action is required. The second part is to compress consecutive white spaces and retest for a match. Due to the possible increase in test time, the most efficient way to use this option is to enable and disable the option as required on sections of the testing that is affected by white space. Do not enable this option to cover your entire test. Tabs in menu items are processed before the actual tags are compared. Do not modify the window declarations of frame files by adding tabs to any of the tags.
OPT_DROPDOWN_PICK_BEFORE_GET	Classic Agent	BOOLEAN TRUE to drop down the combo box before trying to access the content of the combo box. This is usually not needed, but some combo boxes only get populated after they are dropped down. If you are having problems getting the contents of a combo box, set this option to TRUE. Default is FALSE.
OPT_ENABLE_ACCESSIBILITY	Classic Agent Open Agent	BOOLEAN TRUE to enable Accessibility when you are testing a Win32 application and Silk Test Classic cannot recognize objects. Accessibility is designed to enhance object recognition at the class level. FALSE to disable Accessibility. **Note:** For Mozilla Firefox and Google Chrome, Accessibility is always activated and cannot be deactivated.

Agent Option	Agent Supported	Description
		Default is FALSE.
OPT_ENSURE_ACTIVE_WINDOW	Open Agent	BOOLEAN TRUE ensures that the main window of the call is active before a call is executed. By default, this is FALSE.
OPT_EXTENSIONS	Classic Agent	LIST OF STRING The list of loaded extensions. Each extension is identified by the name of the .dll or .vxx file associated with the extension. Unlike the other options, OPT_EXTENSIONS is read-only and works only with `GetOption`().
OPT_GET_MULTITEXT_KEEP_EMPTY_LINES	Classic Agent	BOOLEAN TRUE returns an empty list if no text is selected. FALSE removes any blank lines within the selected text. By default, this is TRUE.
OPT_ITEM_RECORD	Open Agent	BOOLEAN For SWT applications, TRUE records methods that invoke tab items directly rather than recording the tab folder hierarchy. For example, you might record `SWTControls.SWTTabControl1.TabFolder.Select()`. If this option is set to FALSE, SWT tab folder actions are recorded. For example, you might record `SWTControls.SWTTabControl1.Select("TabFolder")`. By default, this is TRUE.
OPT_KEYBOARD_DELAY	Classic Agent Open Agent	NUMBER Default is 0.02 seconds; you can select a number in increments of .001 from .001 to up to 1000 seconds. Be aware that the optimal number can vary, depending on the application that you are testing. For example, if you are testing a Web application, a setting of .001 radically slows down the browser. However, setting this to 0 (zero) may cause basic application testing to fail.
OPT_KEYBOARD_LAYOUT	Classic Agent	STRING

Agent Option	Agent Supported	Description
		Provides support for international keyboard layouts in the Windows environment. Specify an operating-system specific name for the keyboard layout. Refer to the Microsoft Windows documentation to determine what string your operating system expects. Alternatively, use the `GetOption` method to help you determine the current keyboard layout, as in the following example: `Print (Agent.GetOption (OPT_KEYBOARD_LAYOUT))`
OPT_KILL_HANGING_APPS	Classic Agent Open Agent	BOOLEAN Specifies whether to shutdown the application if communication between the Agent and the application fails or times out. Set this option to TRUE when testing applications that cannot run multiple instances. By default, this is FALSE.
OPT_LOCATOR_ATTRIBUTES_CASE_SENSITIVE	Open Agent	BOOLEAN Set to `Yes` to add case-sensitivity to locator attribute names, or to `No` to match the locator names case insensitive.
OPT_MATCH_ITEM_CASE	Classic Agent Open Agent	BOOLEAN Set this option to TRUE to have Silk Test Classic consider case when matching items in combo boxes, list boxes, radio lists, and popup lists, or set this option to FALSE to ignore case differences during execution of a `Select` method. This option has no effect on a `Verify` function or a `VerifyContents` method.
OPT_MENU_INVOKE_POPUP	Classic Agent	STRING The command, keystrokes or mouse buttons, used to display pop-up menus, which are menus that popup over a particular object. To use mouse buttons, specify <button1>, <button2>, or <button3> in the command sequence.
OPT_MENU_PICK_BEFORE_GET	Classic Agent	BOOLEAN TRUE to pick the menu before checking whether an item on it exists,

Agent Option	Agent Supported	Description
		is enabled, or is checked, or FALSE to not pick the menu before checking. When TRUE, you may see menus pop up on the screen even though your script does not explicitly call the Pick method. Default is FALSE.
OPT_MOUSE_DELAY	Classic Agent Open Agent	NUMBER The delay used before each mouse event in a script. The delay affects moving the mouse, pressing buttons, and releasing buttons. By default, this is 0.02.
OPT_MULTIPLE_TAGS	Classic Agent Open Agent	BOOLEAN TRUE to use multiple tags when recording and playing back. FALSE to use one tag only, as done in previous releases. This option cannot be set through the **Agent Options** dialog box. Its default is TRUE and is only set by the INI file, option file, and through Agent.`SetOption`. This option overrides the **Record multiple tags** check box that displays in both the **Recorder Options** dialog box and the **Record Window Declaration Options** dialog box. If the **Record multiple tags** check box is grayed out and you want to change it, check this setting.
OPT_NO_ICONIC_MESSAGE_BOXES	Classic Agent	BOOLEAN TRUE to not have minimized windows automatically recognized as message boxes. Default is FALSE.
OPT_PAUSE_TRUELOG	Classic Agent	BOOLEAN TRUE to disable TrueLog at runtime for a specific portion of a script, or FALSE to enable TrueLog. This option has no effect if Truelog is not enabled. Default is FALSE.
OPT_PLAY_MODE	Classic Agent	STRING

Agent Option	Agent Supported	Description
		Used to specify playback mechanism. For additional information for Windows applications, see *Playing Back Mouse Actions*.
OPT_POST_REPLAY_DELAY	Classic Agent Open Agent	NUMBER The time in seconds to wait after invoking a function or writing properties. Increase this delay if you experience replay problems due to the application taking too long to process mouse and keyboard input. By default, this is 0.00.
OPT_RADIO_LIST	Classic Agent	BOOLEAN TRUE to view option buttons as a group; FALSE to use the pre-Release 2 method of viewing option buttons as individual objects.
OPT_RECORD_LISTVIEW_SELECT_BY_TYPEKEYS	Open Agent	BOOLEAN TRUE records methods with typekeys statements rather than with keyboard input for certain selected values. By default, this is FALSE.
OPT_RECORD_MOUSE_CLICK_RADIUS	Open Agent	INTEGER The number of pixels that defines the radius in which a mouse down and mouse up event must occur in order for the Open Agent to recognize it as a click. If the mouse down and mouse up event radius is greater than the defined value, a `PressMouse` and `ReleaseMouse` event are scripted. By default, this is set to 5 pixels.
OPT_RECORD_MOUSEMOVES	Classic Agent Open Agent	BOOLEAN TRUE records mouse moves for Web pages, Win32 applications, and Windows Forms applications that use mouse move events. You cannot record mouse moves for child domains of the xBrowser technology domain, for example Apache Flex and Swing. By default, this is FALSE.
OPT_RECORD_SCROLLBAR_ABSOLUT	Open Agent	BOOLEAN TRUE records scroll events with absolute values instead of relative to the previous scroll position. By default, this is FALSE.

Agent Option	Agent Supported	Description
OPT_REL1_CLASS_LIBRARY	Classic Agent	BOOLEAN TRUE to use pre-Release 2 versions of `GetChildren`, `GetClass`, and `GetParent`, or FALSE to use current versions.
OPT_REMOVE_FOCUS_ON_CAPTURE_TEXT	Open Agent	BOOLEAN TRUE to remove the focus from a window before text is captured. By default, this is FALSE.
OPT_REPLAY_HIGHLIGHT_TIME	Open Agent	NUMBER The number of seconds before each invoke command that the object is highlighted. By default, this is 0, which means that objects are not highlighted by default.
OPT_REPLAY_MODE	Classic Agent Open Agent	NUMBER The replay mode defines how replays on a control are executed: They can be executed with mouse and keyboard (low level) or using the API (high level). Each control defines which replay mode is the default mode for the control. When the default replay mode is enabled, most controls use a low level replay. The default mode for each control is the mode that works most reliably. If a replay fails, the user can change the replay mode and try again. Each control that supports that mode will execute the replay in the specified mode. If a control does not support the mode, it executes the default mode. For example, if PushButton supports low level replay but uses high level replay by default, it will use low level replay only if the option specifies it. Otherwise, it will use the high level implementation. Possible values include 0, 1, and 2. 0 is default, 1 is high level, 2 is low level. By default, this is 0.
OPT_REQUIRE_ACTIVE	Classic Agent	BOOLEAN Setting this option to FALSE allows 4Test statements to be attempted against inactive windows.

Agent Option	Agent Supported	Description
		Default is TRUE, except when running script statements that were recorded and are in a recording statement.
OPT_SCROLL_INTO_VIEW	Classic Agent	BOOLEAN TRUE to scroll a control into view before recording events against it or capturing its bitmap. This option applies only when OPT_SHOW_OUT_OF_VIEW is set to TRUE. This option is useful for testing Web applications in which dialog boxes contain scroll bars. This option applies only to HTML objects when you are using the DOM extension.
OPT_SET_TARGET_MACHINE	Classic Agent	STRING The IP address and port number to use for the target machine in distributed testing using the `SetOption` method. To set the target machine, type: `Agent.SetOption(OPT_SET_TARGET_MACHINE, < IPAddress >:< PortNumber >)` . **Note:** A colon must separate the IP address and the port number. To return the IP address and port number of the current target machine, type: `Agent.GetOption(OPT_SET_TARGET_MACHINE)`
OPT_SHOW_OUT_OF_VIEW	Classic Agent	BOOLEAN TRUE to have the agent see a control not currently scrolled into view; FALSE to have the Agent consider an out-of-view window to be invisible. This option applies only to HTML objects when you are using the DOM extension.
OPT_SYNC_TIMEOUT	Open Agent	NUMBER Specifies the maximum time in seconds for an object to be ready. **Note:** When you upgrade from a Silk Test version prior to Silk Test 13.0,

Agent Option	Agent Supported	Description
		and you had set the OPT_XBROWSER_SYNC_TIMEOUT option, the **Options** dialog box will display the default value of the OPT_SYNC_TIMEOUT, although your timeout is still set to the value you have defined.
OPT_TEXT_NEW_LINE	Classic Agent	STRING The keys to type to enter a new line using the `SetMultiText` method of the `TextField` class. The default value is "<Enter>".
OPT_TRANSLATE_TABLE	Classic Agent	STRING Specifies the name of the translation table to use. If a translation DLL is in use, the QAP_SetTranslateTable entry point is called with the string specified in this option.
OPT_TRIM_ITEM_SPACE	Classic Agent	BOOLEAN TRUE to trim leading and trailing spaces from items on windows, or FALSE to avoid trimming spaces.
OPT_USE_ANSICALL	Classic Agent	BOOLEAN If set to TRUE, each following DLL function is called as ANSI. If set to FALSE, which is the default value, UTF-8 DLL calls are used. For single ANSI DLL calls you can also use the ansicall keyword.
OPT_USE_SILKBEAN	Classic Agent	BOOLEAN TRUE to enable the agent to interact with the SilkBean running on a UNIX machine. Default is FALSE.
OPT_VERIFY_ACTIVE	Classic Agent Open Agent	BOOLEAN TRUE to verify that windows are active before interacting with them; FALSE to not check. See Active and Enabled Statuses for information about how this option affects Silk Test Classic methods. Default is TRUE, except when running script statements that were recorded and are in a recording statement.

Agent Option	Agent Supported	Description
OPT_VERIFY_APPREADY	Classic Agent	BOOLEAN TRUE to synchronize the agent with the application under test. Calls to the agent will not proceed unless the application is ready.
OPT_VERIFY_CLOSED	Classic Agent	BOOLEAN TRUE to verify that a window has closed. When FALSE, Silk Test Classic closes a window as usual, but does not verify that the window actually closed. Default is TRUE, except when running script statements that were recorded and are in a recording statement.
OPT_VERIFY_COORD	Classic Agent	BOOLEAN TRUE to check that coordinates passed to a method are inside the window before the mouse is pressed; FALSE to not check. Typically, you use the checking feature unless you need to be able to pass coordinates outside of the window, such as negative coordinates. If this option is set to TRUE and coordinates fall outside the window, Silk Test Classic raises the exception E_COORD_OUTSIDE_WINDOW.
OPT_VERIFY_CTRLTYPE	Classic Agent	BOOLEAN TRUE to check that objects are of the specified type before interacting with them; FALSE to not check. When TRUE, Silk Test Classic checks, for example, that an object that claims to be a listbox is actually a listbox. For custom objects, you must map them to the standard types to prevent the checking from signaling an exception, using the Silk Test Classic class map facility. Default is FALSE.
OPT_VERIFY_ENABLED	Classic Agent	BOOLEAN TRUE to verify that windows are enabled before interacting with them; FALSE to not check. For information about how this option affects various

Agent Option	Agent Supported	Description
		Silk Test Classic methods, see *Active and Enabled Statuses*.
OPT_VERIFY_EXPOSED	Classic Agent	BOOLEAN TRUE to verify that windows are exposed (that is, not covered, obscured, or logically hidden by another window) before interacting with them; FALSE to not check. Default is TRUE, except when running script statements that were recorded and are in a recording statement.
OPT_VERIFY_RESPONDING	Classic Agent	BOOLEAN Setting this option to FALSE suppresses "control not responding" errors.
OPT_VERIFY_UNIQUE	Classic Agent Open Agent	BOOLEAN TRUE to raise the E_WINDOW_NOT_UNIQUE exception upon encountering two or more windows with the same tag; FALSE to not raise the exception. When OPT_VERIFY_UNIQUE is FALSE, Silk Test Classic ignores the duplication and chooses the first window with that tag that it encounters. You can use a modified tag syntax to refer to a window with a non-unique tag, even when OPT_VERIFY_UNIQUE is TRUE. You can either include an index number after the object, as in myDialog("Cancel[2]"), or you can specify the window by including the text of a child that uniquely identifies the window, such as "myDialog/uniqueText/...", where the unique text is the tag of a child of that window.
OPT_WAIT_ACTIVE_WINDOW	Open Agent	NUMBER The number of seconds Silk Test Classic waits for a window to become active. If a window does not become active within the specified time, Silk Test Classic raises an exception. To require the Open Agent to check the active state of a window, set

Agent Option	Agent Supported	Description
		OPT_ENSURE_ACTIVE_WINDOW to TRUE. By default, OPT_WAIT_ACTIVE_WINDOW is set to 2 seconds.
OPT_WAIT_ACTIVE_WINDOW_RETRY	Open Agent	NUMBER The number of seconds Silk Test Classic waits for a window to become active before trying to verify again that the window is active. To require the Open Agent to retry the active state of an object, set OPT_ENSURE_ACTIVE_WINDOW to TRUE. By default, OPT_WAIT_ACTIVE_WINDOW_RETRY is set to 0.5 seconds.
OPT_WINDOW_MOVE_TOLERANCE	Classic Agent	INTEGER The number of pixels allowed for a tolerance when a moved window does not end up at the specified position. For some windows and GUIs, you cannot always move the window to the specified pixel. If the ending position is not exactly what was specified and the difference between the expected and actual positions is greater than the tolerance, Silk Test Classic raises an exception. On Windows, the tolerance can be set through the Control Panel, by setting the desktop window granularity option. If the granularity is zero, you can place a window at any pixel location. If the granularity is greater than zero, the desktop is split into a grid of the specified pixels in width, determining where a window can be placed. In general, the tolerance should be greater than or equal to the granularity.
OPT_WINDOW_RETRY	Classic Agent Open Agent	NUMBER The number of seconds Silk Test Classic waits between attempts to verify a window, if the window does not exist or is in the incorrect state. Silk Test Classic continues trying to

Agent Option	Agent Supported	Description
		find the window until the time specified with OPT_WINDOW_TIMEOUT is reached. The correct state of the window depends on various options. For example, Silk Test Classic might check whether a window is enabled, active, exposed, or unique, depending on the settings of the following options: • OPT_VERIFY_ENABLED • OPT_VERIFY_ACTIVE • OPT_VERIFY_EXPOSED • OPT_VERIFY_UNIQUE
OPT_WINDOW_SIZE_TOLERANCE	Classic Agent	INTEGER The number of pixels allowed for a tolerance when a resized window does not end at the specified size. For some windows and GUIs, you cant always resize the window to the particular size specified. If the ending size is not exactly what was specified and the difference between the expected and actual sizes is greater than the tolerance, Silk Test Classic raises an exception. On Windows, windows cannot be sized smaller than will fit comfortably with the menu bar.
OPT_WINDOW_TIMEOUT	Classic Agent Open Agent	NUMBER The number of seconds Silk Test Classic waits for a window to appear and be in the correct state. If a window does not appear within the specified timeout, Silk Test Classic raise an exception. The correct state of the window depends on various options. For example, Silk Test Classic might check whether a window is enabled, active, exposed, or unique, depending on the settings of the following options: • OPT_VERIFY_ENABLED • OPT_VERIFY_ACTIVE • OPT_VERIFY_EXPOSED

Agent Option	Agent Supported	Description
		• OPT_VERIFY_UNIQUE
OPT_WPF_CUSTOM_CLASSES	Open Agent	LIST OF STRING Specify the names of any WPF classes that you want to expose during recording and playback. For example, if a custom class called MyGrid derives from the WPF Grid class, the objects of the MyGrid custom class are not available for recording and playback. Grid objects are not available for recording and playback because the `Grid` class is not relevant for functional testing since it exists only for layout purposes. As a result, Grid objects are not exposed by default. In order to use custom classes that are based on classes that are not relevant to functional testing, add the custom class, in this case MyGrid, to the OPT_WPF_CUSTOM_CLASSES option. Then you can record, playback, find, verify properties, and perform any other supported actions for the specified classes.
OPT_WPF_PREFILL_ITEMS	Open Agent	BOOLEAN Defines whether items in a `WPFItemsControl`, like `WPFComboBox` or `WPFListBox`, are pre-filled during recording and playback. WPF itself lazily loads items for certain controls, so these items are not available for Silk Test Classic if they are not scrolled into view. Turn pre-filling on, which is the default setting, to additionally access items that are not accessible without scrolling them into view. However, some applications have problems when the items are pre-filled by Silk Test Classic in the background, and these applications can therefore crash. In this case turn pre-filling off.
OPT_XBROWSER_SYNC_MODE	Open Agent	STRING Configures the supported synchronization mode for HTML or AJAX. Using the HTML mode ensures that all HTML documents are in an interactive state. With this mode, you

Agent Option	Agent Supported	Description
		can test simple Web pages. If more complex scenarios with Java script are used, it might be necessary to manually script synchronization functions, such as `WaitForObject`, `WaitForProperty`, `WaitForDisappearance`, or `WaitForChildDisappearance`. Using the AJAX mode eliminates the need to manually script synchronization functions. By default, this value is set to AJAX.
OPT_XBROWSER_SYNC_TIMEOUT	Open Agent	NUMBER Specifies the maximum time in seconds for an object to be ready. **Note:** Deprecated. Use the option OPT_SYNC_TIMEOUT instead.
OPT_XBROWSER_SYNC_EXCLUDE_URLS	Open Agent	STRING Specifies the URL for the service or Web page that you want to exclude during page synchronization. Some AJAX frameworks or browser applications use special HTTP requests, which are permanently open in order to retrieve asynchronous data from the server. These requests may let the synchronization hang until the specified synchronization timeout expires. To prevent this situation, either use the HTML synchronization mode or specify the URL of the problematic request in the Synchronization exclude list setting. Type the entire URL or a fragment of the URL, such as `http://test.com/timeService` or `timeService`.

Setting the Default Agent

Silk Test Classic automatically assigns a default agent to your project or scripts. When you create a new project, the agent currently selected in the toolbar is the default agent. Silk Test Classic automatically starts the default agent when you open a project or create a new project. You can configure Silk Test Classic to automatically connect to the Open Agent or the Classic Agent by default.

To set the default agent, perform one of the following:

- Click **Options** > **Runtime** and set the default agent in the **Runtime Options** dialog box.
- Click the appropriate agent icon in the toolbar.

When you enable extensions, set the recovery system, configure the application, or record a test case, Silk Test Classic uses the default agent. When you run a test, Silk Test Classic automatically connects to the appropriate agent. Silk Test Classic uses the window declaration, locator, or `Find` or `FindAll` command to determine which agent to use.

Setting the Default Agent Using the Runtime Options Dialog Box

To set the default agent using the **Runtime Options** dialog box:

1. In the main menu, click **Options** > **Runtime**. The **Runtime Options** dialog box opens.
2. Select the agent that you want to use as the default from the **Default Agent** list box.
3. If you use the Classic Agent, select the type of network you want to use in the **Network** list box. If you select the Open Agent, TCP/IP is automatically selected.
4. If you use named agents, select the local agent name from the **Agent Name** list box. For instance, if your environment uses multiple agents or a port that uses a value other than the default, select the local agent.
5. Click **OK**.

   When you record a test case, Silk Test Classic automatically uses the default agent.

Setting the Default Agent Using the Toolbar Icons

From the main toolbar, click the following icons to set the default agent:

- to use the Classic Agent.
- to use the Open Agent.

Connecting to the Default Agent

Typically, the default agent starts automatically when it is needed by Silk Test Classic. However, you can connect to the default agent manually if it does not start or to verify that it has started.

To connect to the default Agent, from the main menu, click **Tools** > **Connect to Default Agent**.

The command starts the Classic Agent or the Open Agent on the local machine, depending on which agent is specified as the default in the **Runtime Options** dialog box. If the Agent does not start within 30 seconds, a message is displayed. If the default Agent is configured to run on a remote machine, you must connect to it manually.

Creating a Script that Uses Both Agents

You can create a script that uses the Classic Agent and the Open Agent. Recording primarily depends on the default agent while replaying the script primarily depends on the window declaration of the underlying control. If you create a script that does not use window declarations, the default agent is used to replay the script.

1. Set the default agent to the Classic Agent.
2. In the **Basic Workflow** bar, enable extensions for the application automatically.
3. In the **Basic Workflow** bar, click **Record Testcase** and record your test case.

4. When prompted, click **Paste to Editor** and then click **Paste testcase and update window declaration(s)**.
5. Click **OK**. The frame now contains window declarations from the Classic Agent.
6. Click **File** > **Save** to save the test case.
7. Type a name for the file into the **File name** field and click **Save**.
8. Set the default agent to the Open Agent.
9. Click **Options** > **Application Configurations**. The **Edit Application Configurations** dialog box opens.
10. Click **Add**.

    The **Select Application** dialog box opens.
11. Configure a standard or Web site test configuration.
12. Click **OK**.
13. Click **Record Testcase** in the **Basic Workflow** bar and record your test case.
14. When prompted, click **Paste to Editor** and then click **Paste testcase and update window declaration(s)**. The frame now contains window declarations from both the Classic Agent and the Open Agent. Silk Test Classic automatically detects which agent is required for each test based on the window declaration and changes the agent accordingly.
15. Click **File** > **Save** to save the test case.
16. Click **Run Testcase** in the **Basic Workflow** bar to replay the test case. Silk Test Classic automatically recognizes which agent to use based on the underlying window declarations.

You can also use the function `Connect([sMachine, sAgentType])` in a script to connect a machine explicitly with either the Classic Agent or the Open Agent. Using the connect function changes the default agent temporarily for the current test case, but it does not change the default agent of your project. However, this does not override the agent that is used for replay, which is defined by the window declaration.

Overview of Record Functionality Available for the Silk Test Agents

The Open Agent provides the majority of the same record capabilities as the Classic Agent and the same replay capabilities.

The following table lists the record functionality available for each Silk Test agent.

Record Command	Classic Agent	Open Agent
Window Declarations	Supported	Supported
Application State	Supported	Supported
Testcase	Supported	Supported
Actions	Supported	Supported
Window Identifiers	Supported	Not Supported
Window Locations	Supported	Not Supported
Window Locators	Not Supported	Supported
Class/Scripted	Supported	Not Supported

Record Command	Classic Agent	Open Agent
Class/ Accessibility	Supported	Not Supported
Method	Supported	Not Supported
Defined Window	Supported	Not Supported

Note: Silk Test Classic determines which agent to use by detecting whether a locator or `Find` or `FindAll` command is used. If a locator or `Find` or `FindAll` command is present, Silk Test Classic uses the Open Agent. As a result, you do not need to record window declarations for the Open Agent. For calls that use window declarations, the agent choice is made based on the presence or absence of the locator keyword and on the presence or absence of `TAG_IS_OPEN_AGENT` in a tag or multitag. When a window declaration contains both locators and tags and either could be used for resolving the window, check or uncheck the **Prefer Locator** check box in the **General Options** dialog box to determine which method is used.

Setting the Window Timeout Value to Prevent Window Not Found Exceptions

The window timeout value is the number of seconds Silk Test Classic waits for a window to display. If the window does not display within that period, the Window not found exception is raised. For example, loading an Apache Flex application and initializing the Apache Flex automation framework may take some time, depending on the machine on which you are testing and the complexity of your Apache Flex application. In this case, setting the Window timeout value to a higher value enables your application to fully load.

If you suspect that Silk Test Classic is not waiting long enough for a window to display, you can increase the window timeout value in the following ways:

- Change the window timeout value on the **Timing** tab of the **Agent Options** dialog box.
- Manually add a line to the script.

If the window is on the screen within the amount of time specified in the window timeout, the tag for the object might be the problem.

Manually Setting the Window Timeout Value

In some cases, you may want to increase the window timeout value for a specific test, rather than for all tests in general. For example, you may want to increase the timeout for Flex application tests, but not for browser tests.

1. Open the test script.
2. Add the following to the script: `Agent.SetOption (OPT_WINDOW_TIMEOUT, numberOfSeconds)`.

Setting the Window Timeout Value in the Agent Options Dialog Box

To change the window timeout value in the **Agent Options** dialog box:

1. Click **Options** > **Agent**.
2. Click the **Timing** tab.
3. Type the value into the **Window timeout** text box.

The value should be based on the speed of the machine, on which you are testing, and the complexity of the application that you are testing. By default, this value is set to 5 seconds. For example, loading and initializing complex Flex applications generally requires more than 5 seconds.

4. Click **OK**.

Basic Workflow for the Classic Agent

The **Basic Workflow** bar guides you through the process of creating a test case. To create and execute a test case, click each icon in the workflow bar to perform the relevant procedures. The procedures and the appearance of the workflow bar differ depending on whether your test uses the Open Agent or the Classic Agent.

The **Basic Workflow** bar is displayed by default. You can display it or hide it by checking and un-checking the **Workflows** > **Basic** check box. If your test uses both the Open Agent and the Classic Agent, the **Basic Workflow** bar changes when you switch between the agents.

When you use the Classic Agent, the **Basic workflow** uses hierarchical object recognition to record and replay test cases that use window declarations to find and identify objects.

Creating a New Project

You can create a new project and add the appropriate files to the project, or you can have Silk Test Classic automatically create a new project from an existing file.

Since each project is a unique testing environment, by default new projects do not contain any settings, such as extensions, class mappings, or Agent options. If you want to retain the settings from your current test set, save them as an options set by opening Silk Test Classic and clicking **Options** > **Save New Options Set**. You can add the options set to your project.

To create a new project:

1. In Silk Test Classic, click **File** > **New Project**, or click **Open Project** > **New Project** on the basic workflow bar.
2. On the **Create Project** dialog box, type the **Project Name** and **Description**.
3. Click **OK** to save your project in the default location, `C:\Users\<Current user>\Documents\Silk Test Classic Projects`.

   To save your project in a different location, click **Browse** and specify the folder in which you want to save your project.

   Silk Test Classic creates a `<Project name>` folder within this directory, saves the `projectname.vtp` and `projectname.ini` to this location and copies the extension .ini files, which are `appexpex.ini`, `axext.ini`, `domex.ini`, and `javaex.ini`, to the `extend` subdirectory. If you do not want to save your project in the default location, click **Browse** and specify the folder in which you want to save your project. Silk Test Classic then creates your project and displays nodes on the **Files** and **Global** tabs for the files and resources associated with this project.
4. Perform one of the following steps:
   - If your test uses the Open Agent, configure the application to set up the test environment.
   - If your test uses the Classic Agent, enable the appropriate extensions to test your application.

Enabling Extensions Automatically Using the Basic Workflow

An extension is a file that serves to extend the capabilities of, or data available to, a more basic program. Silk Test Classic provides extensions for testing applications that use non-standard controls in specific development and browser environments.

If you are testing a generic project that uses the Classic Agent, perform the following procedure to enable extensions:

1. Start the application or applet for which you want to enable extensions.
2. Start Silk Test Classic and make sure the basic workflow bar is visible. If it is not, click **Workflows** > **Basic** to enable it.

   If you do not see **Enable Extensions** on the workflow bar, ensure that the default agent is set to the Classic Agent.
3. If you are using Silk Test Classic projects, click **Project** and open your project or create a new project.
4. Click **Enable Extensions**.

   You cannot enable extensions for Silk Test Classic (`partner.exe`), the Classic Agent (`agent.exe`), or the Open Agent (`openAgent.exe`).
5. Select your test application from the list on the **Enable Extensions** dialog box, and then click **Select**.
6. If your test application does not display in the list, click **Refresh**. Or, you may need to add your application to this list in order to enable its extension.
7. Click **OK** on the **Extension Settings** dialog box, and then close and restart your application.
8. If you are testing an applet, the **Enable Applet Support** check box is checked by default.
9. When the **Test Extension Settings** dialog box opens, restart your application in the same way in which you opened it; for example, if you started your application by double-clicking the .exe, then restart it by double-clicking the .exe.
10. Make sure the application has finished loading, and then click **Test**. When the test is finished, a dialog box displays indicating that the extension has been successfully enabled and tested. You are now ready to begin testing your application or applet. If the test fails, review the troubleshooting topics.

When you enable extensions, Silk Test Classic adds an include file based on the technology or browser type that you enable to the **Use files location** in the **Runtime Options** dialog box.

Setting the Recovery System for the Classic Agent

The recovery system ensures that each test case begins and ends with the application in its intended state. Silk Test Classic refers to this intended application state as the BaseState. The recovery system allows you to run tests unattended. When your application fails, the recovery system restores the application to the BaseState, so that the rest of your tests can continue to run unattended.

If you are testing an application that uses both the Classic Agent and the Open Agent, set the Agent that will start the application as the default Agent and then set the recovery system. If you use the Open Agent to start the application, set the recovery system for the Open Agent.

1. Make sure the application that you are testing is running.
2. Click **Set Recovery System** on the **Basic Workflow** bar. If the workflow bar is not visible, click **Workflows** > **Basic** to enable it.
3. From the **Application** list, click the name of the application that you are testing.

   All open applications that are not minimized are listed. This list is dynamic and will update if you open a new application. If you are connected to the Open Agent, only those applications that have extensions enabled display in the list.

   Note: If you selected a non-web application as the application:

   - The **Command line** text box displays the path to the executable (`.exe`) for the application that you have selected.
   - The **Working directory** text box displays the path of the application you selected.

   If you selected a web application, the **Start testing on this page** text box displays the URL for the application you selected. If an application displays in the list, but the URL does not display in this text

box, your extensions may not be enabled correctly. Click **Enable Extensions** in the **Basic Workflow** bar to automatically enable and test extension settings.

4. *Optional:* In the **Frame file name** text box, modify the frame file name and click **Browse** to specify the location in which you want to save this file.

   Frame files must have a .inc extension. By default, this field displays the default name and path of the frame file you are creating. The default is `frame.inc`. If `frame.inc` already exists, Silk Test Classic appends the next logical number to the new frame file name; for example, `frame1.inc`.
5. *Optional:* In the **Window name** text box, change the window name to use a short name to identify your application.
6. Click **OK**.
7. Click **OK** when the message indicating that the recovery system is configured displays.
8. A new 4Test include file, `frame.inc`, opens in the Silk Test Editor. Click the plus sign in the file to see the contents of the frame file.
9. Record a test case.

Recording a Test Case With the Classic Agent

When you record a test case with the Classic Agent, Silk Test Classic uses hierarchical object recognition, a fast, easy method to create scripts. However, test cases that use dynamic object recognition are more robust and easy to maintain. You can create tests for both dynamic and hierarchical object recognition in your test environment. Use the method best suited to meet your test requirements. You can use both recognition methods within a single test case if necessary.

1. Enable extensions and set up the recovery system.
2. Click **Record Testcase** on the Basic Workflow bar. If the workflow bar is not visible, click **Workflows** > **Basic** to enable it.
3. Type the name of your test case in the **Testcase name** text box of the **Record Testcase** dialog box.

   Test case names are not case sensitive; they can be any length and consist of any combination of alphabetic characters, numerals, and underscore characters.
4. Select **DefaultBaseState** in the **Application State** field to have the built-in recovery system restore the default BaseState before the test case begins executing. If you chose **DefaultBaseState** as the application state, the test case is recorded in the script file as: `testcase testcase_name ()`. If you chose another application state, the test case is recorded as: `testcase testcase_name () appstate appstate_name`.
5. If you do not want Silk Test Classic to display the status window it normally shows during playback when driving the application to the specified base state—perhaps because the status bar obscures a critical control in the application you are testing—uncheck the **Show AppState status window** check box.
6. Click **Start Recording**. Silk Test Classic:

   - Closes the **Record Testcase** dialog box.
   - Starts your application, if it was not already running.
   - Removes the editor window from the display.
   - Displays the **Record Status** window.
   - Waits for you to take further action.
7. Interact with your application, driving it to the state that you want to test.

   As you interact with your application, Silk Test Classic records your interactions in the **Testcase Code** field of the **Record Testcase** dialog box, which is not visible.
8. To review what you have recorded, click **Done** in the **Record Status** window. Silk Test Classic displays the **Record Testcase** dialog box, which contains the 4Test code that has been recorded for you.

9. To resume recording your interactions, click **Resume Recording** in the dialog box. To temporarily suspend recording, click **Pause Recording** on the **Record Status** window.
10. Verify the test case.

Running a Test Case

When you run a test case, Silk Test Classic interacts with the application by executing all the actions you specified in the test case and testing whether all the features of the application performed as expected.

Silk Test Classic always saves the suite, script, or test plan before running it if you made any changes to it since the last time you saved it. By default, Silk Test Classic also saves all other open modified files whenever you run a script, suite, or test plan. To prevent this automatic saving of other open modified files, uncheck the **Save Files Before Running** check box in the **General Options** dialog box.

1. Make sure that the test case that you want to run is in the active window.
2. Click **Run Testcase** on the **Basic Workflow** bar.

   If the workflow bar is not visible, choose **Workflows** > **Basic** to enable it.

   Silk Test Classic displays the **Run Testcase** dialog box, which lists all the test cases contained in the current script.
3. Select a test case and specify arguments, if necessary, in the **Arguments** field.

   Remember to separate multiple arguments with commas.
4. To wait one second after each interaction with the application under test is executed, check the **Animated Run Mode (Slow-Motion)** check box.

   Typically, you will only use this check box if you want to watch the test case run. For instance, if you want to demonstrate a test case to someone else, you might want to check this check box. Executions of the default base state and functions that include one of the following strings are not delayed:

   - `BaseStateExecutionFinished`
   - `Connecting`
   - `Verify`
   - `Exists`
   - `Is`
   - `Get`
   - `Set`
   - `Print`
   - `ForceActiveXEnum`
   - `Wait`
   - `Sleep`
5. To view results using the TrueLog Explorer, check the **Enable TrueLog** check box. Click **TrueLog Options** to set the options you want to record.
6. Click **Run**. Silk Test Classic runs the test case and generates a results file.

   For the Classic Agent, multiple tags are supported. If you are running test cases using other agents, you can run scripts that use declarations with multiple tags. To do this, check the **Disable Multiple Tag Feature** check box in the **Agent Options** dialog box on the **Compatibility** tab. When you turn off multiple-tag support, 4Test discards all segments of a multiple tag except the first one.

Viewing Test Results

Whenever you run tests, a results file is generated which indicates how many tests passed and how many failed, describes why tests failed, and provides summary information.

1. Click **Explore Results** on the **Basic Workflow** or the **Data Driven Workflow** bars.
2. On the **Results Files** dialog box, navigate to the file name that you want to review and click **Open**.

By default, the results file has the same name as the executed script, suite, or test plan. To review a file in the TrueLog Explorer, open a `.xlg` file. To review a results file, open a `.res` file.

Troubleshooting Basic Workflow Issues

The following troubleshooting tips may help you with the basic workflow:

I restarted my application, but the Test button is not enabled

In order to enable the **Test** button on the **Test Extensions** dialog box, you must restart your application. Do not restart Silk Test Classic; restart the application that you selected on the **Enable Extensions** dialog box.

You must restart the application in the same manner. For example, if you are testing:

- A standalone Java application that you opened through a Command Prompt, make sure that you close and restart both the Java application and the Command Prompt window .
- A browser application or applet, make sure you return to the page that you selected on the **Enable Extensions** dialog box.
- An AOL browser application, make sure that you do not change the state of the application, for example resizing, or you may have issues with playback.

You can configure only one Visual Basic application at a time.

The test of my enabled Extension failed – what should I do?

If the test of your application fails, see *Troubleshooting Configuration Test Failures* for general information.

Migrating from the Classic Agent to the Open Agent

This section includes several useful topics that explain the differences between the Classic Agent and the Open Agent. If you plan to migrate from testing using the Classic Agent to the Open Agent, review this information to learn how to migrate your existing assets including window declarations and scripts.

Differences for Agent Options Between the Classic Agent and the Open Agent

Before you migrate existing Classic Agent scripts to the Open Agent, review the Agent Options listed below to determine if any additional action is required to facilitate the migration.

Agent Option	Action for Open Agent
OPT_AGENT_CLICKS_ONLY	Option not needed. **Note:** Use `OPT_REPLAY_MODE` for switching between high-level (API) clicks and low-level clicks.
OPT_CLOSE_MENU_NAME	Not supported by Open Agent.
OPT_COMPATIBLE_TAGS	Option not needed.
OPT_COMPRESS_WHITESPACE	Not supported by Open Agent.
OPT_DROPDOWN_PICK_BEFORE_GET	Option not needed. The Open Agent performs this action by default during replay.
OPT_EXTENSIONS	Option not needed.
OPT_GET_MULTITEXT_KEEP_EMPTY_LINES	Not supported by Open Agent.
OPT_KEYBOARD_LAYOUT	Not supported by Open Agent.
OPT_MENU_INVOKE_POPUP	No action. Pop-up menu handling using the Open Agent does not need such an option.
OPT_MENU_PICK_BEFORE_GET	Option not needed.
OPT_NO_ICONIC_MESSAGE_BOXES	Option not needed.
OPT_PLAY_MODE	Option not needed.
OPT_RADIO_LIST	Open Agent always sees `RadioList` items as individual objects.
OPT_REL1_CLASS_LIBRARY	Obsolete option.
OPT_REQUIRE_ACTIVE	Use the option `OPT_ENSURE_ACTIVE` instead.
OPT_SCROLL_INTO_VIEW	Option not needed. Open Agent only requires scrolling into view for low-level replay. By default, high-level replay is used, so no scrolling needs to be performed. However, `CaptureBitmap` never scrolls an object into view.
OPT_SET_TARGET_MACHINE	Option not needed.

Agent Option	Action for Open Agent
OPT_SHOW_OUT_OF_VIEW	Option not needed. Out-of-view objects are always recognized.
OPT_TEXT_NEW_LINE	Option not needed. The Open Agent always uses **Enter** to type a new line.
OPT_TRANSLATE_TABLE	Not supported by Open Agent.
OPT_TRAP_FAULTS	Fault trap is no longer active.
OPT_TRAP_FAULTS_FLAGS	Fault trap is no longer active.
OPT_TRIM_ITEM_SPACE	Option not needed. If required, use a * wildcard instead.
OPT_USE_ANSICALL	Not supported by Open Agent.
OPT_USE_SILKBEAN	SilkBean is not supported on the Open Agent.
OPT_VERIFY_APPREADY	Option not needed. The Open Agent performs this action by default.
OPT_VERIFY_CLOSED	Option not needed. The Open Agent performs this action by default.
OPT_VERIFY_COORD	Option not needed. The Open Agent does not typically check for native input in order to allow clicking outside of an object.
OPT_VERIFY_CTRLTYPE	Option not needed.
OPT_VERIFY_EXPOSED	Option not needed. The Open Agent performs this action when it sets a window to active. `OPT_ENSURE_ACTIVE_OBJECT_DEF` should yield the same result.
OPT_VERIFY_RESPONDING	Option not needed.
OPT_WINDOW_MOVE_TOLERANCE	Option not needed.

Differences in Object Recognition Between the Classic Agent and the Open Agent

When recording and executing test cases, the Classic Agent uses the keywords tag or multitag in a window declaration to uniquely identify an object in the test application. The tag is the actual name, as opposed to the identifier, which is the logical name.

When using the Open Agent, you typically use dynamic object recognition with a `Find` or `FindAll` function and an XPath query to locate objects in your test application. To make calls that use window declarations using the Open Agent, you must use the keyword locator in your window declarations. Similar to the tag or multitag keyword, the locator is the actual name, as opposed to the identifier, which is the logical name. This similarity facilitates a smooth transition of legacy window declarations, which use the Classic Agent, to dynamic object recognition, which leverages the Open Agent.

The following sections explain how to migrate the different tag types to valid locator strings.

Caption

Classic Agent tag “<caption string>”

Open Agent locator “//<class name>[@caption=’<caption string>’]”

Note: For convenience, you can use shortened forms for the XPath locator strings. Silk Test Classic automatically expands the syntax to use full XPath strings when you run a script.

You can omit:

- The hierarchy separator, “.//”. Silk Test Classic defaults to “//”.
- The class name. Silk Test Classic defaults to the class name of the window that contains the locator.
- The surrounding square brackets of the attributes, “[ ]”.
- The “@caption=” if the XPath string refers to the caption.

Note: Classic Agent removes ellipses (…) and ampersands (&) from captions. Open Agent removes ampersands, but not ellipses.

Example

Classic Agent:

```
CheckBox CaseSensitive
  tag “Case sensitive”
```

Open Agent:

```
CheckBox CaseSensitive
  locator “//CheckBox[@caption='Case sensitive']”
```

Or, if using the shortened form:

```
CheckBox CaseSensitive
  locator “Case sensitive”
```

Prior text

Classic Agent	tag “^Find What:”
Open Agent	locator “//<class name>[@priorlabel=’Find What:’]”

Note: Only available for Windows API-based and Java Swing applications. For other technology domains, use the **Locator Spy** to find an alternative locator.

Index

Classic Agent	tag “#1”
Open Agent	Record window locators for the test application. The Classic Agent creates index values based on the position of controls, while the Open Agent uses the controls in the order provided by the operating system. As a result, you must record window locators to identify the current index value for controls in the test application.

Window ID

Classic Agent	tag “$1041”
Open Agent	locator “//<class name>[@windowid=’1041’]”

Location

Classic Agent	tag “@(57,75)”
Open Agent	not supported

Note: If you have location tags in your window declarations, use the **Locator Spy** to find an alternative locator.

Multitag

Classic Agent multitag “Case sensitive” “$1011”

Open Agent locator “//CheckBox[@caption=’Case sensitive’ or @windowid=’1011’]” ‘parent’ statement

No changes needed. Multitag works the same way for the Open Agent.

Differences in the Classes Supported by the Open Agent and the Classic Agent

The Classic Agent and the Open Agent differ slightly in the types of classes that they support. These differences are important if you want to manually script your test cases. Or, if you are testing a single test environment with both the Classic Agent and the Open Agent. Otherwise, the Open Agent provides the majority of the same record capabilities as the Classic Agent and the same replay capabilities.

Windows-based applications

Both Agents support testing Windows API-based client/server applications. The Open Agent classes, functions, and properties differ slightly from those supported on the Classic Agent for Windows API-based client/server applications.

Classic Agent	Open Agent
AnyWin	AnyWin
AgentClass (Agent)	AgentClass (Agent)
CheckBox	CheckBox
ChildWin	<no corresponding class>
ClipboardClass (Clipboard)	ClipboardClass (Clipboard)
ComboBox	ComboBox
Control	Control
CursorClass (Cursor)	CursorClass (Cursor)
CustomWin	CustomWin
DefinedWin	<no corresponding class>
DesktopWin (Desktop)	DesktopWin (Desktop)
DialogBox	DialogBox
DynamicText	<no corresponding class>
Header	HeaderEx
ListBox	ListBox
ListView	ListViewEx
MainWin	MainWin
Menu	Menu
MenuItem	MenuItem
MessageBoxClass	<no corresponding class>

Classic Agent	Open Agent
MoveableWin	MoveableWin
PageList	PageList
PopupList	ComboBox
PopupMenu	<no corresponding class>
PopupStart	<no corresponding class>
PopupSelect	<no corresponding class>
PushButton	PushButton
RadioButton	**Note:** Items in Radiolists are recognized as RadioButtons on the CA. OA only identifies all of those buttons as RadioList.
RadioList	RadioList
Scale	Scale
ScrollBar	ScrollBar, VerticalScrollBar, HorizontalScrollBar
StaticText	StaticText
StatusBar	StatusBar
SysMenu	<no corresponding class>
Table	TableEx
TaskbarWin (Taskbar)	<no corresponding class>
TextField	TextField
ToolBar	ToolBar Additionally: PushToolItem, CheckBoxToolItem
TreeView, TreeViewEx	TreeView
UpDown	UpDownEx

The following core classes are supported on the Open Agent only:

- CheckBoxToolItem
- DropDownToolItem
- Group
- Item
- Link
- MonthCalendar
- Pager
- PushToolItem
- RadioListToolItem
- ToggleButton
- ToolItem

Web-based Applications

Both Agents support testing Web-based applications. The Open Agent classes, functions, and properties differ slightly from those supported on the Classic Agent for Windows API-based client/server applications.

Classic Agent	Open Agent
Browser	BrowserApplication
BrowserChild	BrowserWindow
HtmlCheckBox	DomCheckBox
HtmlColumn	<no corresponding class>
HtmlComboBox	<no corresponding class>
HtmlForm	DomForm
HtmlHeading	<no corresponding class>
HtmlHidden	<no corresponding class>
HtmlImage	<no corresponding class>
HtmlLink	DomLink
HtmlList	<no corresponding class>
HtmlListBox	DomListBox
HtmlMarquee	<no corresponding class>
HtmlMeta	<no corresponding class>
HtmlPopupList	DomListBox
HtmlPushButton	DomButton
HtmlRadioButton	DomRadioButton
HtmlRadioList	<no corresponding class>
HtmlTable	DomTable
HtmlText	<no corresponding class>
HtmlTextField	DomTextField
XmlNode	<no corresponding class>
Xul* Controls	<no corresponding class>

Note: The `DomElement` class of the Open Agent enables you to access any element on an HTML page. If the Open Agent has no class associated with a specific class supported on the Classic Agent, you can use the `DomElement` class to access the controls in the class.

Java AWT/Swing Applications

Both Agents support testing Java AWT/Swing applications. The Open Agent classes, functions, and properties differ slightly from those supported on the Classic Agent for Windows API-based client/server applications.

Classic Agent	Open Agent
JavaApplet	AppletContainer
JavaDialogBox	AWTDialog, JDialog
JavaMainWin	AWTFrame, JFrame
JavaAwtCheckBox	AWTCheckBox

Classic Agent	Open Agent
JavaAwtListBox	AWTList
JavaAwtPopupList	AWTChoice
JavaAwtPopupMenu	<no corresponding class>
JavaAwtPushButton	AWTPushButton
JavaAwtRadioButton	AWTRadioButton
JavaAwtRadioList	<no corresponding class>
JavaAwtScrollBar	AWTScrollBar
JavaAwtStaticText	AWTLabel
JavaAwtTextField	AWTTextField, AWTTextArea
JavaJFCCheckBox	JCheckBox
JavaJFCCheckBoxMenuItem	JCheckBoxMenuItem
JavaJFCChildWin	<no corresponding class>
JavaJFCComboBox	JComboBox
JavaJFCImage	<no corresponding class>
JavaJFCListBox	JList
JavaJFCMenu	JMenu
JavaJFCMenuItem	JMenuItem
JavaJFCPageList	JTabbedPane
JavaJFCPopupList	JList
JavaJFCPopupMenu	JPopupMenu
JavaJFCProgressBar	JProgressBar
JavaJFCPushButton	JButton
JavaJFCRadioButton	JRadioButton
JavaJFCRadioButtonMenuItem	JRadioButtonMenuItem
JavaJFCRadioList	<no corresponding class>
JavaJFCScale	JSlider
JavaJFCScrollBar	JScrollBar, JHorizontalScrollBar, JVerticalScrollBar
JavaJFCSeparator	JComponent
JavaJFCStaticText	JLabel
JavaJFCTable	JTable
JavaJFCTextField	JTextField, JTextArea
JavaJFCToggleButton	JToggleButton
JavaJFCToolBar	JToolBar
JavaJFCTreeView	JTree

Classic Agent	Open Agent
JavaJFCUpDown	JSpinner

Java SWT/RCP Applications

Only the Open Agent supports testing Java SWT/RCP-based applications. For a list of the classes, see *Supported SWT Widgets for the Open Agent*.

Differences in the Parameters Supported by the Open Agent and the Classic Agent

The Classic Agent and the Open Agent differ slightly in the function parameters that they support. These differences are important if you want to manually script your test cases. Or, if you are testing a single test environment with both the Classic Agent and the Open Agent. Otherwise, the Open Agent provides the majority of the same record capabilities as the Classic Agent and the same replay capabilities.

For some parameters, the Open Agent uses a hard-coded default value internally. If one of these parameters is set in a 4Test script, the Open Agent ignores the value and uses the value listed here.

Function	Parameter	Classic Agent Value	Open Agent Value
AnyWin::PressKeys/ ReleaseKeys	nDelay	Any number.	0
AnyWin::PressKeys/ ReleaseKeys	sKeys	More than one key is supported.	Only one key is supported. The first key is used and the remaining keys are ignored. For example `MainWin.PressKeys( "<Shift><Left>")` will only press the **Shift** key. To press both keys, specify `MainWin.PressKeys( "<Shift>") MainWin.PressKeys( "<Left >")`.
AnyWin::TypeKeys	sEvents	Keystrokes to type or mouse buttons to press.	The Open Agent supports keystrokes only.
AnyWin::GetChildren	bInvisible	TRUE or FALSE.	FALSE.
AnyWin::GetChildren	bNoTopLevel	TRUE or FALSE.	FALSE.
TextField::GetFontName	iLine	The Classic Agent recognizes this parameter.	The Open Agent ignores this parameter.
AnyWin::GetCaption	bNoStaticText	TRUE or FALSE.	FALSE.
AnyWin::GetCaption, Control::GetPriorStatic	bRawMode	TRUE or FALSE.	FALSE. However, the returned strings include trailing and leading spaces, but ellipses, accelerators, and hot keys are removed.
PageList::GetContents/ GetPageName	bRawMode	TRUE or FALSE.	FALSE. However, the returned strings include trailing and leading spaces, ellipses, and hot keys but accelerators are removed.

Function	Parameter	Classic Agent Value	Open Agent Value
AnyWin::Click/ DoubleClick/ MoveMouse/ MultiClick/ PressMouse/ ReleaseMouse, PushButton::Click	bRawEvent	The Classic Agent recognizes this parameter.	The Open Agent ignores this value.

Overview of the Methods Supported by the Silk Test Classic Agents

The `winclass.inc` file includes information about which methods are supported for each Silk Test Classic Agent. The following 4Test keywords indicate Agent support:

supported_ca	Supported on the Classic Agent only.
supported_oa	Supported on the Open Agent only.

Standard 4Test methods, such as `AnyWin::GetCaption()`, can be marked with one of the preceding keywords. A method that is marked with the supported_ca or supported_oa keyword can only be executed successfully on the corresponding Agent. Methods that do not have a keyword applied will run on both Agents.

To find out which methods are supported on each Agent, open the .inc file, for instance `winclass.inc`, and verify whether the supported_ca or supported_oa keyword is applied to it.

Classic Agent

Certain functions and methods run on the Classic Agent only. When these are recorded and replayed, they default to the Classic Agent automatically. You can use these in an environment that uses the Open Agent. Silk Test Classic automatically uses the appropriate Agent. The functions and methods include:

- C data types for use in calling functions in DLLs.
- `ClipboardClass` methods.
- `CursorClass` methods.
- Certain SYS functions.

SYS Functions Supported by the Open Agent and the Classic Agent

The Classic Agent supports all SYS functions. The Open Agent supports all SYS functions with the exception of `SYS_GetMemoryInfo`. `SYS_GetMemoryInfo` defaults to the Classic Agent when a script is executed.

You can use the following SYS functions with the Open Agent or the Classic Agent.

SYS Function	Description
SYS_GetRegistryValue	With the Classic Agent, `SYS_GetRegistryValue` returns an incorrect value when a binary value is used. Use the Open Agent with `SYS_GetRegistryValue` to avoid this issue.

SYS Function	Description
SYS_FileSetPointer	When setting the pointer after the end of the file, the Open Agent does not throw an exception, while the Classic Agent does throw an exception.
SYS_IniFileGetValue	The Open Agent does not allow the ‘]’ character to be part of a section name, while the Classic Agent does allow it. Also, with the Open Agent, ‘=’ must not be part of a key name. The Classic Agent allows ‘=’ to be part of a key name, but produces incorrect results.

Note: Error messages and exceptions may differ between the Open Agent and the Classic Agent.

Silk Test Classic Projects

Silk Test Classic projects organize all the resources associated with a test set and present them visually in the **Project Explorer**, making it easy for you to see your test environment, and to manage it and work within it.

Silk Test Classic projects store relevant information about your project, including the following:

- References to all the resources associated with a test set, such as plans, scripts, data, options sets, .ini files, results, frame files, and include files.
- Configuration information.
- Editor settings.
- Data files for attributes and queries.

All of this information is stored at the project level, meaning that once you add the appropriate files to your project and configure it once, you may never need to do it again. Switching among projects is easy - since you need to configure the project only once, you can simply open the project and run your tests.

When you create a new project, Silk Test Classic automatically uses the agent that is selected in the toolbar.

Each project is a unique testing environment

By default, new projects do not contain any settings, such as enabled extensions, class mappings, or agent options. If you want to retain the settings from your current test set, save them as a options set by opening Silk Test Classic and clicking **Options** > **Save New Options Set**. You can include the options set when you create your project. You can create a project manually or ha automatically generate a project for you, based on existing files that you specify.

Storing Project Information

Silk Test Classic stores project-related information in the following project files:

`projectname.vtp`	The project file has a Verify Test Project (.vtp) extension and is organized as an .ini file. It stores the names and locations of files used by the project.
`projectname.ini`	The project initialization file, similar to the `partner.ini` file, stores information about options sets, queries, and other resources included in your project.
`SilkTestClassic.ini`	A user-specific initialization file that stores user-specific information about the location of the last projects, the size of the project history, and the location of the current project.

These files are created in the `projectname` folder. When you create your project, Silk Test Classic prompts you to store your project in the default location `C:\Users\<Current user>\Documents \Silk Test Classic Projects`. Silk Test Classic creates a `<Project name>` folder within this directory, saves the `projectname.vtp` and `projectname.ini` to this location and copies the extension .ini files, which are `appexpex.ini`, `axext.ini`, `domex.ini`, and `javaex.ini`, to the `extend` subdirectory. If you do not want to save your project in the default location, click **Browse** and specify the folder in which you want to save your project.

When you export a project, the default location is the project directory.

Note: The extension .ini files, which are `appexpex.ini`, `axext.ini`, `domex.ini`, and `javaex.ini`, located in your `<Silk Test Classic installation directory>\extend` folder are copied to the `extend` directory of your project, regardless of what extension you have enabled. Do not rename the `extend` directory; this directory must exist in order for Silk Test Classic to open your project.

You can have Silk Test Classic automatically enable the appropriate extension using the basic workflow bar, or you can manually enable extensions. The current project uses the extension options in the extension .ini file copied to the `extend` directory of your project. Any modifications you make to the options for this enabled extension will be saved to the copy stored within the current project in the `extend` directory.

The `extend` directory is used only for local testing on the host machine. If you want to test on remote agent machines, you must copy the .ini files from the `extend` directory of your project to the `extend` directory on the target machines.

File references

Whether you are emailing, packaging, or adding files to a project, it is important to understand how Silk Test Classic stores the path of the file. The .vtp files of Silk Test Classic use relative paths for files on the same root drive and absolute paths for files with different root drives. The use of relative and absolute file paths is not configurable and cannot be overridden. If you modify the .vtp file to change file references from relative paths to absolute paths, the next time you open and close the project it will have relative paths and your changes will be lost.

Accessing Files Within Your Project

Working with Silk Test Classic projects makes it easy to access your files - once you have added a file to your project, you can open it by double-clicking it in the **Project Explorer**. The **Project Explorer** contains the following two tabs:

Tab	Description
Files	Lists all of the files included in the project. From the **Files** tab, you can view, edit, add, and remove files from the project, as well as right-click to access menu options for each of the file types. From the **Files** tab, you can also add, rename, remove and work with folders within each category.
Global	Displays all the resources that are defined at a global level within the project's files. For example test cases, functions, classes, window declarations, and others. When you double-click an object on the **Global** tab, the file in which the object is defined opens and your cursor displays at the beginning of the line in which the object is defined. You can run and debug test cases and application states from the **Global** tab. You can also sort the elements that display within the folders on the **Global** tab.

Existing test sets do not display in the **Project Explorer** by default; you must convert them into projects.

Sharing a Project Among a Group

Apply the following guidelines to share a Silk Test Classic project among a group:

- Create the project in the location from which it will be shared. For example, you can create the project on a network drive.
- Ensure that testers create the same directory structure on their machines.

Project Explorer

Use the **Project Explorer** to view and work with all the resources within a Silk Test Classic project. You can access the **Project Explorer** by clicking:

- **File** > **Open Project** and specifying the project you want to open.
- **File** > **New Project** and creating a new project.
- **Project** > **View Explorer**, if you currently have a project open and do not have the **Project Explorer** view on.
- **Project** > **New Project** or **Open Project** on the **Basic Workflow** bar.

The resources associated with the project are grouped into categories. You can easily navigate among and access all of these resources using the **Files** and **Global** tabs. When you double-click a file on the **Files** tab, or an object on the **Global** tab, the file opens in the right pane. You can drag the divider to adjust the size of the **Project Explorer** windows and click **Project** > **Align** to change the orientation of the tabs from left to right.

Files tab

The **Files** tab lists all of the files that have been added to the project. The file name displays first, followed by the path. If files exist on a network drive, they are referenced using Universal Naming Conventions (UNC). Files are grouped into the following categories:

Category	Description
Profile	Contains project-specific initialization files, such as the `projectname.ini` and option sets files, which means .opt files, that are associated with the project.
Script	Contains test scripts, which means .t and .g.t files, that are associated with the project.
Include/Frame	Contains include files, which means .inc files, and frame/object files that are associated with the project.
Plan	Contains test plans and suite files, which means .pln and .s files, that are associated with the project.
Results	Contains results, which means .res and .rex files, that are associated with the project.
Data	Contains data associated with the project, such as Microsoft Word documents, text files, bitmaps, and others. Double-click the file to open it in the appropriate application. You must open files that are not associated with application types in the Windows Registry using the **File/Open** dialog box.

From the **Files** tab, you can view, edit, add, remove and work with files within the project. For example, to add a file to the project, right-click the category name, for example Script, and then click **Add File**. After you have added the file, you can right-click the file name to view options for working with the file, such as record test case and run test case. Silk Test Classic functionality has not changed - it is now accessible through a project.

You can work with the folders within the categories on the **Files** tab, by adding, renaming, moving, and deleting folders within each category.

Global tab

The **Global** tab lists resources that are defined at a global level within the entire project. The resource name displays first, followed by the file in which it is defined. Resources contained within the project's files are grouped into the following categories:

- Records

- Classes
- Enums
- Window Declarations
- Testcases
- Appstates
- Functions
- Constants

From the **Global** tab, you can go directly to the location in which a global object or resource is defined. Double-click any object within the folders to go to the location in which the object is defined. Silk Test Classic opens the file and positions your cursor at the beginning of the line in which the object is defined.

You can also run and debug test cases and application states by right-clicking a test case or application state, and then selecting the appropriate option. For example, right-click a test case within the `Testcase` folder and then click **Run**. Silk Test Classic opens the file containing the test case you selected, and displays the **Run Testcase** dialog box with the selected test case highlighted. You can input argument values and run or debug the test case.

On the **Global** tab, you can sort the resources within each node by resource name, file name, or file date.

Note: Methods and properties are not listed on the **Global** tab since they are specific to classes or window declarations. You can access methods and properties by double-clicking the class or window declaration in which they are defined.

You cannot move files within the **Project Explorer**. For example, you cannot drag a script file under the **Frame** file node. However, you can drag the file to another folder within the same category node.

Note: If you change the location or name of a file included in your project, outside of Silk Test Classic, you must make sure the `projectname.vtp` contains the correct reference.

Creating a New Project

You can create a new project and add the appropriate files to the project, or you can have Silk Test Classic automatically create a new project from an existing file.

Since each project is a unique testing environment, by default new projects do not contain any settings, such as extensions, class mappings, or Agent options. If you want to retain the settings from your current test set, save them as an options set by opening Silk Test Classic and clicking **Options** > **Save New Options Set**. You can add the options set to your project.

To create a new project:

1. In Silk Test Classic, click **File** > **New Project**, or click **Open Project** > **New Project** on the basic workflow bar.
2. On the **Create Project** dialog box, type the **Project Name** and **Description**.
3. Click **OK** to save your project in the default location, `C:\Users\<Current user>\Documents\Silk Test Classic Projects`.

   To save your project in a different location, click **Browse** and specify the folder in which you want to save your project.

   Silk Test Classic creates a `<Project name>` folder within this directory, saves the `projectname.vtp` and `projectname.ini` to this location and copies the extension .ini files, which are `appexpex.ini`, `axext.ini`, `domex.ini`, and `javaex.ini`, to the `extend` subdirectory. If you do not want to save your project in the default location, click **Browse** and specify the folder in which you want to save your project. Silk Test Classic then creates your project and displays nodes on the **Files** and **Global** tabs for the files and resources associated with this project.
4. Perform one of the following steps:

- If your test uses the Open Agent, configure the application to set up the test environment.
- If your test uses the Classic Agent, enable the appropriate extensions to test your application.

Opening an Existing Project

You can open a Silk Test Classic project as well as open an archived Silk Test Classic project. You can also open a Silk Test Classic project or archived project through the command line.

To open an existing project:

1. Click **File** > **Open Project** or click **Open Project** > **Open Project** on the basic workflow bar.

   If you already have a project open, a dialog box opens informing you that the open project will be closed. If you associated Silk Test Classic file types with Silk Test Classic during installation, then you can open a Silk Test Classic project or package by double-clicking the .vtp or .stp file.
2. If you are opening a packaged Silk Test Classic project, which means an .stp file, you must perform the following steps:
   a) Indicate into what directory you want to unpack the project in the **Base path** text box. The files are unpacked to the directory you indicate in the **Base path** text box.
   b) Enter a password into the **Password** text box if the archived Silk Test Classic project was saved with a password.

      If you open a package by double-clicking the .stp file, the base path is the directory that contains the .stp file.

      When you select a location for unpacking the archive on the **Open Project** dialog box, Silk Test Classic uses that directory path, the base path, to substitute for the drive and root directory in the Use Path and Use Files paths.

      The **Base path** and **Password** text boxes are enabled only if you are opening an .stp file.
3. On the **Open Project** dialog box, specify the project that you want to open, and then click **Open**.

   If you open a project file (.vtp) by clicking **File** > **Open** command, the `projectname.vtp` file will open in the **4Test Editor**, but the project and its associated settings will not be loaded. Projects do not display in the recently opened files list. To close all open files within a project, click **Window** > **Close All**.

Converting Existing Tests to a Project

Since each project is a unique testing environment, by default new projects do not contain any settings, such as extensions, class mappings, or Agent options. If you want to retain the settings from your current test set, save them as an options set by clicking **Options** > **Save New Options Set**. You can include the options set when you create your project.

To convert existing test sets to a project:

1. Create a new project.
2. Manually add the files to the project.

Using Option Sets in Your Project

To use an options set within your project, you must make sure that the options set is loaded into memory. You can tell if an options set is loaded by looking at the Silk Test Classic title bar. If `filename.opt` displays in the title bar, then the options set `filename.opt` is loaded. If an options set is loaded, it overrides the settings contained in the `projectname.ini` file.

Note: When an options set is loaded, the context menu options are available only for the loaded options set; these menu options are grayed out for .ini and .opt files that are not loaded.

You can load an options set into your project using any of the following methods:

- If the options set is included in your project, within the **Profile** node on the **Files** tab, right-click the options set that you want to load and then click **Open Options Set**.
- Right-click **Save New Options Set** to load the options set and add it under the **Profile** node on the **Files** tab.
- Use the **Options** menu; click **Options** > **Open Options Set**, browse to the options set (.opt) that you want to load, and then click **Open**.
- Load the options set at runtime using the optionset keyword. This loads the options set at the point in the plan file in which the options set is called. All test cases that follow use this options set.

If an options set was loaded when you closed Silk Test Classic, Silk Test Classic automatically re-loads this options set when you re-start Silk Test Classic.

To include an options set in your project, you can add the options set by right-clicking **Profile** on the **Files** tab, clicking **Add File**, selecting the options set you want to add to the project, and then clicking **OK**. You can also click **Save New Options Set**; this loads the options set and adds it under the **Profile** node on the **Files** tab.

Editing an Options Set

To edit an options set in your project:

1. On the **Files** tab, expand the **Profile** node.
2. Right-click the options set that you want to edit and click **Open Options Set**. The options set is loaded into memory.
3. Right-click the options set that you want to edit again and select the type of option you want to edit.
   For example Runtime, Agent, Extensions, and others.
4. Modify your options and then click **OK**. Your current settings are changed and saved to the .opt file.

If you want to change settings for future use, double-click the options set that you want to edit on the **Files** tab. This opens the options file in the Editor without loading the options file into memory. Changes you make to the options set in the Editor will be saved, but will not take effect until you load the options set by selecting **Open Options Set** from the **Options** menu or the right-click shortcut.

Silk Test Classic File Types

Silk Test Classic uses the following types of files in the automated testing process, each with a specific function. The files marked with an * are required by Silk Test Classic to create and run test cases.

File Type	Extension	Description
Project	.vtp	Silk Test Classic projects organize all the resources associated with a test set and present them visually in the **Project Explorer**, making it easy to see, manage, and work within your test environment. The project file has a Verify Test Project (`.vtp`) extension and is organized as an `.ini` file; it stores the names and locations of files used by the project. Each project file also has an associated project initialization file: `projectname.ini`.
Exported project	.stp	A Silk Test Project (`.stp`) file is a compressed file that includes all the data that Silk Test Classic exports for a project. A file of this type is created when you click **File** > **Export Project**.

File Type	Extension	Description
		The `.stp` file includes the configuration files that are necessary for Silk Test Classic to set up the proper testing environment.
Testplan	.pln	An automated test plan is an outline that organizes and enhances the testing process, references test cases, and allows execution of test cases according to the test plan detail. It can be of type masterplan or of subplan that is referenced by a masterplan.
Test Frame*	.inc	A specific kind of include file that upon creation automatically captures a declaration of the AUT's main window including the URL of the Web application or path and executable name for client/server applications; acts as a central repository of information about the AUT; can also include declarations for other windows, as well as application states, variables, and constants.
4Test Script*	.t	Contains recorded and hand-written automated test cases, written in the 4Test language, that verify the behavior of the AUT.
Data driven Script	.g.t	Contains data-driven test cases that pull their data from databases.
4Test Include File	.inc	A file that contains window declarations, constants, variables, classes, and user defined functions.
Suite	.s	Allows sequential execution of several test scripts.
Text File	.txt	An ASCII file that can be used for the following: • Store data that will be used to drive a data driven test case. • Print a file in another document (Word) or presentation (PowerPoint). • Accompany your automation as a readme file. • Transform a tab-delimited plan into a Silk Test Classic plan.
Results File	.res	Is automatically created to store a history of results for a test plan or script execution.
Results Export File	.rex	A single compressed results file that you can relocate to a different machine. Click **Results** > **Export** to create a `.rex` file out of the existing results files of a project.
TrueLog File	.xlg	A file that contains the captured bitmaps and the logging information that is captured when TrueLog is enabled during a test case run.

Organizing Projects

This section includes the topics that are available for organizing projects.

Adding Existing Files to a Project

You can add existing files to a project or create new files to add to the project. We recommend adding all referenced files to your project so that you can easily see and access the files, and the objects contained within them. Referenced files do not have to be included in the project. Plans and scripts will continue to run, provided the paths that are referenced are accurate.

When you add a file to a project, project files (.vtp files) use relative paths for files on the same root drive and absolute paths for files with different root drives. The use of relative and absolute files is not configurable and cannot be overridden.

To add an existing file to a project:

1. If your project is not already open, click **File** > **Open Project** or click **Open Project** > **Open Project** on the basic workflow bar, select the project to which you want to add a file, and then click **Open**.
2. On the **Project Explorer**, select the **Files** tab, right-click the node associated with the type of file you want to add, and then click **Add File**.

For example, to add a script file to the project, right-click **Script**, and then click **Add File**.

3. On the **Add File to Project** dialog box, specify the file you want to add to the open project, and then click **Open**.

   The file name, followed by the path, displays under the appropriate category on the **Files** tab sorted alphabetically by name and is associated with the project through the `projectname.vtp` file. If files exist on a network drive, they are referenced using Universal Naming Conventions (UNC).

You can also add existing files to the project by clicking **Project** > **Add File**. Silk Test Classic automatically places the file in the appropriate node, based on the file type; for example if you add a file with a .pln extension, it will display under the **Plan** node on the **Files** tab. We do not recommend adding application .ini files or Silk Test Classic .ini files, which are `qaplans.ini`, `propset.ini`, and the `extension.ini` files, to your project. If you add object files, which are .to and .ino files, to your project, the files will display under the **Data** node on the **Files** tab. Objects defined in object files will not display in the **Global** tab. You cannot modify object files within the Silk Test Classic editor because object files are binary. To modify an object file, open the source file, which is a .t or .inc file, edit it, and then recompile.

Renaming Your Project

The `projectname.ini` and the `projectname.vtp` refer to each other; make sure the references are correct in both files when you rename your project.

To rename your project:

1. Make sure the project you want to rename is closed.
2. In Windows Explorer, locate the `projectname.vtp` and `projectname.ini` associated with the project name you want to change.
3. Change the names of `projectname.vtp` and `projectname.ini`. Make sure that you use the same projectname for both files.
4. In a text editor outside of Silk Test Classic, open `projectname.vtp`, change the reference to the `projectname.ini` file to the new name, and then save and close the file. Do not open the project in Silk Test Classic yet.
5. In a text editor outside of Silk Test Classic, open `projectname.ini`, change the reference to the `projectname.vtp` file to the new name, and then save and close the file.
6. In Silk Test Classic, open the project by clicking **File** > **Open Project** or **Open ProjectOpen Project** on the basic workflow bar. The new project name displays.

Working with Folders in a Project

In addition to working with files, you can also add your own folders to all nodes listed on the **File** tab of the **Project Explorer**. For example, the **Files** tab of the **Project Explorer** can include notes.

You can also right-click a folder and click the following:

- **Expand All** to display all contents of a folder.
- **Collapse All** to collapse the contents of the folder.
- **Display Full Path** to show the full path for the contents.
- **Display Date/Time** to show creation information for the content file.

Adding a Folder to the Files Tab of the Project Explorer

You may add a folder to any of the categories (nodes) on the **Files** tab of the **Project Explorer**. You may not add a folder to the root project folder, nor change the titles of the root nodes.

To add a folder to a project:

1. If your project is not already open, click **File** > **Open Project** or click **Open Project** > **Open Project** on the basic workflow bar. Select a project, then click **Open**.
2. In the **Project Explorer**, click the **Files** tab, right-click a folder and select **Add Folder**.
3. On the **Add Folder** dialog box, enter the name of the new folder, then click **OK**.

   When you are naming a folder, you may use alphanumeric characters, underscore character, character space, or hyphens. Folder names may be a maximum of 256 characters long. Creating folders with more than 256 characters is possible, but Silk Test Classic will truncate the name when you save the project. The concatenated length of the names of all folders within a project may not exceed 256 characters. You may not use periods or parentheses in folder names. Within a node, folder names must be unique.

Moving Files and Folders

You may move an individual file or files between folders within the same category on the **Files** tab of the **Project Explorer**. You cannot move the predefined Silk Test Classic folders (nodes) such as Profile Script, Plan, Frame, and Data.

You may also move sub-folders within the same category on the **Files** tab. You cannot move sub-folders across categories.

To move a folder or file:

1. If your project is not already open, click **File** > **Open Project** or click **Open Project** > **Open Project** on the basic workflow bar. Select a project, then click **Open**.
2. In the **Project Explorer**, click the **Files** tab. Click a file, a folder, or shift-click to select several files or folders, then drag the items to the new location.
3. Release the mouse to move the items.

   There is no undo.

Removing a Folder from the Files tab of the Project Explorer

You may delete folders on the **Files** tab of the **Project Explorer**, however, you may not delete any of the predefined Silk Test Classic categories (nodes) such as Profile Script, Plan, Frame, and Data.

Note: There is no undo.

To remove a folder:

1. If your project is not already open, click **File** > **Open Project** or click **Open Project** > **Open Project** on the basic workflow bar. Select a project, then click **Open**.
2. In the **Project Explorer**, click the **Files** tab, right-click a folder and select **Remove Folder** to delete it from the **Project Explorer**. If you select a folder with child folders or a folder that contains items, Silk Test Classic displays a warning before deleting the folder.

Renaming a Folder on the Files Tab of the Project Explorer

You may rename any folder that you have added to a project. You may not rename any of the predefined Silk Test Classic folders (nodes) such as Profile, Script, Include/Frame, Plan, Results, or Data.

To rename a folder:

1. If your project is not already open, click **File** > **Open Project** or click **Open Project** > **Open Project** on the basic workflow bar. Select a project, then click **Open**.
2. In the **Project Explorer**, click the **Files** tab, then navigate to the folder you want to rename.
3. Right-click the folder and select **Rename Folder**.
4. On the **Rename Folder** dialog box, enter the new name of the folder then click **OK**.

When naming a folder, you may use alphanumeric characters, underscore character, character space, or hyphens. Folder names may be a maximum of 64 characters long. You may not use periods or parentheses in folder names. Within a node, folder names must be unique.

Sorting Resources within the Global Tab of the Project Explorer

On the **Global** tab of the **Project Explorer**, you can sort the resources within each category (node) by resource name, file name, or file date.

To sort resources:

1. If your project is not already open, click **File** > **Open Project** or click **Open Project** > **Open Project** on the basic workflow bar, select the project whose elements you want to sort, and then click **Open**.
2. On the **Project Explorer**, click the **Global** tab, right-click the node associated with the type of element you want to sort, and then click **Sort by FileName** or **Sort by FileDate**.

   The default is sort by element name.
3. Click **Ascending** or **Descending** to indicate how you want to organize the sort.

   For example, to sort the elements of a script file by file date in reverse chronological order, right-click the **Script** node and select **Sort by FileDate**, then click **Descending**.

   When you release the mouse, the elements are sorted by the parameters you selected.

Moving Files Between Projects

We recommend that you use **Export Project** to move projects, but if you want to move only a few files rather than an entire project, you can open the project in Silk Test Classic and remove the files that you want to move from the project. Move the files to their new location in Windows Explorer, and then add the files back to the currently open project.

You can also move your project by opening the `projectname.vtp` and `projectname.ini` files in a text editor outside of Silk Test Classic and updating references to the location of source files. However, we recommend that you have strong knowledge of your files and how the partner and projectname .ini files work before attempting this. We advise you to use great caution if you decide to edit the projectname .vtp and projectname .ini files.

Removing Files from a Project

You cannot remove the `projectname.ini` file.

To remove a file from a project:

1. Click **File** > **Open Project** or click **Open Project** > **Open Project** on the basic workflow bar.
2. Click the plus sign [+] to expand the node associated with the type of file you want to remove, and then choose one of the following:

   - Right-click the file you want to remove, and then click **Remove File**.
   - Select the file in the **Project Explorer** and press the **Delete** key.
   - Select the file you want to remove on the **Files** tab, and then click **Project** > **Remove File**.

The file is removed from the project and references to the file are deleted from the `projectname.vtp`. The file itself is not deleted; it is just removed from the project.

Turning the Project Explorer View On and Off

The **Project Explorer** view is the default. If you do not want to view the **Project Explorer**, uncheck **Project** > **View Explorer**. You can continue to work with your files within the project, you just will not see the **Project Explorer**.

To turn **Project Explorer** view on, check **Project** > **View Explorer**.

If you do not want to use projects in Silk Test Classic, close the open project, if any, by clicking **File** > **Close Project**, and then use Silk Test Classic as you would have in the past.

Viewing Resources Within a Project

1. Click **File** > **Open Project** or click **Open Project** > **Open Project** on the basic workflow bar and select the project that you want to open.
2. Click one of the following:
   - The **Files** tab to view all the files associated with the open project.
   - The **Global** tab to view global objects defined in the files associated with the project.
3. To close all open files within a project, click **Window** > **Close All**.

Packaging a Silk Test Classic Project

You can package your Silk Test Classic project into a single compressed file that you can relocate to a different computer. When you unpack your project you will have a fully functional set of test files. This is useful if you need to relocate a project, email a project to a co-worker, or send a project to technical support.

Source files included in the packaged project

When you package a project, Silk Test Classic includes all of the source files, meaning the related files used by a project, such as:

Description	Extension
plan files	.pln
script files	.t
include files	.inc
suite files	.s
results files (optional)	.res and .rex
data files	-

Silk Test Classic takes these files and bundles them up into a new file with an .stp extension. The .stp file includes the configuration files necessary for Silk Test Classic to set up the proper testing environment such as `project.ini`, `testplan.ini`, optionset .opt files, and any .ini files found in the `…\Silk Test Classic projects\<Project name>\extend` directory.

You have the option of including .res and .rex files when you package a Silk Test Classic project because these files are sometimes quite large and not necessary to run the project.

Relative paths in comparison to absolute paths

When you work with Silk Test Classic projects, the files that make up the project are identified by pathnames that are either absolute or relative. A relative pathname begins at a current folder or some number of folders up the hierarchy and specifies the file's location from there. An absolute pathname begins at the root of the file system (the topmost folder) and fully specifies the file's location from there. For example:

Absolute path `C:\Users\<Current user>\Documents\Silk Test Classic Projects\<Project name>\options.ini`

Relative path `..\tesla\Silk Test\options\options.ini` or `SUSDir\options.inc`

When you package a project, Silk Test Classic checks to make sure that the paths used within the project are properly maintained. If you try to compress a project containing ambiguous paths, Silk Test Classic displays a warning message. Silk Test Classic tracks the paths in a project in a log file.

Including all files needed to run tests

Files associated with a project, but not necessary to run tests, for example bitmap or document files, which you have manually added to the project are included when Silk Test Classic packages a project.

If Silk Test Classic finds any include:, script:, or use: statements in the project files that refer to files with absolute paths, `c:\program files\Silk\Silk Test\`, Silk Test Classic verifies if you have checked the **Use links for absolute files?** check box on the **Export Project** or on the **Email Project** dialog boxes.

- If you check the **Use links for absolute files?** check box, Silk Test Classic treats any file referenced by an absolute path in an include, script, or use statement as a placeholder and does not include those files in the package. For example, if there are use files within the **Runtime Options** dialog box referred to as "q:\qaplans\SilkTest\frame.inc" or "c", these files are not included in the package. The assumption is that these files will also be available from wherever you unpack the project.
- If you uncheck the **Use links for absolute files?** check box, Silk Test Classic includes the files referenced by absolute paths in the packaged project. For example, if the original file is stored on `c:\temp\myfile.t`, when unpacked at the new location, the file is placed on `c:\temp\myfile.t`.

The following table compares the results of packaging projects based on whether there are any absolute file references in your source files, and how you respond to the **Use links for absolute files?** check box on the **Export Project** or on the **Email Project** dialog boxes.

Any absolute references in source files?	Use links for absolute files?	Results
No	Checked or unchecked	Package unpacks to any location.
Yes	Checked	Files referenced by absolute paths are not included in the packaged project.
Yes	Unchecked	Files referenced by absolute paths are put into a ZIP file within the packaged project.

Note:

- If there are any source files located on a different drive than the .vtp project file, and if there are files referenced by absolute paths in the source files, Silk Test Classic treats the source files as referenced by absolute paths. The assumption is that the absolute paths will be available from the new location. Silk Test Classic therefore puts the files into a zip file within the packaged project for you to unpack after you unpack the project.
- Files not included in the package - The assumption is that since these files are referenced by absolute paths, these same files and paths will be available when the files are unpacked. On unpacking, Silk Test Classic warns you about these files and lists them in a log file (manifest.XXX).
- ip files – Because you elected not to use links for files referenced by absolute paths, these files are put into a zip file within the packaged project. The zip file is named with the root of the absolute path. For example, if the files are located on c:/, the zip file is named `c.zip`.

Tips for successful packaging and unpacking

For best results when packaging and unpacking Silk Test Classic projects:

- Put your .vtp project file and source files on the same drive.

- Use relative paths to reference the following:
- • include statements
  • options sets
  • use paths set within the **Runtime Options** dialog box
  • use statements in 4Test scripts
  • script statements
- Uncheck the default **Use links for absolute files?** check box if your source files are on a different drive as the .vtp project file and if there are files referenced by absolute paths in your source files.

Packaging with Silk Test Classic Runtime and the Agent

If you are running Silk Test Classic Runtime, you may not package or email a project.

If you are running the Agent, you may package or email a project.

Emailing a Packaged Project

Emailing a project automatically packages a Silk Test Classic project and then emails it to an email address. In order for this to work, you must have an email client installed on the computer that is running Silk Test Classic.

You cannot email a project if you are running Silk Test Classic Runtime.

One of the options you can select before emailing is to compile your project. If a compile error occurs, Silk Test Classic displays a warning message, and you can opt to continue or to cancel the email.

Silk Test Classic supports any MAPI-compliant e-mail clients such as Outlook Express.

The maximum size for the emailed project is determined by your e-mail client. Silk Test Classic does not place any limits on the size of the project.

To email your project:

1. If your project is not already open, click **File** > **Open Project** or click **Open Project** > **Open Project** on the basic workflow bar. Select a project, then click **Open**.
2. Click **File** > **Email Project**.

   You can only email a project if you have that project open.
3. On the **Email Project** dialog box, type the email address where you want to send the Silk Test Classic project.

   For example, enter `support@acme.com` to send a package to Acme Technical Support. You do not have to specify an email address here; your email program will prompt you for one before sending the email.
4. Select the options for the package you want to email.

   For an explanation of these options, see the description of the **Email Project** dialog box. The **Email Address** text box is required, though you can edit it later.
5. Click **OK**. If you opted to compile the project before packaging it, Silk Test Classic displays a warning message if any file failed to compile. Silk Test Classic opens a new email message and attaches the packaged project to a message. You can edit the recipient, add a subject line, and text, just as you can for any outgoing message.
6. Click **Send** to add the project to your outgoing queue. If your email client is already open, your message is sent automatically. If your email client was not open, the message is placed in your outgoing queue.

Note: If you have a crash during the email process, we recommend deleting the partially packaged project or draft email message, if any, and starting the process again.

Exporting a Project

Exporting a Silk Test Classic project lets you copy all the files associated with a project to a directory or a single compressed file in a directory.

You cannot export a project if you are running Silk Test Classic Runtime.

Silk Test Classic will not change the file creation dates when copying the project's files.

One of the options you can select before exporting is to compile your project. If a compile error occurs, Silk Test Classic displays a warning message, and you can opt to continue or to cancel the compile.

To export your project:

1. If your project is not already open, click **File** > **Open Project** or click **Open Project** > **Open Project** on the basic workflow bar. Select a project, then click **Open**.
2. Click **File** > **Export Project**.

   You can only export a project if you have the project open.
3. On the **Export Project** dialog box, enter the directory to which you want to export the project or click to locate the export folder.

   The default location is the parent directory of the project folder, which means the folder containing the project file, not the project's current location.
4. Check the **Export to single Silk Test Classic package** check box if you want to package the Silk Test Classic project into a single compressed file.
5. In the **Options** area, select the appropriate options for your project.

   For an explanation of these options, see the description of the **Export Project** dialog box.

   Note: Using references for absolute paths produces a smaller package that can be opened more quickly.
6. Click **OK**. Silk Test Classic determines all the files necessary for the project and copies them to the selected directory or compresses them into a package. Silk Test Classic displays a warning message if any of the files could not be successfully packaged and gives you the option of continuing.

If you have a crash during the export process, we recommend deleting the partially packaged project, if any, and starting the process over again.

Troubleshooting Projects

This section provides solutions to common problems that you might encounter when you are working with projects in Silk Test Classic.

Files Not Found When Opening Project

If, when opening your project, Silk Test Classic cannot find a file in the location referenced in the project file, which is a `.vtp` file, an error message displays noting the file that cannot be found.

Silk Test Classic may not be able to find files that have been moved or renamed outside of Silk Test Classic, for example in Windows Explorer, or files that are located on a shared network folder that is no longer accessible.

- If Silk Test Classic cannot find a file in your project, we suggest that you note the name of missing file, and click **OK**. Silk Test Classic will open the project and remove the file that it cannot find from the project list. You can then add the missing file to your project.

- If Silk Test Classic cannot open multiple files in your project, we suggest you click **Cancel** and determine why the files cannot be found. For example a directory might have been moved. Depending upon the problem, you can determine how to make the files accessible to the project. You may need to add the files from their new location.

Silk Test Classic Cannot Load My Project File

If Silk Test Classic cannot load your project file, the contents of your .vtp file might have changed or your .ini file might have been moved.

If you remove or incorrectly edit the `ProjectIni=` line in the `ProjectProfile` section of your `<projectname>.vtp` file, or if you have moved your `<projectname>.ini` file and the `ProjectIni=` line no longer points to the correct location of the `.ini` file, Silk Test Classic is not able to load your project.

To avoid this, make sure that the `ProjectProfile` section exists in your `.vtp` file and that the section refers to the correct name and location of your `.ini` file. Additionally, the `<projectname>.ini` file and the `<projectname>.vtp` file refer to each other, so ensure that these references are correct in both files. Perform these changes in a text editor outside of Silk Test Classic.

Example

The following code sample shows a sample `ProjectProfile` section in a `<projectname>.vtp` file:

```
[ProjectProfile]
ProjectIni=C:\Program Files\<Silk Test install directory>
\SilkTest\Projects\<projectname>.ini
```

Silk Test Classic Cannot Save Files to My Project

You cannot add or remove files from a read-only project. If you attempt to make any changes to a read-only project, a message box displays indicating that your changes will not be saved to the project.

For example, `Unable to save changes to the current project. The project file has read-only attributes.`

When you click **OK** on the error message box, Silk Test Classic adds or removes the file from the project temporarily for that session, but when you close the project, the message box displays again. When you re-open the project, you will see your files have not been added or removed.

Additionally, if you are using Microsoft Windows 7 or later, you might need to run Silk Test Classic as an administrator. To run Silk Test Classic as an administrator, right-click the Silk Test Classic icon in the **Start Menu** and click **Run as administrator**.

Silk Test Classic Does Not Run

The following table describes what you can do if Silk Test Classic does not start.

If Silk Test Classic does not run because it is looking for the following:	You can do the following:
Project files that are moved or corrupted.	Open the `SilkTestClassic.ini` file in a text editor and remove the `CurrentProject=` line from the `ProjectState` section. Silk Test Classic should then start, however your project will not open. You can examine your `<projectname>.ini` and

If Silk Test Classic does not run because it is looking for the following:	You can do the following:
	<projectname>.vtp files to determine and correct the problem. The following code example shows the ProjectState section in a sample partner.ini file: [ProjectState] CurrentProject=C:\Program Files \<SilkTest install directory> \SilkTest\Examples\ProjectName.vtp
A testplan.ini file that is corrupted.	Delete or rename the corrupted testplan.ini file, and then restart Silk Test Classic.

My Files No Longer Display In the Recent Files List

After you open or create a project, files that you had recently opened outside of the project do no longer display in the **Recent Files** list.

Cannot Find Items In Classic 4Test

If you are working with Classic 4Test, objects display in the correct nodes on the **Global** tab, however when you double-click an object, the file opens and the cursor displays at the top of the file, instead of in the line in which the object is defined.

Editing the Project Files

You require good knowledge of your files and how the partner and <projectname>.ini files work before attempting to edit these files. Be cautious when editing the <projectname>.vtp and <projectname>.ini files.

To edit the <projectname>.vtp and <projectname>.ini files:

1. Update the references to the source location of your files. If the location of your projectname.vtp and projectname.ini files has changed, make sure you update that as well. Each file refers to the other.

The ProjectProfile section in the projectname.vtp file is required. Silk Test Classic will not be able to load your project if this section does not exist.

1. Ensure that your project is closed and that all the files referenced by the project exist.
2. Open the <projectname>.vtp and <projectname>.ini files in a text editor outside of Silk Test Classic.

   Note: Do not edit the projectname.vtp and projectname.ini files in the 4Test Editor.
3. Update the references to the source location of your files.
4. The <projectname>.vtp and <projectname>.ini files refer to each other. If the relative location of these files has changed, update the location in the files.

The ProjectProfile section in the <projectname>.vtp file is required. Silk Test Classic is not able to load your project if this section does not exist.

Enabling Extensions for Applications Under Test

This functionality is supported only if you are using the Classic Agent.

This section describes how you can use extensions to extend the capabilities of a program or the data that is available to the program.

An extension is a file that serves to extend the capabilities of, or the data available to, a basic program. Silk Test Classic provides extensions for testing applications that use non-standard controls in specific development and browser environments.

For information about new features, supported platforms and versions, known issues, and work-arounds, refer to the *Release Notes*.

Related Files

If you are using a project, the extension configuration information is stored in the `partner.ini` file. If you are not using a project, the extension configuration information is stored in the `extend.ini` file.

When you enable extensions, Silk Test Classic adds an include file based on the technology or browser type that you enable to the **Use files** location in the **Runtime Options** dialog box. Extensions that use technologies on the Classic Agent are located in the `<Silk Test Classic project directory>\extend\` directory.

Extensions that Silk Test Classic can Automatically Configure

This functionality is supported only if you are using the Classic Agent.

Using the **Basic Workflow**, Silk Test Classic can automatically configure extensions for many development environments, including:

- Browser applications and applets running in one of the supported browsers.
- .NET standalone Windows Forms applications.
- Standalone Java and Java AWT applications.
- Java Web Start applications and InstallAnywhere applications and applets.
- Java SWT applications.
- Visual Basic applications.
- Client/Server applications.

For information about new features, supported platforms and versions, known issues, and work-arounds, refer to the *Release Notes*.

You cannot enable extensions for Silk Test Classic (`partner.exe`), Classic Agent (`agent.exe`), or Open Agent (`openAgent.exe`).

You can also click **Tools** > **Enable Extensions** to have Silk Test Classic automatically set your extension.

If the **Basic workflow** does not support your configuration, you can enable the extension manually.

If you use the Classic Agent, the **Basic Workflow** does not automatically configure browser applications containing ActiveX objects. To configure a browser application with ActiveX objects, check the **ActiveX** check box in the row for the extension that you are enabling in the **Extensions** dialog box. Or use the Open Agent.

Extensions that Must be Set Manually

This functionality is supported only if you are using the Classic Agent.

Using the **Basic Workflow**, Silk Test Classic can automatically enable extensions for many different development environments. If the **Basic Workflow** does not support your configuration or you prefer to enable extensions manually, enable the extension on your host machine and enable the extension on your target machine, regardless of whether the application you plan to test will run locally or on remote machines. Enable extensions manually if you:

- Want to change your currently enabled extension.
- Want to enable additional options for the extension you are using, such as Accessibility, Active X, or Java.
- Are testing embedded browser applications using the Classic Agent, for example, if DOM controls are embedded within a Windows Forms application.
- Are testing an application that does not have a standard name.

If you are testing Web applications using the Classic Agent, Silk Test Classic enables the extension associated with the default browser you specified on the **Select Default Browser** dialog box during the Silk Test Classic installation. If you want to use the extension you specified during the Silk Test Classic installation, you do not need to complete this procedure unless you need additional options, such as Accessibility, Java, or ActiveX.

If you are not testing Java but do have Java installed, we recommend that you disable the classpath before using Silk Test Classic.

Silk Test Classic automatically enables Java support in the browser if your web page contains an applet. The **Enable Applet Support** check box on the **Extension Settings** dialog for browser is automatically selected when the **Enable Extensions** workflow detects an applet. You can uncheck the check box to prevent Silk Test Classic from loading the extension. If no applet is detected, the check box is not available.

Extensions on Host and Target Machines

This functionality is supported only if you are using the Classic Agent.

You must define which extensions Silk Test Classic should load for each application under test, regardless of whether the application will run locally or on remote machines. You do this by enabling extensions on your host machine and on each target machine before you record or run tests.

Extensions on the host machine

On the host machine, we recommend that you enable only those extensions required for testing the current application. Extensions for all other applications should be disabled on the host to conserve memory and other system resources. By default, the installation program:

- Enables the extension for your default Web browser environment on the host machine.
- Disables extensions on the host machine for all other browser environments.
- Disables extensions for all other development environments.

When you enable an extension on the host machine, Silk Test Classic does the following:

- Adds the include file of the extension to the **Use Files** text box in the **Runtime Options** dialog box, so that the classes of the extension are available to you.
- Makes sure that the classes defined in the extension display in the **Library Browser**. Silk Test Classic does this by adding the name of the extension's help file, which is `browser.ht`, to the **Help Files For**

Library Browser text box in **General Options** dialog box and recompiling the help file used by the **Library Browser**.

- Merges the property sets defined for the extension with the default property sets. The web-based property sets are in the `browser.ps` file in the `Extend` directory. The file defines the following property sets: Color, Font, Values, and Location.

Extensions on the target machine

The **Extension Enabler** dialog box is the utility that allows you to enable or disable extensions on your target machines. All information that you enter in the **Extension Enabler** is stored in the `extend.ini` file and allows the Agent to recognize the non-standard controls you want to test on target machines.

Enabling Extensions Automatically Using the Basic Workflow

An extension is a file that serves to extend the capabilities of, or data available to, a more basic program. Silk Test Classic provides extensions for testing applications that use non-standard controls in specific development and browser environments.

If you are testing a generic project that uses the Classic Agent, perform the following procedure to enable extensions:

1. Start the application or applet for which you want to enable extensions.
2. Start Silk Test Classic and make sure the basic workflow bar is visible. If it is not, click **Workflows** > **Basic** to enable it.

   If you do not see **Enable Extensions** on the workflow bar, ensure that the default agent is set to the Classic Agent.
3. If you are using Silk Test Classic projects, click **Project** and open your project or create a new project.
4. Click **Enable Extensions**.

   You cannot enable extensions for Silk Test Classic (`partner.exe`), the Classic Agent (`agent.exe`), or the Open Agent (`openAgent.exe`).
5. Select your test application from the list on the **Enable Extensions** dialog box, and then click **Select**.
6. If your test application does not display in the list, click **Refresh**. Or, you may need to add your application to this list in order to enable its extension.
7. Click **OK** on the **Extension Settings** dialog box, and then close and restart your application.
8. If you are testing an applet, the **Enable Applet Support** check box is checked by default.
9. When the **Test Extension Settings** dialog box opens, restart your application in the same way in which you opened it; for example, if you started your application by double-clicking the .exe, then restart it by double-clicking the .exe.
10. Make sure the application has finished loading, and then click **Test**. When the test is finished, a dialog box displays indicating that the extension has been successfully enabled and tested. You are now ready to begin testing your application or applet. If the test fails, review the troubleshooting topics.

When you enable extensions, Silk Test Classic adds an include file based on the technology or browser type that you enable to the **Use files location** in the **Runtime Options** dialog box.

Enabling Extensions on a Host Machine Manually

This functionality is supported only if you are using the Classic Agent.

Using the **Basic workflow**, Silk Test Classic can automatically enable extensions for many different development environments. If you would rather enable the extension manually, or the basic workflow does not support your configuration, follow the steps described in this topic.

A host machine is the system that runs the Silk Test Classic software process, in which you develop, edit, compile, run, and debug 4Test scripts and test plans.

There is overhead to having more than one browser extension enabled, so you should enable only one browser extension unless you are actually testing more than one browser in an automated session.

1. Start Silk Test Classic and click **Options** > **Extensions**.
2. If you are testing a client/server project, rich internet application project, or a generic project that uses the Classic Agent, perform the following steps:
   a) On the **Extensions** dialog box, click the extension you want to enable. You may need to add your application to this list in order to enable its extension.
   b) Enable other extensions, such as Java, ActiveX, Accessibility, and .NET, as appropriate.
   c) Disable other extensions that you will not be using by selecting **Disabled** in the **Primary Extension** field. To disable a Visual Basic extension, uncheck the **ActiveX** check box for the Visual Basic application.
   d) Click **OK**.

Manually Enabling Extensions on a Target Machine

This functionality is supported only if you are using the Classic Agent.

Using the basic workflow, Silk Test Classic can automatically enable extensions for many different development environments. If you would rather enable the extension manually, or the basic workflow does not support your configuration, follow the steps described in this topic.

A target machine is a system (or systems) that runs the 4Test Agent, which is the software process that translates the commands in your scripts into GUI-specific commands, in essence, driving and monitoring your applications under test. One Agent process can run locally on the host machine, but in a networked environment, any number of Agents can run on remote machines.

If you are running local tests, that is, your target and host are the same machine, complete this procedure and enable extensions on a host machine manually.

1. Make sure that your browser is closed.
2. From the Silk Test Classic program group, choose **Extension Enabler**. To invoke the **Extension Enabler** on a remote non-Windows target machine, run `extinst.exe`, located in the directory on the target machine in which you installed the Classic Agent.
3. Enable other extensions, such as Java, ActiveX, Accessibility, and .NET, as appropriate. To get information about the files used by an extension, select an extension and click **Details**. You may need to add your application to this list in order to enable its extension.
4. Click **OK** to close the **Extension Enabler** dialog box.

   If you enable support for ActiveX in this dialog box, make sure that it is enabled in the **Extensions** dialog box as well.
5. Restart your browser, if you enabled extensions for web testing.

   Once you have set your extension(s) on your target and host machines, verify the extension settings to check your work. Be sure to consider how you want to work with borderless tables. If you are testing non-Web applications, you must disable browser extensions on your host machine. This is because the recovery system works differently when testing Web applications than when testing non-Web applications. For more information about the recovery system for testing Web applications, see Web applications and the recovery system. When you select one or both of the Internet Explorer extensions on the host machine's **Extension** dialog box, Silk Test Classic automatically picks the correct version of the host machine's Internet Explorer application in the **Runtime Options** dialog box. If the target

machine's version of Internet Explorer is not the same as the host machine's, you must remember to change the target machine's version.

Enabling Extensions for Embedded Browser Applications that Use the Classic Agent

This functionality is supported only if you are using the Classic Agent.

To test an embedded browser application, enable the Web browser as the primary extension for the application in both the **Extension Enabler** and in the Silk Test Classic **Extensions** dialog boxes. For instance, if you are testing an application with DOM controls that are embedded within a .NET application, follow the following instructions to enable extensions.

1. Click **Start** > **Programs** > **Silk** > **Silk Test** > **Tools** > **Extension Enabler**.
2. Browse to the location of the application executable.
3. Select the executable file and then click **Open**.
4. Click **OK**.
5. From the **Primary Extension** list box, select the DOM extension for the application that you added.
6. Enable other extensions, such as Java, ActiveX, Accessibility, and .NET, as appropriate.

   For example, to test a .NET application with embedded Web controls, select a browser in the **Primary Extension** list box and check the .NET check box for the application within the grid.
7. Click **OK**.
8. Start Silk Test Classic and then choose **Options** > **Extensions**. The **Extensions** dialog box opens.
9. Click **New**.
10. Browse to the location of the application executable.
11. Select the executable file and then click **Open**.
12. Click **OK**.
13. From the **Primary Extension** list box, select the DOM extension for the application that you added.
14. Enable other extensions, such as Java, ActiveX, Accessibility, and .NET, as appropriate.

    For example, to test a .NET application with embedded Web controls, select a browser in the **Primary Extension** list box and check the .NET check box for the application within the grid.
15. Click **OK**.
16. Restart Silk Test Classic.

Note: The IE DOM extension may not detect changes to a web page that occur when JavaScript replaces a set of elements with another set of elements without changing the total number of elements. To force the DOM extension to detect changes in this situation, call the `FlushCache`() method on the top-level browserchild for the embedded browser. This problem might occur more often for embedded browsers than for browser pages, because Silk Test Classic is not notified of as many browser events for embedded browsers. Also call `FlushCache`() if you get a `Coordinate out of bounds` exception when calling a method, for example `Click`(), on an object that previously had been scrolled into view. The *BrowserPage* window identifier is not valid when using embedded browsers because the default browser type is `'(none)' (NULL)`.

Enabling Extensions for HTML Applications (HTAs)

This functionality is supported only if you are using the Classic Agent.

You must enable extensions on the host and target machines manually in order to use HTML applications (HTAs).

Before you begin, create a project that uses the Classic Agent.

1. Click **Options** > **Extensions** to open the **Extensions** dialog box.
2. Click **New** to open the **Extension Application** dialog box.
3. Click to navigate to the location of the `.hta` file that you want to enable. If the file name contains spaces, be sure to enclose the name in quotation marks.
4. Select the `.hta` file and then click **Open**.
5. Click **OK**.
6. In the **Primary Extension** column next to the `.hta` application that you just enabled, select `Internet Explorer`.
7. Click **OK**.
8. Click **Start** > **Programs** > **Silk** > **Silk Test** > **Tools** > **Extension Enabler**. (Or use the command line to launch `"C:\Progam Files\Silk\SilkTest\Tools\extinst.exe"`.)
9. On the **Extension Enabler** dialog box, click **New** to open the **Extension Application** dialog box.
10. Click to navigate to the location of the `.hta` file that you want to enable. If the file name contains spaces, be sure to enclose the name in quotation marks.
11. Select the `.hta` file and then click **Open**.
12. Click **OK**.
13. In the **Primary Extension** column next to the `.hta` application that you just enabled, select `Internet Explorer`.
14. Click **OK**.

Adding a Test Application to the Extension Dialog Boxes

This functionality is available only for projects or scripts that use the Classic Agent.

You must manually add the following applications to the **Extensions** dialog box and the **Extension Enabler** dialog box:

- Applications that are embedded in Web pages and use the Classic Agent.
- All test applications that do not have standard names and use the Classic Agent.
- When you add a test application to the **Extensions** dialog box on the host machine, you should immediately add it to the **Extension Enabler** dialog box on each target machine on which you intend to test the application.

You may also add new applications by duplicating existing applications and then changing the application name.

To add a test application to the **Extension** dialog boxes:

1. Click **Options** > **Extensions** to open the **Extensions** dialog box, or open the **Extension Enabler** dialog box from the Silk Test program group.
2. If you are testing a client/server project, Rich Internet Application project, or a generic project that uses the Classic Agent, perform the following steps:
   a) Click **New** to open the **Extension Application** dialog box.
   b) Click **...** to browse to the application's executable or DLL file.

      Separate multiple application names with commas. If the executable name contains spaces, be sure to enclose the name in quotation marks.
   c) Select the executable file and then click **Open**.
   d) Click **OK**.

3. Click **OK** to close the dialog box.

Verifying Extension Settings

This functionality is available only for projects or scripts that use the Classic Agent.

If the extension settings for the host and target machines do not match, neither extension will load properly.

- To see the target machine setting, choose **Options** > **Extensions**. Verify that the Primary Extension is enabled and other extensions are enabled, if appropriate. If you enabled a browser extension, you can also verify the extension settings on the target machine by starting the browser and Silk Test Classic, and then right-clicking the task bar Agent icon and selecting **Extensions** > **Detail**.
- To verify that the setting on the host machine is correct, choose **Options** > **Runtime**. Make sure that the default browser in the **Default Browser** field on the **Runtime Options** dialog box is correct.

Why Applications do not have Standard Names

This functionality is supported only if you are using the Classic Agent.

In the following situations applications might not have standard names, in which case you must add them to the **Extension Enabler** dialog box and the **Extensions** dialog box:

- Visual Basic applications can have any name, and therefore the Silk Test Classic installation program cannot add them to the dialog box automatically.
- You are running an application developed in Java as a stand-alone application, outside of its normal runtime environment.
- You have explicitly changed the name of a Java application.

Duplicating the Settings of a Test Application in Another Test Application

This functionality is supported only if you are using the Classic Agent.

You can add new applications to the **Extension Enabler** dialog box or the **Extensions** dialog box by duplicating existing applications and renaming the new application. All the settings of the original application, that is, primary extension, other extensions, or options set on the **Extensions** dialog box, are copied.

You can only duplicate applications that you entered manually and that use the Classic Agent.

To copy a test application's settings into another application:

1. Click **Options** > **Extensions** to open the **Extensions** dialog box, or open the **Extension Enabler** dialog box from the Silk Test Classic program group.
2. Select the application that you want to copy.
3. Click **Duplicate**. The **Extension Application** dialog box opens.
4. Type the name of the new application you want to copy.

   Separate multiple application names with commas.
5. Click **OK** to close the **Extension Application** dialog box. The new applications display in the dialog box you opened.
6. Click **OK** to close the dialog box.

Deleting an Application from the Extension Enabler or Extensions Dialog Box

This functionality is supported only if you are using the Classic Agent.

After completing your testing of an application or if you make a mistake, you might want to delete the application from the **Extension Enabler** dialog box or the **Extensions** dialog box. You can delete only applications that you have entered manually. Visual Basic applications fall into this category.

To remove an application from the **Extension Enabler** or **Extensions** dialog box:

1. Click **Options** > **Extensions** to open the **Extensions** dialog box, or open the **Extension Enabler** dialog box from the Silk Test Classic program group.
2. Select the application that you want to delete from the dialog box.
3. Click **Remove**. The application name is removed from the dialog box.
4. Click **OK**.

Disabling Browser Extensions

This functionality is supported only if you are using the Classic Agent.

1. In Silk Test Classic, choose **Options** > **Extensions**.
2. From the **Primary Extension** list, select **Disabled** for the extension you want to disable.
3. In the **Other extensions** field, uncheck any checked check boxes.
4. Click **OK**.

If you are testing non-Web applications, you must disable browser extensions on your host machine. This is because the recovery system works differently when testing Web applications than when testing non-Web applications.

Comparison of the Extensions Dialog Box and the Extension Enabler Dialog Box

This functionality is supported only if you are using the Classic Agent.

The **Extensions** dialog box and the **Extension Enabler** dialog box look similar; they are both based on a grid and have identical column headings and have some of the same buttons. However, they configure different aspects of the product:

	Extensions Dialog Box	**Extension Enabler Dialog Box**
Enables AUTs and extensions	On host machine	On target machines
Provides information for	Silk Test Classic	Agent
Available from	**Options** menu	Silk Test Classic program group
Information stored in	`partner.ini`	`extend.ini`
When to enable/disable AUTs and extensions	Enable the AUTs and extensions you want to test now; disable others.	Enable all AUTs and extensions you ever intend to test. No harm in leaving them enabled, even if you are not testing them now.

	Extensions Dialog Box	Extension Enabler Dialog Box
What you specify on each: • Primary environment • Java or ActiveX, if required • Accessibility	• Yes, according to the type • Enable and set options • Enable and set options	• Yes, according to the type • Enable only • Enable only
What installation does: • Default browser (If any) • Other browsers (if any) • Java runtime environment • Oracle Forms runtime environment • Visual Basic 5 & 6	• Displayed and enabled • Displayed but disabled • Displayed but disabled • Displayed but disabled • Not displayed or enabled	• Displayed and enabled • Displayed and enabled • Displayed and enabled • Displayed but disabled • Not displayed or enabled

Configuring the Browser

This functionality is supported only if you are using the Classic Agent.

In order for Silk Test Classic to work properly, make sure that your browser is configured correctly.

If your tests use the recovery system of Silk Test Classic, that is, your tests are based on DefaultBaseState or on an application state that is ultimately based on DefaultBaseState, Silk Test Classic makes sure that your browser is configured correctly.

If your tests do not use the recovery system, you must manually configure your browser to make sure that your browser displays the following items:

- The standard toolbar buttons, for example **Home**, **Back**, and **Stop**, with the button text showing. If you customize your toolbars, then you must display at least the **Stop** button.
- The text box where you specify URLs. **Address** in Internet Explorer.
- Links as underlined text.
- The browser window's menu bar in your Web application. It is possible through some development tools to hide the browser window's menu bar in a Web application. Silk Test Classic will not work properly unless the menu bar is displayed. The recovery system cannot restore the menu bar, so you must make sure the menu bar is displayed.
- The status bar at the bottom of the window shows the full URL when your mouse pointer is over a link.

We recommend that you configure your browser to update cached pages on a frequent basis.

Internet Explorer

1. Click **Tools** > **Internet Options**, then click the **General** tab.
2. In the **Temporary Internet Files** area, click **Settings**.
3. On the **Settings** dialog box, select **Every visit to the page** for the **Check for newer versions of stored pages** setting.

Mozilla Firefox

1. Choose **Edit** > **Preferences** > **Advanced** > **Cache**.
2. Indicate when you want to compare files and update the cache. Select **Every time I view the page** at the **Compare the page in the cache to the page on the network** field.

AOL

Even though AOL's Proxy cache is updated every 24 hours, you can clear the AOL Browser Cache and force a page to reload. To do this, perform one of the following steps:

- Delete the files in the temporary internet files folder located in the Windows directory.
- Press the **CTRL** key on your keyboard and click the AOL browser reload icon (Windows PC only).

Friendly URLs

Some browsers allow you to display "friendly URLs," which are relative to the current page. To make sure you are not displaying these relative URLs, in your browser, display a page of a web site and move your mouse pointer over a link in the page.

- If the status bar displays the full URL (one that begins with the http:// protocol name and contains the site location and path), the settings are fine. For example: `http://www.mycompany.com/products.htm`
- If the status bar displays only part of the URL (for example, `products.htm`), turn off "friendly URLs." (In Internet Explorer, this setting is on the **Advanced** tab of the **Internet Options** dialog box.)

Setting Agent Options for Web Testing

This functionality is supported only if you are using the Classic Agent.

When you first install Silk Test Classic, all the options for Web testing are set appropriately. If, for some reason, for example if you were testing non-Web applications and changed them, you have problems with testing Web applications, perform the following steps:

1. Click **Options** > **Agent**. The **Agent Options** dialog box opens.
2. Ensure the following settings are correct.

Tab	Option	Specifies	Setting
Timing	`OPT_APPREADY_TIMEOUT`	The number of seconds that the agent waits for an application to become ready. Browser extensions support this option.	Site-specific; default is 180 seconds.
Timing	`OPT_APPREADY_RETRY`	The number of seconds that the agent waits between attempts to verify that the application is ready.	Site-specific; default is 0.1 seconds.
Other	`OPT_SCROLL_INTO_VIEW`	That the agent scrolls a control into view before recording events against it.	TRUE (checked); default is TRUE.
Other	`OPT_SHOW_OUT_OF_VIEW`	Enables Silk Test Classic to see objects not currently scrolled into view.	TRUE (checked); default is TRUE.
Verification	`OPT_VERIFY_APPREADY`	Whether to verify that an application is ready. Browser extensions support this option.	TRUE (checked); default is TRUE.

3. Click **OK**. The **Agent Options** dialog box closes.

Specifying a Browser for Silk Test Classic to Use in Testing a Web Application

This functionality is supported only if you are using the Classic Agent.

You can specify a browser for Silk Test Classic to use when testing a Web application at runtime or you can use the browser specified through the **Runtime Options** dialog box.

To completely automate your testing, consider specifying the browser at runtime. You can do this in one of the following ways:

- Use the `SetBrowserType` function in a script. This function takes an argument of type `BROWSERTYPE`.

- Pass an argument of type BROWSERTYPE to a test case as the first argument.

For an example of passing browser specifiers to a test case, see the second example in BROWSERTYPE. It shows you how to automate the process of running a test case against multiple browsers.

Specifying a browser through the Runtime Options dialog box

When you run a test and do not explicitly specify a browser, Silk Test Classic uses the browser specified in **Runtime Options** dialog box. To change the browser type, you can:

1. Run a series of tests with a specific browser.
2. Specify a different browser in the **Runtime Options** dialog box.
3. Run the tests again with the new browser.

Most tests will run unchanged between browsers.

Specifying your Default Browser

Whenever you record and run test cases, you must specify the default browser that Silk Test Classic should use. If you did not choose a default browser during the installation of Silk Test Classic or if want to change the default browser, perform the following steps:

1. Click **Options** > **Runtime**. The **Runtime Options** dialog box opens.
2. Select the browser that you want to use from the **Default Browser** list box.
   The list box displays the browsers whose extensions you have enabled.
3. Click **OK**.

Understanding the Recovery System for the Classic Agent

The built-in recovery system is one of the most powerful features of Silk Test Classic because it allows you to run tests unattended. When your application fails, the recovery system restores the application to a stable state, known as the BaseState, so that the rest of your tests can continue to run unattended.

The recovery system can restore your application to its BaseState at any point during test case execution:

- Before the first line of your test case begins running, the recovery system restores the application to the BaseState even if an unexpected event corrupted the application between test cases.
- During a test case, if an application error occurs, the recovery system terminates the execution of the test case, writes a message in the error log, and restores the application to the BaseState before running the next test case.
- After the test case completes, if the test case was not able to clean up after itself, for example it could not close a dialog box it opened, the recovery system restores the application to the BaseState.
- The recovery system cannot recover from an application crash that produces a modal dialog box, such as a **General Protection Fault (GPF)**.

Silk Test Classic uses the recovery system for all test cases that are based on DefaultBaseState or based on a chain of application states that ultimately are based on DefaultBaseState.

- If your test case is based on an application state of none or a chain of application states ultimately based on none, all functions within the recovery system are not called. For example, SetAppState and SetBaseState are not called, while DefaultTestCaseEnter, DefaultTestCaseExit, and error handling are called.

Such a test case will be defined in the script file as:

```
testcase Name () appstate none
```

Silk Test Classic records test cases based on DefaultBaseState as:

```
testcase Name ()
```

How the default recovery system is implemented

The default recovery system is implemented through several functions.

Function	Purpose
`DefaultBaseState`	Restores the default BaseState, then call the application's BaseState function, if defined.
`DefaultScriptEnter`	Executed when a script file is first accessed. Default action: none.
`DefaultScriptExit`	Executed when a script file is exited. Default action: Call the ExceptLog function if the script had errors.
`DefaultTestCaseEnter`	Executed when a test case is about to start. Default action: Set the application state.
`DefaultTestCaseExit`	Executed when a test case has ended. Default action: Call the ExceptLog function if the script had errors, then set the BaseState.

Function	Purpose
`DefaultTestPlanEnter`	Executed when a test plan is entered. Default action: none.
`DefaultTestPlanExit`	Executed when a test plan is exited. Default action: none.

You can write functions that override some of the default behavior of the recovery system.

Setting the Recovery System for the Classic Agent

The recovery system ensures that each test case begins and ends with the application in its intended state. Silk Test Classic refers to this intended application state as the BaseState. The recovery system allows you to run tests unattended. When your application fails, the recovery system restores the application to the BaseState, so that the rest of your tests can continue to run unattended.

If you are testing an application that uses both the Classic Agent and the Open Agent, set the Agent that will start the application as the default Agent and then set the recovery system. If you use the Open Agent to start the application, set the recovery system for the Open Agent.

1. Make sure the application that you are testing is running.
2. Click **Set Recovery System** on the **Basic Workflow** bar. If the workflow bar is not visible, click **Workflows** > **Basic** to enable it.
3. From the **Application** list, click the name of the application that you are testing.

   All open applications that are not minimized are listed. This list is dynamic and will update if you open a new application. If you are connected to the Open Agent, only those applications that have extensions enabled display in the list.

   Note: If you selected a non-web application as the application:

   - The **Command line** text box displays the path to the executable (`.exe`) for the application that you have selected.
   - The **Working directory** text box displays the path of the application you selected.

   If you selected a web application, the **Start testing on this page** text box displays the URL for the application you selected. If an application displays in the list, but the URL does not display in this text box, your extensions may not be enabled correctly. Click **Enable Extensions** in the **Basic Workflow** bar to automatically enable and test extension settings.
4. *Optional:* In the **Frame file name** text box, modify the frame file name and click **Browse** to specify the location in which you want to save this file.

   Frame files must have a .inc extension. By default, this field displays the default name and path of the frame file you are creating. The default is `frame.inc`. If `frame.inc` already exists, Silk Test Classic appends the next logical number to the new frame file name; for example, `frame1.inc`.
5. *Optional:* In the **Window name** text box, change the window name to use a short name to identify your application.
6. Click **OK**.
7. Click **OK** when the message indicating that the recovery system is configured displays.
8. A new 4Test include file, `frame.inc`, opens in the Silk Test Editor. Click the plus sign in the file to see the contents of the frame file.
9. Record a test case.

Base State

An application's base state is the known, stable state that you expect the application to be in before each test case begins execution, and the state the application can be returned to after each test case has ended execution. This state may be the state of an application when it is first started.

Base states are important because they ensure the integrity of your tests. By guaranteeing that each test case can start from a stable base state, you can be assured that an error in one test case does not cause subsequent test cases to fail.

Silk Test Classic automatically ensures that your application is at its base state during the following stages:

- Before a test case runs.
- During the execution of a test case.
- After a test case completes successfully.

When an error occurs, Silk Test Classic does the following:

- Stops execution of the test case.
- Transfers control to the recovery system, which restores the application to its base state and logs the error in a results file.
- Resumes script execution by running the next test case after the failed test case.

The recovery system makes sure that the test case was able to "clean up" after itself, so that the next test case runs under valid conditions.

DefaultBaseState Function

Silk Test Classic provides a `DefaultBaseState` for applications, which ensures the following conditions are met before recording and executing a test case:

- The application is running.
- The application is not minimized.
- The application is the active application.
- No windows other than the application's main window are open. If the UI of the application is localized, you need to replace the strings, which are used to close a window, with the localized strings. The preferred way to replace these buttons is with the *IsCloseWindowButtons* variable in the object's declaration. You can also replace the strings in the **Close** tab of the **Agent Options** dialog box.

For Web applications that use the Open Agent, the `DefaultBaseState` also ensures the following for browsers, in addition to the general conditions listed above:

- The browser is running.
- Only one browser tab is open, if the browser supports tabs and the frame file does not specify otherwise.
- The active tab is navigated to the URL that is specified in the frame file.

For web applications that use the Classic Agent, the `DefaultBaseState` also ensures the following for browsers, in addition to the general conditions listed above:

- The browser is ready.
- Constants are set.
- The browser has toolbars, location and status bar are displayed.
- Only one tab is opened, if the browser supports tabs.

DefaultBaseState Types

Silk Test Classic includes two slightly different base state types depending on whether you use the Open Agent and dynamic object recognition or traditional hierarchical object recognition. When you use dynamic object recognition, Silk Test Classic creates a window object named `wDynamicMainWindow` in the base state. When you set the recovery system for a test that uses hierarchical object recognition, Silk Test Classic creates a window object called `wMainWindow` in the base state. Silk Test Classic uses the window object to determine which type of `DefaultBaseState` to execute.

Adding Tests that Use the Classic Agent to the DefaultBaseState

If you want the recovery system to perform additional steps after it restores the default base state, record a new method named `BaseState` and paste it into the declaration for your application's main window. Silk Test Classic provides the Record/Method menu command to record a `BaseState` method.

1. Open your application and the application's test frame file.
2. Place the insertion point on the declaration for the application's main window.
3. Click **Record** > **Method**. Silk Test Classic displays the **Record Method** dialog box, which allows you to record a method for a class or window declaration.
4. From the **Method Name** list box, select `BaseState`.
5. Click **Start Recording**. Silk Test Classic closes the **Record Method** dialog box and displays the **Record Status** window, which indicates that you can begin recording the `BaseState` method. The Status field flashes the word `Recording`.
6. When you have finished recording the `BaseState` method, click **Done** on the **Record Status** window. Silk Test Classic redisplays the **Record Method** dialog box. The **Method Code** field contains the 4Test code you recorded.
7. Click **OK** to close the **Record Method** dialog box and place the new `BaseState` method in the declaration for your main window.

DefaultBaseState and wMainWindow

Silk Test Classic executes the `DefaultBaseState` for hierarchical object recognition when the global constant `wMainWindow` is defined. `DefaultBaseState` works with the `wMainWindow` object in the following ways:

1. If the `wMainWindow` object does not exist, invoke it, either using the `Invoke` method defined for the `MainWin` class or a user-defined `Invoke` method built into the object. If `wMainWindow` is a `BrowserChild` object and the browser is not loaded, load the browser before loading the web page into it.
2. If the `wMainWindow` object is minimized, restore it. If `wMainWindow` is a `BrowserChild` object and the browser is minimized, restore it.
3. If there are child objects of the `wMainWindow` open, close them. If `wMainWindow` is a `BrowserChild` object, close any children of the browser.
4. If the `wMainWindow` object is not active, make it active.
5. If there is a `BaseState` method defined for the `wMainWindow` object, execute it.

In a scenario where a dialog box or a different Web page loads on request, the recovery system expects that web page to be loaded. However, it might not be if a Login page loads first. You can configure Silk Test Classic to handle login pages.

Flow of Control

This section describes the flow of control during the execution of each of your test cases.

The Non-Web Recovery Systems Flow of Control

Before you modify the recovery system, you need to understand the flow of control during the execution of each of your test cases. The recovery system executes the `DefaultTestcaseEnter` function. This function, in turn, calls the `SetAppState` function, which does the following:

1. Executes the test case.
2. Executes the `DefaultTestcaseExit` function, which calls the `SetBaseState` function, which calls the lowest level application state, which is either the `DefaultBaseState` or any user defined application state.

Note: If the test case uses `AppState` none, the `SetBaseState` function is not called.

`DefaultTestCaseEnter`() is considered part of the test case, but `DefaultTestCaseExit`() is not. Instead, `DefaultTestCaseExit`() is considered part of the function that runs the test case, which implicitly is `main`() if the test case is run standalone. Therefore an unhandled exception that occurs during `DefaultTestCaseEnter`() will abort the current test case, but the next test case will run. However, if the exception occurs during `DefaultTestCaseExit`(), then it is occurring in the function that is calling the test case, and the function itself will abort. Since an application state may be called from both `TestCaseEnter`() and `TestCaseExit`(), an unhandled exception within the application state may cause different behavior depending on whether the exception occurs upon entering or exiting the test case.

Web Applications and the Recovery System

This functionality is supported only if you are using the Classic Agent.

When the recovery system needs to restore the base state of a Web application that uses the Classic Agent, it does the following:

1. Invokes the default browser if it is not running.
2. Restores the browser if it is minimized.
3. Closes any open additional browser instances or message boxes.
4. Makes sure the browser is active and is not loading a page.
5. Sets up the browser as required by Silk Test Classic.

   The recovery system performs the next four steps only if the `wMainWindow` constant is set and points to the home page in your application.
6. If `bDefaultFont` is defined and set to TRUE for the home page, sets the fonts.
7. If `BrowserSize` is defined and set for the home page, sets the size of the browser window.
8. If `sLocation` is defined and set for the home page, loads the page specified by `sLocation`.
9. If `wMainWindow` defines a `BaseState` method, executes it.
10. For additional information, see *DefaultBaseState and the wMainWindow Object*.

To use the recovery system, you must have specified your default browser in the **Runtime Options** dialog box. If the default browser is not set, the recovery system is disabled. There is one exception to this rule: You can pass a browser specifier as the first argument to a test case. This sets the default browser at runtime. For more information, see *BROWSERTYPE Data Type*.

The constant `wMainWindow` must be defined and set to the identifier of the home page in the Web application for the recovery system to restore the browser to your application's main page. This window

must be of class `BrowserChild`. When you record a test frame, the constant is automatically defined and set appropriately. If you want, you can also define a `BaseState` method for the window to execute additional code for the base state, for example if the home page has a form, you might want to reset the form in the `BaseState` method, so that it will be empty at your base state.

On Internet Explorer 7.x and 8.x, when recording a new frame file using **Set Recovery System**, by default Silk Test Classic does not explicitly state that the parent of the window is a browser. To resolve this issue, add the "parent Browser" line to the frame file.

How the Non-Web Recovery System Closes Windows

The built-in recovery system restores the base state by making sure that the non-Web application is running, is not minimized, is active, and has no open windows except for the main window. To ensure that only the main window is open, the recovery system attempts to close all other open windows, using an internal procedure that you can customize as you see fit.

To make sure that there are no application windows open except the main window, the recovery system calls the built-in `CloseWindows` method. This method starts with the currently active window and attempts to close it using the sequence of steps below, stopping when the window closes.

1. If a `Close` method is defined for the window, call it.
2. Click the **Close** menu item on the system menu, on platforms and windows that have system menus.
3. Click the window's close box, if one exists.
4. If the window is a dialog box, type each of the keys specified by the `OPT_ CLOSE_DIALOG_KEYS` option and wait one second for the dialog box to close. By default, this option specifies the **Esc** key.
5. If there is a single button in the window, click that button.
6. Click each of the buttons specified by the `OPT_CLOSE_WINDOW_ BUTTONS` option. By default, this option specifies the **Cancel**, **Close**, **Exit**, and **Done** keys.
7. Select each of the menu items specified by the `OPT_CLOSE_WINDOW_ MENUS` option. By default, this option specifies the **File** > **Exit** and the **File** > **Quit** menu items.
8. If the closing of a window causes a confirmation dialog box to open, `CloseWindows` attempts to close the dialog box by clicking each of the buttons specified with the `OPT_CLOSE_CONFIRM_BUTTONS` option. By default, this option specifies the **No** button.

When the window, and any resulting confirmation dialog box, closes, `CloseWindows` repeats the preceding sequence of steps with the next window, until all windows are closed.

If any of the steps fails, none of the following steps is executed and the recovery system raises an exception. You may specify new window closing procedures.

In a Web application, you are usually loading new pages into the same browser, not closing a page before opening a new one.

How the Non-Web Recovery System Starts the Application

To start a non-Web application, the recovery system executes the `Invoke` method for the main window of the application. The `Invoke` method relies on the `sCmdLine` constant as recorded for the main window when you create a test frame.

For example, here is how a declaration for the `sCmdLine` constant might look for a sample Text Editor application running under Windows:

```
const sCmdLine = "c:\ProgramFiles\<SilkTest install directory>\SilkTest
\TextEdit.exe"
```

After it starts the application, the recovery system checks whether the main window is minimized, and, if it is, uses the `Restore` method to open the icon and restore the application to its proper size.

The limit on the `sCmdLine` constant is 8191 characters.

Modifying the Default Recovery System

The default recovery system is implemented in `defaults.inc`, which is located in the directory in which you installed Silk Test Classic. If you want to modify the default recovery system, instead of overriding some of its features, as described in Overriding the default recovery system, you can modify `defaults.inc`.

We cannot provide support for modifying `defaults.inc` or the results. We recommend that you do not modify `defaults.inc`. This file might change from version to version. As a result, if you manually modify `defaults.inc`, you will encounter issues when upgrading to a new version of Silk Test Classic.

If you decide to modify `defaults.inc`, be sure that you:

- Make a backup copy of the shipped `defaults.inc` file.
- Tell Technical Support when reporting problems that you have modified the default recovery system.

Overriding the Default Recovery System

The default recovery system specifies what Silk Test Classic does to restore the base state of your application. It also specifies what Silk Test Classic does whenever:

- A script file is first accessed.
- A script file is exited.
- A test case is about to begin.
- A test case is about to exit.

You can write functions that override some of the default behavior of the recovery system.

To override	Define the following
`DefaultScriptEnter`	`ScriptEnter`
`DefaultScriptExit`	`ScriptExit`
`DefaultTestCaseEnter`	`TestCaseEnter`
`DefaultTestCaseExit`	`TestCaseExit`
`DefaultTestPlanEnter`	`TestPlanEnter`
`DefaultTestPlanExit`	`TestPlanExit`

If `ScriptEnter`, `ScriptExit`, `TestcaseEnter`, `TestcaseExit`, `TestPlanEnter`, or `TestPlanExit` are defined, Silk Test Classic uses them instead of the corresponding default function. For example, you might want to specify that certain test files are copied from a server in preparation for running a script. You might specify such processing in a function called `ScriptEnter` in your test frame.

If you want to modify the default recovery system, instead of overriding some of its features, you can modify `defaults.inc`. We do not recommend modifying `defaults.inc` and cannot provide support for modifying `defaults.inc` or the results.

Example

If you are planning on overriding the recovery system, you need to write your own `TestCaseExit(Boolean bException)`. In the following example, `DefaultTestcaseExit()` is

called inside `TestCaseExit()` to perform standard recovery systems steps and the `bException` argument is passed into `DefaultTestCaseExit`().

```
if (bException)
  DefaultTestcaseExit(bException)
```

If you are not planning to call `DefaultTestcaseExit`() and plan to handle the error logging in your own way, then you can use the `TestcaseExit()` signature without any arguments.

Use the following function signature if you plan on calling `DefaultTestCaseExit(Boolean bException)` or if your logic depends on whether an exception occurred. Otherwise, you can simply use the function signature of `TestcaseExit()` without any arguments. For example, the following is from the description of the `ExceptLog()` function.

```
TestCaseExit (BOOLEAN bException)
if (bException)
  ExceptLog()
```

Here, `DefaultTestcaseExit()` is not called, but the value of `bException` is used to determine if an error occurred during the test case execution.

Handling Login Windows

Silk Test Classic handles login windows differently, depending on whether you are testing Web or client/server applications. These topics provide information on how to handle login windows in your application under test.

Handling Login Windows in Web Applications that Use the Classic Agent

This procedure describes how to handle web applications with different possible startup pages or dialog box objects that use the Classic Agent. For example,

- A Web application requires the user to login the first time he or she visits the site in a day (a non-persistent cookie). If the user has already logged in for this browser session, the user will not be prompted for user name and password again, as the "cookie" is still available with their authorization. This could be either a login web page or a dialog box.
- A dialog box that sometimes gives a "tip of the day" or reminds the user to perform some action because it is a certain date.
- A dialog box might popup asking the user whether it is okay to download a certificate, a Java module, or some other component.

In cases such as these, you can use the *sLocation* data-member from the `wMainWindow` object as a property. You can create a property and it will look exactly like a data-member and will be called like a data-member. When trying to retrieve information from a property the `Get` portion of the property is executed. And you can add code to deal with login Web pages here.

Here are the steps of what will happen when `DefaultBaseState` runs:

1. `DefaultBaseState` will try to retrieve the *sLocation* data-member and, as such, will execute the `Get` function.
2. The `Get` function that is part of the property will actually load the web page by putting the URL of the page into the **Location** comboBox that is part of the browser and pressing `Enter`. It will then wait for the browser to report to Silk Test Classic a ready state.
3. If the **Login** page exists rather than the page we were expecting, the user name and password will be entered and the `HtmlPushButton` **OK** will be clicked. Again, Silk Test Classic will wait for the browser to return to a ready state.
4. The `Get` function returns a NULL even though at the definition of the `Get` function it was specified that a STRING would be returned. If you were to return the URL, `DefaultBaseState` would load the page

again. Of course, since we have already dealt with login, it would work this time, but would add some more time into the process by loading the page again.

5. Although `DefaultBaseState` will not try to load the page, it will find it there and continue with the other steps of closing any open windows and setting the browser and Web page active.

   You can also handle unexpected and occasional dialog boxes in this way, by changing the *sLocation* data-member to a property and handling different possibilities through a Get function that is part of the property, or you can re-write the Close method. For expected security or login dialog boxes, you can set the *sUsername* and *sPassword* for the `wMainWindow` object.

```
Window BrowserChild RealPage
const PAGE_URL = http://www.somepage.com
property sLocation
STRING Get ( )

// actually load the page
Browser.SetActive ( )
Browser.Location.SetText (PAGE_URL)
Browser.Location.TypeKeys ("<Enter>")

// wait for the browser to be "ready" Browser.WaitForReady ( )
// if the Login page has shown up…
if Login.Exists ( )

// deal with it
Login.UserName.SetText (USERNAME)
Login.Password.SetText (PASSWORD)
Login.OK.Click ( )

// now wait for the browser to be ready
Browser.WaitForReady ( )

//this way DefaultBaseState will not try to load the page again
return NULL
```

Handling Login Windows in Non-Web Applications that Use the Classic Agent

Although a non-Web application's main window is usually displayed first, it is also common for a login or security window to be displayed before the main window.

Use the `wStartup` constant and the `Invoke` method

To handle login windows, record a declaration for the login window, set the value of the `wStartup` constant, and write a new `Invoke` method for the main window that enters the appropriate information into the login window and dismisses it. This enables the `DefaultBaseState` routine to perform the actions necessary to get past the login window.

You do not need to use this procedure for splash screens, which disappear on their own.

1. Open the login window that precedes the application's main window.
2. Open the test frame.
3. Click **Record** > **Window Declarations** to record a declaration for the window.
4. Point to the title bar of the window and then press **Ctrl+Alt**. The declaration is captured in the **Record Window Declarations** dialog box.
5. Click **Paste to Editor** to paste the declaration into the test frame.
6. In the **Record Window Declarations** dialog box, click **Close**.
7. Close your application.

8. In your test frame file, find the stub of the declaration for the `wStartup` constant, located at the top of the declaration for the main window:

```
// First window to appear when application is invoked
// const wStartup = ?
```

9. Complete the declaration for the `wStartup` constant by:
   - Removing the comment characters, the two forward slash characters, at the beginning of the declaration.
   - Replacing the question mark with the identifier of the login window, as recorded in the window declaration for the login window.
10. Click the `wStartup` constant and then click **Record** > **Method**.
11. On the **Record Method** dialog box, from the **Method Name** list box, select **Invoke**.
12. Open your application, but do not dismiss the login window.
13. Click **Start Recording**. Silk Test Classic is minimized and your application and the **Silk Test Record Status** dialog box open.
14. Perform and the record the actions that you require.
15. On the **Silk Test Record Status** dialog box, click **Done**. The **Record Method** dialog box opens with the actions you recorded translated into 4Test statements.
16. On the **Record Method** dialog box, click **OK** to paste the code into your include file.
17. Edit the 4Test statements that were recorded, if necessary.
18. Define an `Invoke` method in the main window declaration that calls the built-in `Invoke` method and additionally performs any actions required by the login window, such as entering a name and password.

    After following this procedure, your test frame might look like this:

```
window MainWin MyApp
   tag "My App"
   const wStartup = Login

   // the declarations for the MainWin should go here
   Invoke ()
      derived::Invoke ()
      Login.Name.SetText ("Your name")
      Login.Password.SetText ("password")
      Login.OK.Click ()

window DialogBox Login
   tag "Login"

   // the declarations for the Login window go here
   PushButton OK
      tag "OK"
```

    About the derived keyword and scope resolution operator

    Notice the statement `derived::Invoke ( )`. That statement uses the derived keyword followed by the scope resolution operator (`::`) to call the built-in Invoke method, before performing the operations needed to fill in and dismiss the login window.

Handling Browser Pop-up Windows in Tests that Use the Classic Agent

Browser pop-up windows are recognized as instances of `Browser`.

When the popup window is active, it is seen as Browser and the original browser is seen as Browser 2. In order to make `DefaultBaseState()` close the pop-up window instead of the original browser, add the following line to the end of the test case:

```
Browser2.SetActive()
```

This is the standard way of ensuring that the pop-up becomes Browser2 and is closed by `DefaultBaseState()`.

Specifying Windows to be Left Open for Tests that Use the Classic Agent

By default, the non-web recovery system closes all windows in your test application except the main window. To specify which windows, if any, need to be left open, such as a child window that is always open, use the `lwLeaveOpen` constant.

lwLeaveOpen constant

When you record and paste the declarations for your application's main window, the stub of a declaration for the `lwLeaveOpen` constant is automatically included, as shown in this example:

```
// The list of windows the recovery system is to leave open
// const lwLeaveOpen = {?}
```

To complete the declaration for the `lwLeaveOpen` constant:

1. Replace the question mark in the comment with the 4Test identifiers of the windows you want to be left open. Separate each identifier with a comma.
2. Remove the comment characters (the two forward slash characters) at the beginning of the declaration.

> **Example**
>
> he following 4Test code shows how to set the `lwLeaveOpen` constant so that the recovery system leaves open the window with the 4Test identifier `DocumentWindow` when it restores the base state.
>
> ```
> const lwLeaveOpen = {DocumentWindow}
> ```

Specifying New Window Closing Procedures

When the recovery system cannot close a window using its normal procedure, you can reconfigure it in one of two ways:

- If the window can be closed by a button press, key press, or menu selection, specify the appropriate option either statically in the **Close** tab of the **Agent Options** dialog box or dynamically at runtime.
- Otherwise, record a `Close` method for the window.

This is only for classes derived from the `MoveableWin` class: `DialogBox`, `ChildWin`, and `MessageBox`. Specifying window closing procedures is not necessary for web pages, so this does not apply to `BrowserChild` objects/classes.

Specifying Buttons, Keys, and Menus that Close Windows

Specify statically

To specify statically the keys, menu items, and buttons that the non-Web recovery system should use to close all windows, choose **Options** > **Agent** and then click the **Close** tab.

The **Close** tab of the **Agent Options** dialog box contains a number of options, each of which takes a comma-delimited list of character string values.

Specify dynamically

As you set close options in the **Agent Options** dialog box, the informational text at the bottom of the dialog box shows the 4Test command you can use to specify the same option from within a script; add this 4Test command to a script if you need to change the option dynamically as a script is running.

Specify for individual objects

If you want to specify the keys, menu items, and buttons that the non-web recovery system should use to close an individual dialog box, define the appropriate variable in the window declaration for the dialog box:

- `lsCloseWindowButtons`
- `lsCloseConfirmButtons`
- `lsCloseDialogKeys`
- `lsCloseWindowMenus`

This is only for classes derived from the `MoveableWin` class: `DialogBox`, `ChildWin`, and `MessageBox`. Specifying window closing procedures is not necessary for web pages, so this does not apply to `BrowserChild` objects/classes.

Recording a Close Method for Tests that Use the Classic Agent

To specify the keys, menu items, and buttons that the non-Web recovery system uses to close an individual dialog box, record a `Close` method to define the appropriate variable in the window declaration for the dialog box.

1. Open your application.
2. Open the application's test frame file.
3. Place the insertion point on the window declaration for the dialog box.
4. Choose **Record** > **Method** .
5. From the **Method Name** list, select **Close** .
6. Click **Start Recording**. Silk Test Classic displays the **Record Status** dialog box, which indicates that you can begin recording the `Close` method. The **Status field** flashes the word `Recording`.
7. When you have finished recording the `Close` method, click **Done** on the **Record Status** dialog box. Silk Test Classic opens the **Record Method** dialog box. The **Method Code** field contains the 4Test code that you have recorded.
8. Click **OK** to close the **Record Method** dialog box and paste the new `Close` method in the declaration for the dialog box.

You can also specify buttons, keys, and menus that close windows. This is only for classes derived from the `MoveableWin` class: `DialogBox`, `ChildWin`, and `MessageBox`. Specifying window closing procedures is not necessary for web pages, so this does not apply to `BrowserChild` objects/classes.

Test Plans

A test plan usually is a hierarchically-structured document that describes the test requirements and contains the statements, 4Test scripts, and test cases that implement the test requirements. A test plan is displayed in an easy-to-read outline format, which lists the test requirements in high-level prose descriptions. The structure can be flat or many levels deep.

Indentation and color indicate the level of detail and various test plan elements. Large test plans can be divided into a master plan and one or more sub-plans. A test plan file has a .pln extension, such as `find.pln`.

Structuring your test plan as an hierarchical outline provides the following advantages:

- Assists the test plan author in developing thoughts about the test problem by promoting and supporting a top-down approach to test planning.
- Yields a comprehensive inventory of test requirements, from the most general, through finer and finer levels of detail, to the most specific.
- Allows the statements that actually implement the tests to be shared by group descriptions or used by just a single test description.
- Provides reviewers with a framework for evaluating the thoroughness of the plan and for following the logic of the test plan author.
- If you are using the test plan editor, the first step in creating automated tests is to create a test plan. If you are not using the test plan editor, the first step is creating a test frame.

Structure of a Test Plan

A test plan is made up of the following elements, each of which is identified by color and indentation on the test plan.

Element	Description	Color
Comment	Provide documentation throughout the test plan; preceded by `//`.	Green
Group Description	High level line in the test requirements outline that describes a group of tests.	Black
Test Description	Lowest level line describing a single test case; is a statement of the functionality to be tested by the associated test case.	Blue
Test Plan Statement	Used to provide script name, test case name, test data, or `include` statement.	Red when a sub plan is not expanded. Magenta statement when sub-plan is expanded

A statement placed at the group description level applies to all the test descriptions contained by the group. Conversely, a statement placed at the test description level applies only to that test description. Levels in the test plan are represented by indentation.

Because there are many ways to organize information, you can structure a test plan using as few or as many levels of detail as you feel are necessary. For example, you can use a list structure, which is a list of test descriptions with no group description, or a hierarchical structure, which is a group description and test description. The goal when writing test plans is to create a top-down outline that describes all of the test requirements, from the most general to the most specific.

Overview of Test Plan Templates

Because a test plan is initially empty, you may want to insert a template, which is a hierarchical outline you can use as a guide when you create a new test plan.

The template contains placeholders for each GUI object in your application. Although you may not want to structure the test plan in a way which mirrors the hierarchy of your application's GUI, this can be a good starting point if you are new to creating test plans.

In order to be able to insert a template, you must first record a test frame, which contains declarations for each of the GUI objects in your application.

Example Outline for Word Search Feature

Because a test plan is made up of a large amount of information, a structured, hierarchical outline provides an ideal model for organizing and developing the details of the plan. You can structure an outline using as few or as many levels of detail as you feel necessary.

The following is a series of sample outlines, ranging from a simple *list structure* to a more specific *hierarchical structure*. For completeness, each of the plans also shows the script and test case statements that link the descriptions to the 4Test scripts and test cases that implement the test requirements.

For example, consider the **Find** dialog box from the Text Editor application, which allows a user to search in a document. A user enters the characters to search for in the **Find What** text box, checks the **Case sensitive** check box to consider case, and clicks either the **Up** or **Down** radio button to indicate the direction of the search.

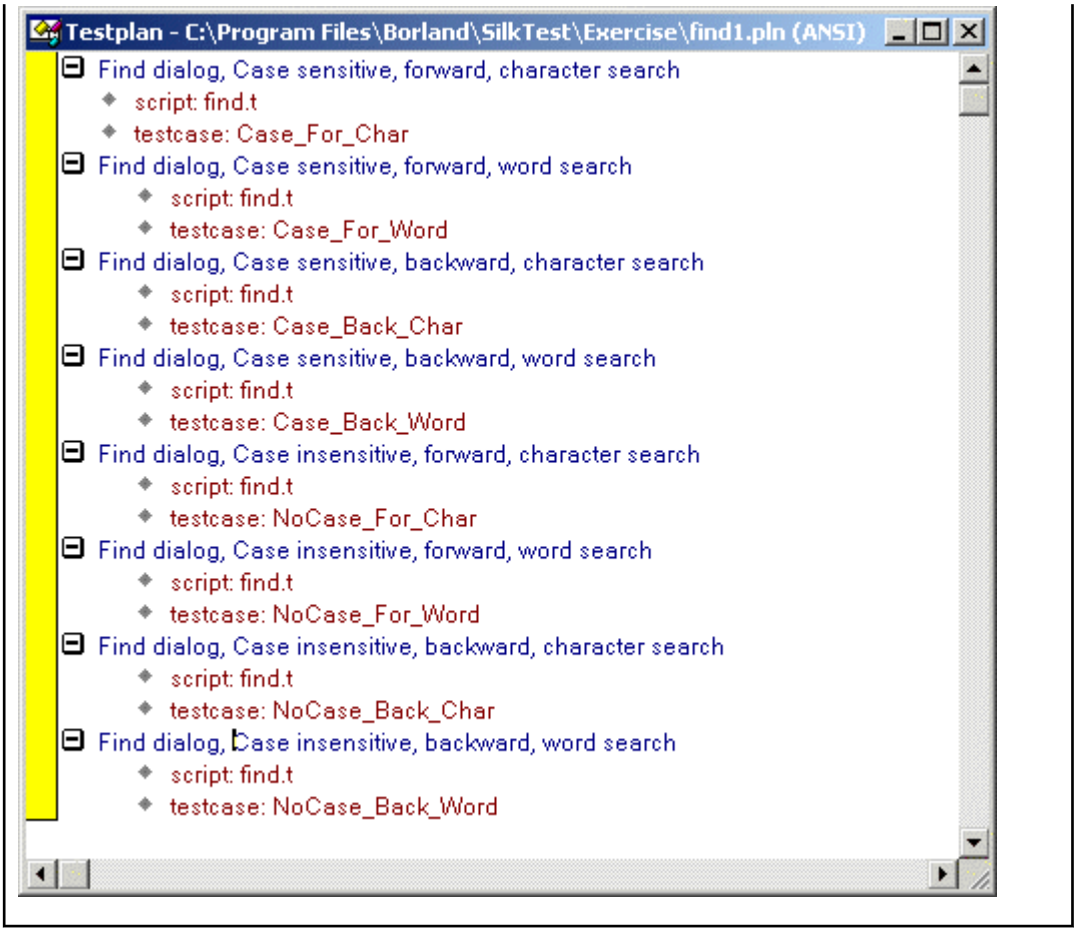
List Structure

At its simplest, an outline is a hierarchy with just a single level of detail. In other words, it is a list of test descriptions, with no group descriptions.

Using the list structure, each test is fully described by a single line, which is followed by the script and test case that implement the test. You may find this style of plan useful in the beginning stages of test plan design, when you are brainstorming the list of test requirements, without regard for the way in which the test requirements are related. It is also useful if you are creating an ad hoc test plan that runs a set of unrelated 4Test scripts and test cases.

Example for List Structure

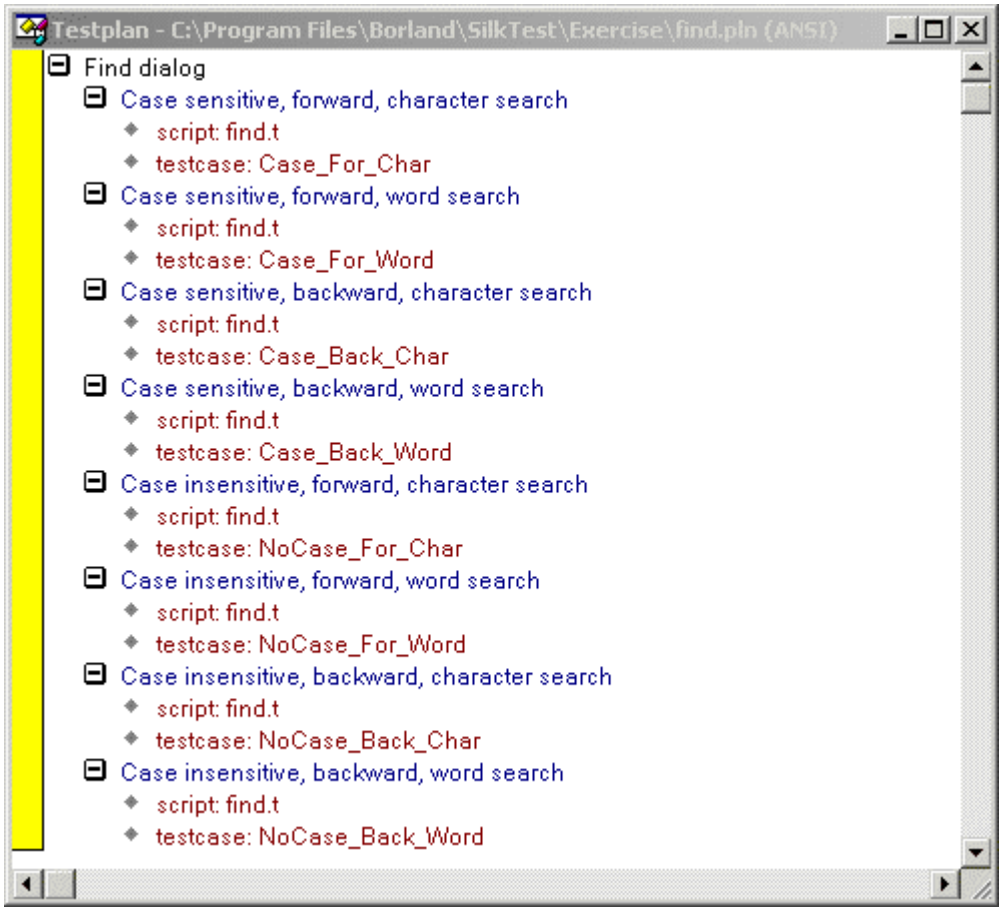
For example:



Hierarchical Structure

The following test plan has a single level of group description, preceding the level that contains each of the test descriptions. The group description indicates that all the tests are for the **Find** dialog box.

As the figure shows, the test plan editor indicates levels in the outline with indentation. Each successive level is indented one level to the right. The minus icons indicate that each of the levels is fully expanded. By clicking on the minus icon at any level, you collapse the branch below that level. When working with large test plans, collapsing and expanding test plan detail makes it easy to see as much or as little of the test plan as you need. You could continue this test plan by adding a second level of group description, indicating whether or not the tests in the group are case sensitive, and even more detail by adding a third level of group descriptions which indicate whether the tests in the group search in the forward or backward direction.



Converting a Results File to a Test Plan

Using Silk Test Classic, you can convert a results file into a test plan. This is useful when converting suite-based tests into test plans.

1. Open a results file that was generated by Silk Test Classic, not one generated by the test plan editor from a test plan.
2. Click **Results** > **Convert to Plan**.
3. Select the results file you want to convert, which is typically the most recent, and click **OK**. The test plan editor converts the results file to a test plan.

   When creating a test plan from a results file generated for a script, the test plan editor uses the # symbol so that when this test plan is run, the `testdata` statement doubles as description. Since the results file was for a script, not a test plan, it does not contain any group or test case descriptions. The # symbol can be used with any test plan editor statement so that the statement will double as description.

Working with Test Plans

This section describes how you can work with test plans.

Creating a New Test Plan

1. Click **File** > **New**.

2. Click **Test plan** and click **OK**. An empty test plan window opens.
3. Create your test plan and then click **File** > **Save**.
4. Specify the name and location in which to save the file, and then click **OK**.
5. If you are working within a project, Silk Test Classic prompts you to add the file to the project. Click **Yes** if you want to add the file to the open project, or **No** if you do not want to add this file to the project.

Before you can begin testing, you must enable extensions for applications you plan to test on both the target machine and the host machine.

Indent and Change Levels in an Outline

You can use menu, keyboard, or toolbar commands to enter or change group and test descriptions as you are typing them. The following table summarizes the commands:

Action	Menu Item	Key
Indent one level	Outline/Move Right	**ALT + forward arrow**
Outdent one level	Outline/Move Left	**ALT + back arrow**
Swap with line above	Outline/Transpose Up	**ALT + up arrow**
Swap with line below	Outline/Transpose Down	**ALT + down arrow**

Each command acts on the current line or currently selected lines.

Silk Test Classic ignores comments when compiling, with the exception of functions and test cases. Comments within functions and test cases must be within the scope of the function/test case. If a comment is outdented beyond the scope of the function/test case, the compiler assumes that the function/test case has ended. As long as comments do not violate the function/test case scope, they can be placed anywhere on a line.

Note: Comments beyond the scope can also impact expand/collapse functionality and may prevent a function/test case from being fully expanded/collapsed. We recommend that you keep comments within scope.

Adding Comments to Test Plan Results

You can add comments to your test plans which will display in the results when you run your tests. You can annotate your tests with such comments to ease the interpretation of the test results.

To add a comment to a test plan, include the following statement in the test plan:

```
comment: Your comment text
```

For example, running the following piece of a test plan:

```
Find dialog
  Get the default button
    comment: This test should return Find.FindNext
    script: find.t
    testcase: GetButton
```

produces the following in the results file:

```
Find dialog
  Get the default button
    Find.FindNext
    comment: This test should return Find.FindNext
```

Note: You can also preface lines in all 4Test files with // to indicate a single-line comment. Such comments do not display in test plan results.

Documenting Manual Tests in the Test Plan

Your QA department might do some of its testing manually. You can document the manual testing in the test plan. In this way, the planning, organization, and reporting of all your testing can be centralized in one place. You can describe the state of each of your manual tests. This information is used in reports.

To indicate that a test description in the test plan is implemented with a manual test, use the value `manual` in the `testcase` statement, as in:

```
testcase: manual
```

By default, whenever you generate a report, it includes information on the tests run for that results file, plus the current results of any manual tests specified in the test plan. If the manual test results are subsequently updated, the next time you generate the report, it incorporates the latest manual results. However, this might not be what you want. If you want the report to use a snapshot of manual results, not the most recent manual results, merge the results of manual tests into the results file.

Describing the State of a Manual Test

1. Open a test plan containing manual tests.
2. Click **Testplan** > **Run Manual Tests**.
3. Select a manual test from the **Update Manual Tests** dialog box and document it. The **Update Manual Tests** dialog box lists all manual tests in the current test plan.

Mark the test complete	Click the **Complete** option button. `Complete` means that a test has been defined. A manual test marked here as `Complete` will be tabulated as complete in Completion reports.
Indicate whether the test passed or failed	1. Click the **Has been run** option button. 2. Select **Passed** or **Failed**. 3. Specify when the test was run and optionally, specify the machine. To specify when the test was run, use the following syntax: `YYYY-MM-DD HH:MM:SS` Hours, minutes, and seconds are optional. For example, enter `2006-01-10` to indicate that the test was run Jan 10, 2006. Manual tests marked as `Passed` or `Failed` will be tabulated as such in Pass/Fail reports, as long as you have also specified the time at which they were run. A test marked `Has been run` is also considered complete in Completion reports.
Add any comments you want about the test	Fill in the **Comments** text box.

Inserting a Template

1. Click **Testplan** > **Insert Template**. The **Insert Testplan Template** dialog box, which lists all the GUI objects declared in your test frame, opens.
2. Select each of the GUI objects that are related to the application features you want to test.

   Because this is a multi-select list box, the objects do not have to be contiguous.

   For each selected object, Silk Test Classic inserts two lines of descriptive text into the test plan.

For example, the test plan editor would create the following template for the **Find** dialog box of the Text Editor application:

```
Tests for DialogBox Find
Tests for StaticText FindWhatText
(Insert tests here)
Tests for TextField FindWhat
(Insert tests here)
Tests for CheckBox CaseSensitive
(Insert tests here)
Tests for StaticText DirectionText
(Insert tests here)
Tests for PushButton FindNext
(Insert tests here)
Tests for PushButton Cancel
(Insert tests here)
Tests for RadioList Direction
(Insert tests here)
```

Changing Colors in a Test Plan

You can customize your test plan so that different test plan components display in unique colors.

To change the default colors:

1. Click **Options** > **Editor Colors**.
2. On the **Editor Colors** dialog box, select the outline editor item you want to change in the **Editor Item** list box at the left of the dialog box.
3. Apply a color to the item by selecting a pushbutton from the list of predefined colors or create a new color to apply by selecting the red, green, and blue values that compose the color.

Default color	Component	Description
Blue	Test description	Lowest level of the hierarchical test plan outline that describes a single testcase.
Red	Test plan statement	Link scripts, test cases, test data, closed sub-plans, or an include file (such as a test frame) to the test plan.
Magenta	Include statement when sub-plan is open	Sub-plans to be included in a master plan.
Green	Comment	Additional user information that is incidental to the outline; preceded by double slashes (//); provides documentation throughout the test plan.
Black	Other line (group description)	Higher level lines of the hierarchical test plan outline that describe a group of tests; may be several levels in depth.

Linking the Test Plan to Scripts and Test Cases

After you create your test plan, you can associate the appropriate 4Test scripts and test cases that implement your test plan. You create this association by inserting `script` and `testcase` statements in the appropriate locations in the test plan.

There are three ways to link a script or test case to a test plan:

- Linking a description to a script or test case using the **Testplan Detail** dialog box if you want to automate the process of linking scripts and test cases to the test plan.
- Linking to a test plan manually.
- Linking scripts and test cases to a test plan: the test plan editor automatically inserts the `script` and `testcase` statements into the plan once the recording is finished, linking the plan to the 4Test code.

You can insert a `script` and `testcase` statement for each test description, although placing a statement at the group level when possible eliminates redundancy in the test plan. For example, since it is usually

good practice to place all the test cases for a given application feature into a single script file, you can reduce the redundancy in the test plan by specifying the `script` statement at the group level that describes that feature.

You can also insert a `testcase` statement at the group level, although doing so is only appropriate when the test case is data driven, meaning that it receives test data from the plan. Otherwise the same test case would be called several times with no difference in outcome.

Working with Large Test Plans

For large or complicated applications, the test plan can become quite large. This raises the following issues:

Issue	Solution
How to keep track of where you are in the test plan and what is in scope at that level.	Use the **Testplan Detail** dialog box.
How to determine which portions of the test plan have been implemented.	Produce a **Completion** report.
How to allow several staff members to work on the test plan at the same time.	Structure your test plan as a master plan with one or more sub-plans.

This section describes how you can divide your test plan into a master plan with one or more sub-plans to allow several staff members to work on the test plan at the same time.

Determining Where Values are Defined in a Large Test Plan

1. Place the insertion point at the relevant point in the test plan and click **Testplan** > **Detail**. The **Testplan Detail** dialog box opens.
2. Click the level in the list box at the top of the **Testplan Detail** dialog box, to see just the set of symbols, attributes, and statements that are defined on a particular level.
3. Once you find the level at which a symbol, attribute, or statement was defined, you can change the value at that level, causing the inherited value at the lower levels to change also.

Dividing a Test Plan into a Master Plan and Sub-Plans

If several engineers in your QA department will be working on a test plan, it makes sense to break up the plan into a master plan and sub-plans. This approach allows multi-user access, while at the same time maintaining a single point of control for the entire project.

The master plan contains only the top few levels of group descriptions, and the sub-plans contain the remaining levels of group descriptions and test descriptions. Statements, attributes, symbols, and test data defined in the master plan are accessible within each of the sub-plans.

Sub-plans are specified with an `include` statement. To expand the sub-plan files so that they are visible within the master plan, double-click in the left margin next to the `include` statement. Once a sub-plan is expanded inline, the sub-plan statement changes from red (the default color for statements) to magenta, indicating that the line is now read-only and that the sub-plan is expanded inline. At the end of the expanded sub-plan is the `<eof>` marker, which indicates the end of the sub-plan file.

Creating a Sub-Plan

You create a sub-plan in the same way you create any test plan: by opening a new test plan file and entering the group descriptions, test descriptions, and the test plan editor statements that comprise the sub-plan, either manually or using the **Testplan Detail** dialog.

Copying a Sub-Plan

When you copy and paste the include statement and the contents of an open include file, note that only the include statement will be pasted.

To view the contents of the sub-plan, open the pasted include file by clicking **Include** > **Open** or double-click the margin to the left of the include statement.

Opening a Sub-Plan

Open the sub-plan from within the master plan. To do this, you can either:

- double-click the margin to the left of the include statement or
- highlight the include statement and choose **Include** > **Open**. (Compiling a script also automatically opens all sub-plans.)

If a sub-plan does not inherit anything (that is, statements, attributes, symbols, or data) from the master plan, you can open the sub-plan directly from the **File** > **Open** dialog box.

Connecting a Sub-Plan with a Master Plan

To connect the master plan to a sub-plan file, you enter an `include` statement in the master plan at the point where the sub-plan logically fits. The `include` statement cannot be entered through the **Testplan Detail** dialog box; you must enter it manually.

The `include` statement uses this syntax:

```
include:  myinclude.pln
```

where `myinclude` is the name of the test plan file that contains the sub-plan.

If you enter the `include` statement correctly, it displays in red, the default color used for the test plan editor statements. Otherwise, the statement displays in blue or black, indicating a syntax error (the compiler is interpreting the line as a description, not a statement).

Refreshing a Local Copy of a Sub-Plan

When another user modifies a sub-plan, those changes are not automatically reflected in your read-only copy of the sub-plan. Once the other user has released the lock on the sub-plan, there are two ways to refresh your copy:

1. Close and then reopen the sub-plan.
2. Acquire a lock for the sub-plan.

Sharing a Test Plan Initialization File

All QA engineers working on a test plan that is broken up into a master plan and sub-plans must use the same test plan initialization file.

To share a test plan initialization file:

1. Click **Options** > **General**.
2. On the **General Options** dialog box, specify the same file name in the **Data File for Attributes and Queries** text box.

Saving Changes

When you finish editing, choose **Include** > **Save** to save the changes to the sub-plan.

Include > **Save** saves changes to the current sub-plan while **File** > **Save** saves all open master plans and sub-plans.

Overview of Locks

When first opened, a master plan and its related sub-plans are read-only. This allows many users to open, read, run, and generate reports on the plan. When you need to edit the master plan or a sub-plan, you must first acquire a lock, which prevents others from making changes that conflict with your changes.

Acquiring and Releasing a Lock

Acquire a lock	Place the cursor in or highlight one or more sub-plans and then choose **Include** > **Acquire Lock**. The bar in the left margin of the test plan changes from gray to yellow.
Release a lock	Select **Include** > **Release Lock**. The margin bar changes from yellow to gray.

Generating a Test Plan Completion Report

To measure your QA department's progress in implementing a large test plan, you can generate a completion report. The completion report considers a test complete if the test description is linked to a test case with two exceptions:

- If the test case statement invokes a data-driven test case and a symbol being passed to the data-driven test case is assigned the value ? (undefined), the test is considered incomplete.
- If the test case is manual and marked as Incomplete in the **Update Manual Tests** dialog box, the test is considered incomplete. A manual test case is indicated with the `testcase:manual` syntax.

To generate a test plan completion report:

1. Open the test plan on which you want to report.
2. Click **Testplan** > **Completion Report** to display the **Testplan Completion Report** dialog box.
3. In the **Report Scope** group box, indicate whether the report is for the entire plan or only for those tests that are marked.
4. To subtotal the report by a given attribute, select an attribute from the **Subtotal by Attribute** text box.
5. Click **Generate**.

   The test plan editor generates the report and displays it in the lower half of the dialog box. If the test plan is structured as a master plan with associated sub-plans, the test plan editor opens any closed sub-plans before generating the report.

   You can:

   - Print the report.
   - Export the report to a comma-delimited ASCII file. You can then bring the report into a spreadsheet application that accepts comma-delimited data.

- Chart (graph) the report, just as you can chart a Pass/Fail report.

Adding Data to a Test Plan

This section describes how you can add data to a test plan.

Specifying Unique and Shared Data

If a data value is unique to a single test description	You should place it in the plan at the same level as the test description, using the `testdata` statement. You can add the `testdata` statement using the **Testplan Detail** dialog box or type the `testdata` statement directly into the test plan.
If data is common to several tests	You can factor out the data that is common to a group of tests and define it at a level in the test plan where it can be shared by the group. To do this, you define symbols and assign them values. Using symbols results in less redundant data, and therefore, less maintenance.

Adding Comments in the Test Plan Editor

Use two forward slash characters to indicate that a line in a test plan is a comment. For example:

```
// This is a comment
```

Comments preceded by `//` do not display in the results file. You can also specify comments using the comment statement; these comments will display in the results files.

Testplan Editor Statements

You use the test plan editor keywords to construct statements, using this syntax:

```
keyword : value
```

`keyword`: One of the test plan editor keywords.

`value`: A comment, script, test case, include file, attribute name, or data value.

For example, this statement associates the script `myscript.t` with the plan:

```
script : myscript.t
```

Spaces before and after the colon are optional.

The # Operator in the Testplan Editor

When a # character precedes a statement, the statement will double as a test description in the test plan. This helps eliminate possible redundancies in the test plan. For example, the following test description and script statement:

```
Script is test.t
        script:test.t
```

can be reduced to one line in the test plan:

```
#script: test.t
```

The test plan editor considers this line an executable statement as well as a description. Any statements that follow this "description" in the test plan and that trigger test execution must be indented.

Using the Testplan Detail Dialog Box to Enter the testdata Statement

1. Place the insertion point at the end of the test description. If a `testdata` statement is not associated with a test description, the compiler generates an error.
2. Click **Testplan** > **Detail**. To provide context, the multi-line list box at the top of the **Testplan Detail** dialog box displays the line in the test plan that the cursor was on when the dialog box was invoked, indicated by the black arrow icon. If the test case and script associated with the current test description are inherited from a higher level in the test plan, they are shown in blue; otherwise, they are shown in black.
3. Enter the data in the **Test Data** text box, separating each data element with a comma.

   Remember, if the test case expects a record, you need to enclose the list of data with the list constructor operator (the curly braces); otherwise, Silk Test Classic interprets the data as individual variables, not a record, and will generate a data type mismatch compiler error.
4. Click **OK**. Silk Test Classic closes the **Testplan Detail** dialog box and enters the testdata statement and data values in the plan.

Entering the testdata Statement Manually

1. Open up a new line after the test description and indent the line one level.
2. Enter the testdata statement as follows.

   - If the test case expects one or more variables, use this syntax: `testdata: data [,data]`, where data is any valid 4Test expression.
   - A record, use the same syntax as above, but open and close the list of record fields with curly braces: `testdata: {data [,data]}`, where data is any valid 4Test expression.

   Be sure to follow the testdata keyword with a colon. If you enter the keyword correctly, the statement displays in dark red, the default color. Otherwise, the statement displays in either blue or black, indicating the compiler is interpreting the line as a description.

Linking Test Plans

This section describes how Silk Test Classic handles linking from a test plan to a script or test case.

Linking a Description to a Script or Test Case using the Testplan Detail Dialog Box

1. Place the insertion cursor on either a test description or a group description.
2. Click **Testplan** > **Detail**. The test plan editor invokes the **Testplan Detail** dialog box, with the **Test Execution** tab showing. The multi-line list box at the top of the dialog box displays the line in the test plan that the cursor was on when the dialog box was invoked, as well as its ancestor lines. The black arrow icon indicates the current line. The current line appears in black and white, and the preceding lines display in blue.
3. If you:

   - know the names of the script and test case, enter them in the **Script** and **Testcase** fields, respectively.
   - are unsure of the script name, click the **Scripts** button to the right of the **Script** field to browse for the script file.

4. On the **Testplan Detail - Script** dialog box, navigate to the appropriate directory and select a script name by double-clicking or by selecting and then clicking **OK**. Silk Test Classic closes the **Testplan Detail - Script** dialog box and enters the script name in the **Script** field.
5. Click the **Testcases** button to the right of the **Testcase** field, to browse for the test case name.

   The **Testplan Detail – Testcase** dialog box shows the names of the test cases that are contained in the selected script. Test cases are listed alphabetically, not in the order in which they occur in the script.
6. Select a test case from the list and click **OK**.
7. Click **OK**. The script and test case statements are entered in the plan.

If you feel comfortable with the syntax of the test plan editor statements and know the locations of the appropriate script and test case, you can enter the script and test case statements manually.

Linking a Test Plan to a Data-Driven Test Case

To link a group of test descriptions in the plan with a data-driven test case, add the test case declaration to the group description level. There are three ways to do this:

- Linking a test case or script to a test plan using the **Testplan Detail** dialog box to automate the process.
- Link to a test plan manually.
- Record the test case from within the test plan.

Linking to a Test Plan Manually

If you feel comfortable with the syntax of the test plan editor statements and know the locations of the appropriate script and test case, you can enter the `script` and `testcase` statements manually.

1. Place the insertion cursor at the end of a test or group description and press **Enter** to create a new line.
2. Indent the new line one level.
3. Enter the script and/or test case statements using the following syntax:

   ```
   script:
   scriptfilename.t testcase:
   testcasename
   ```

   Where script and testcase are keywords followed by a colon, `scriptfilename.t` is the name of the script file, and `testcasename` is the name of the test case.

   If you enter a statement correctly, it displays in dark red, the default color used for statements. If not, it will either display in blue, indicating the line is being interpreted as a test description, or black, indicating it is being interpreted as a group description.

Linking a Test Case or Script to a Test Plan using the Testplan Detail Dialog Box

The **Testplan Detail** dialog box automates the process of linking to scripts and test cases. It lets you browse directories and select script and test case names, and it enters the correct the test plan editor syntax into the plan for you.

Linking the Test Plan to Scripts and Test Cases

After you create your test plan, you can associate the appropriate 4Test scripts and test cases that implement your test plan. You create this association by inserting `script` and `testcase` statements in the appropriate locations in the test plan.

There are three ways to link a script or test case to a test plan:

- Linking a description to a script or test case using the **Testplan Detail** dialog box if you want to automate the process of linking scripts and test cases to the test plan.
- Linking to a test plan manually.
- Linking scripts and test cases to a test plan: the test plan editor automatically inserts the `script` and `testcase` statements into the plan once the recording is finished, linking the plan to the 4Test code.

You can insert a `script` and `testcase` statement for each test description, although placing a statement at the group level when possible eliminates redundancy in the test plan. For example, since it is usually good practice to place all the test cases for a given application feature into a single script file, you can reduce the redundancy in the test plan by specifying the `script` statement at the group level that describes that feature.

You can also insert a `testcase` statement at the group level, although doing so is only appropriate when the test case is data driven, meaning that it receives test data from the plan. Otherwise the same test case would be called several times with no difference in outcome.

Example of Linking a Test Plan to a Test Case

For example, consider the data driven test case FindTest, which takes a record of type `SEARCHINFO` as a parameter:

```
type SEARCHINFO is record
   STRING  sText      // Text to type in document window
   STRING  sPos       // Starting position of search
   STRING  sPattern   // String to look for
   BOOLEAN bCase      // Case-sensitive or not
   STRING  sDirection // Direction of search
   STRING  sExpected  // The expected match

testcase FindTest (SEARCHINFO Data)
   TextEditor.File.New.Pick ()
   DocumentWindow.Document.TypeKeys (Data.sText + Data.sPos)
   TextEditor.Search.Find.Pick ()
   Find.FindWhat.SetText (Data.sPattern)
   Find.CaseSensitive.SetState (Data.bCase)
   Find.Direction.Select (Data.sDirection)
   Find.FindNext.Click ()
   Find.Cancel.Click ()
   DocumentWindow.Document.VerifySelText ({Data.sExpected})
   TextEditor.File.Close.Pick ()
   MessageBox.No.Click ()
```

The following test plan is associated with the `FindTest testcase` (the `testcase` statement is highlighted for emphasis). The statement occurs at the **Find** dialog group description level, so that each of the test descriptions in the group can call the test case, passing it a unique set of data:

```
Testplan FindTest.pln

Find dialog
script: findtest.t
testcase: FindTest
. . . .
```

Categorizing and Marking Test Plans

This section describes how you can work with selected tests in a test plan.

Marking a Test Plan

Marks are temporary denotations that allow you to work with selected tests in a test plan. For example, you might want to run only those tests that exercise a particular area of the application or to report on only the tests that were assigned to a particular QA engineer. To work with selected tests rather than the entire test plan, you denote or **mark** those tests in the test plan.

Marks can be removed at any time, and last only as long as the current work session. You can recognize a marked test case by the black stripe in the margin.

You can mark test cases by:

Choice	Select the individual test description, group description, or entire plan that you want to mark, and then choosing the appropriate marking command on the **Testplan** menu.
Query	You can also mark a test plan according to a certain set of characteristics it possesses. This is called marking by query. You build a query based on one or more specific test characteristics; its script file, data, symbols, or attributes, and then mark those tests that match the criteria set up in the query. For example, you might want to mark all tests that live in the `find.t` script and that were created by the developer named Peter. If you name and save the query, you can reapply it in subsequent work sessions without having to rebuild the query or manually remark the tests that you're interested in working with.
Test failure	After running a test plan, the generated results file might indicate test failures. You can mark these failures in the plan by selecting **Results** > **Mark Failures in Plan**. You then might fix the errors and re-run the failed tests.

How the Marking Commands Interact

When you apply a mark using the **Mark** command, the new mark is added to existing marks.

When you mark tests through the query marking commands, the test plan editor by default clears all existing marks before running the query. **Mark by Named Query** supports sophisticated query combinations, and it would not make sense to retain previous marks. However, **Mark by Query**, which allows one-time-only queries, lets you override the default behavior and retain existing marks.

To retain existing marks, uncheck the **Unmark All Before Query** check box in the **Mark by Query** dialog box.

Marking One or More Tests

To mark:

A single test	Place the cursor on the test description and click **Testplan** > **Mark**.
A group of related tests	Place the cursor on the group description and click **Testplan** > **Mark**. The test plan editor marks the group description, its associated statements, and all test descriptions and statements subordinate to the group description.
Two or more adjacent tests and their subordinate tests	Select the test description of the adjacent tests and click **Testplan** > **Mark**. The test plan editor marks the test descriptions and statements of each selected test and any subordinate tests.

Printing Marked Tests

1. Click **File** > **Print**.
2. In the **Print** dialog box, make sure the **Print Marked Only** check box is checked, as well as any other options you want.

3. Click **OK**.

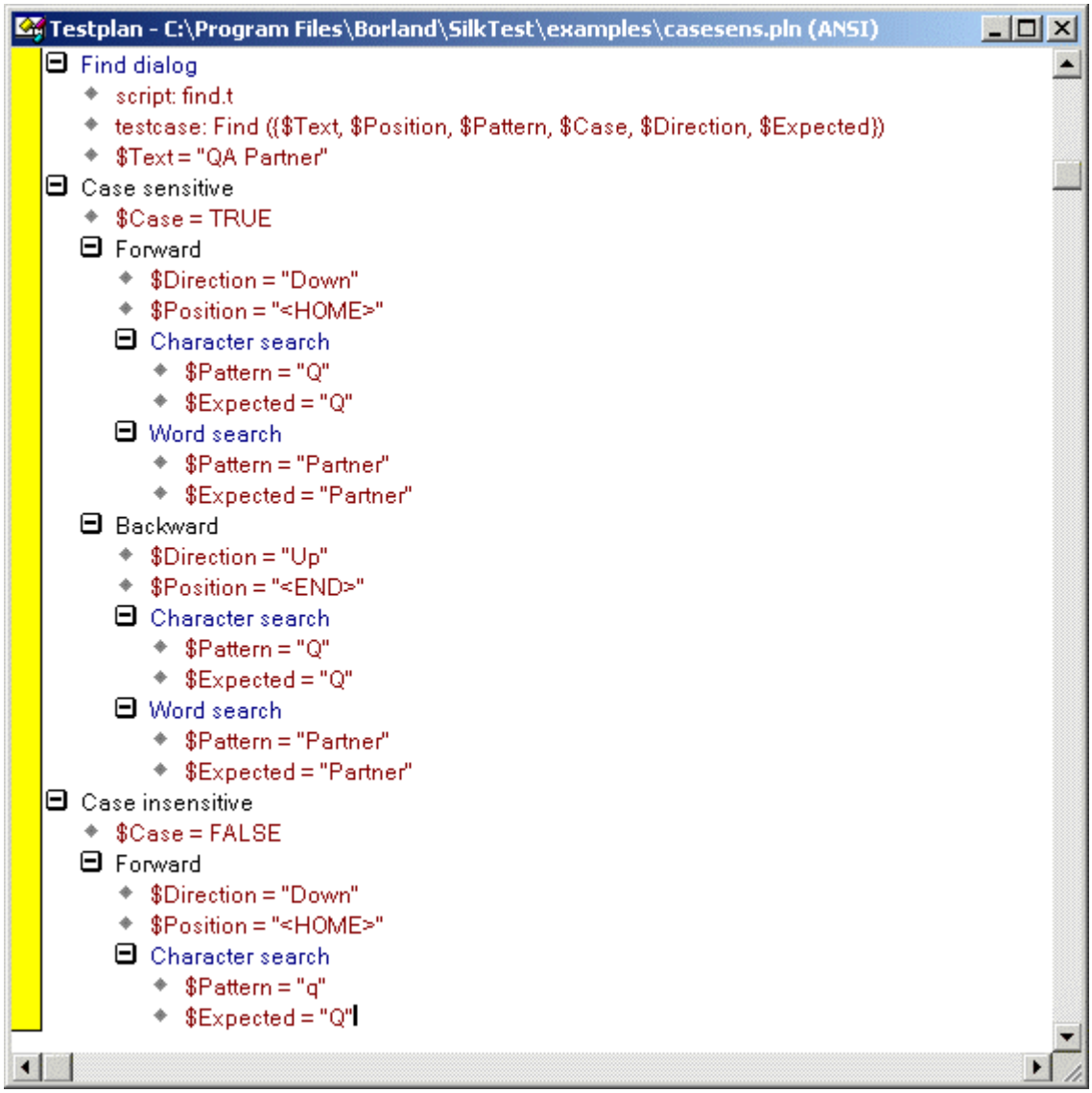
Using Symbols

This section describes symbols, which represent pieces of data in a data driven test case.

Overview of Symbols

A symbol represents a piece of data in a data driven test case. It is like a 4Test identifier, except that its name begins with the $ character. The value of a symbol can be assigned locally or inherited. Locally assigned symbols display in black and symbols that inherit their value display in blue in the **Testplan Detail** dialog box.

For example, consider the following test plan:



The test plan in the figure uses six symbols:

- `$Text` is the text to enter in the document window.
- `$Position` is the position of the insertion point in the document window.

- `$Pattern` is the pattern to search for in the document window.
- `$Case` is the state of the **Case Sensitive** check box.
- `$Direction` is the direction of the search.
- `$Expected` is the expected match.

The symbols are named in the parameter list to the `FindTest testcase`, within the parentheses after the test case name.

```
testcase: FindTest ({ $Text, $Position, $Pattern, $Case, $Direction,
$Expected })
```

- The symbols are only named in the parameter list; they are not assigned values. The values are assigned at either the group or test description level, depending on whether the values are shared by several tests or are unique to a single test. If a symbol is defined at a level in the plan where it can be shared by a group of tests, each test can assign its own local value to the symbol, overriding whatever value it had at the higher level. You can tell whether a symbol is locally assigned by using the **Testplan Detail** dialog box: Locally assigned symbols display in black. Symbols that inherit their values display in blue.

For example, in the preceding figure, each test description assigns its own unique values to the $Pattern and the $Expected symbols. The remaining four symbols are assigned values at a group description level:

- The `$Text` symbol is assigned its value at the Find dialog group description level, because all eight tests of the Find dialog enter the text Silk Test Classic into the document window of the Text Editor application.
- The `$Case` symbol is assigned the value TRUE at the Case sensitive group description level and the value FALSE at the Case insensitive group description level.
- The `$Direction` symbol is assigned the value Down at the Forward group description level, and the value Up at the Backward group description level.
- The `$Position` symbol is assigned the value <HOME> at the Forward group description level, and the value <END> at the Backward group description level.

Because the data that is common is factored out and defined at a higher level, it is easy to see exactly what is unique to each test.

Symbol Definition Statements in the Test Plan Editor

Use symbols to define data that is shared by a group of tests in the plan. Symbol definitions follow these syntax conventions:

- The symbol name can be any valid 4Test identifier name, but must begin with the $ character.
- The symbol value can be any text. When the test plan editor encounters the symbol, it expands it (in the same sense that another language expands macros). For example, the following test plan editor statement defines a symbol named Color and assigns it the `STRING` value "Red":

  ```
  $Color = "Red"
  ```
- To use a $ in a symbol value, precede it with another $. Otherwise, the compiler will interpret everything after the $ as another symbol. For example, this statement defines a symbol with the value Some $String: `$MySymbol = "Some$$String "`
- To assign a null value to a symbol, do not specify a value after the equals sign. For example: `$MyNullSymbol =`
- To indicate that a test is incomplete when generating a test plan completion report, assign the symbol the ? character. For example: `$MySymbol = ?`

If a symbol is listed in the argument list of a test case, but is not assigned a value before the test case is actually called, the test plan editor generates a runtime error that indicates that the symbol is undefined. To avoid this error, assign the symbol a value or a ? if the data is not yet finalized.

Defining Symbols in the Testplan Detail Dialog box

Place the insertion cursor in the plan where you need to assign a value to a symbol.

1. Click **Testplan** > **Detail**.
2. Select the **Symbols** tab on the **Testplan Detail** dialog box, and enter the symbol definition in the text box to the left of the **Add** button.

   You do not need to enter the $ character; the test plan editor takes care of this for you when it inserts the definitions into the test plan.
3. Click **Add**. Silk Test Classic adds the symbol to the list box above the **Add text** text box.
4. Define additional symbols in the same manner, and then click **OK** when finished.

   Silk Test Classic closes the **Testplan Detail** dialog box and enters the symbol definitions, including the $ character, into the plan. If a symbol is defined at a level in the plan where it can be shared by a group of tests, each test can assign its own local value to the symbol, overriding whatever value it had at the higher level. You can tell whether a symbol is locally assigned by using the **Testplan Detail** dialog box: Locally assigned symbols display in black. Symbols that inherit their values display in blue.

Assigning a Value to a Symbol

You can define symbols and assign values to them by typing them into the test plan, using this syntax:

```
$symbolname = symbolvalue
```

where `symbolname` is any valid 4Test identifier name, prefixed with the $ character and `symbolvalue` is any string, list, array, or the ? character (which indicates an undefined value).

For example, the following statement defines a symbol named Color and assigns it the `STRING` value "Red":

```
$Color = "Red"
```

If a symbol is defined at a level in the plan where it can be shared by a group of tests, each test can assign its own local value to the symbol, overriding whatever value it had at the higher level.

Specifying Symbols as Arguments when Entering a testcase Statement

1. Place the insertion cursor in the test plan at the location where the `testcase` statement is to be inserted. Placing a symbol name in the argument list of a `testcase` statement only specifies the name of the symbol; you also need to define the symbol and assign it a value at either the group or test case description level, as appropriate.

   If you do not know the value when you are initially writing the test plan, assign a question mark (?) to avoid getting a compiler error when you compile the test plan; doing so will also cause the tests to be counted as incomplete when a **Completion report** is generated.
2. Click **Testplan** > **Detail**.
3. Enter the name of a data driven test case on the **Testplan Detail** dialog box, followed by the argument list enclosed in parenthesis. If the test case expects a record, and not individual values, you must use the list constructor operator (curly braces).
4. Click **OK**. Silk Test Classic dismisses the **Testplan Detail** dialog box and inserts the `testcase` statement into the test plan.

Attributes and Values

This section describes site-specific characteristics that you can define for your test plan and assign to test descriptions and group descriptions.

Overview of Attributes and Values

Attributes are site-specific characteristics that you can define for your test plan and assign to test descriptions and group descriptions. Attributes are used to categorize tests, so that you can reference them as a group. Attributes can also be incorporated into queries, which allow you to mark tests that match the query's criteria. Marked tests can be run as a group.

By assigning attributes to parts of the test plan, you can:

- Group tests in the plan to distinguish them from the whole test plan.
- Report on the test plan based on a given attribute value.
- Run parts of the test plan that have a given attribute value.

For example, you might define an attribute called Engineer that represents the set of QA engineers that are testing an application through a given test plan. You might then define values for Engineer like David, Jesse, Craig, and Zoe, the individual engineers who are testing this plan. You can then assign the values of Engineer to the tests in the test plan. Certain tests are assigned the value of David, others the value of Craig, and so on. You can then run a query to mark the tests that have a given value for the Engineer attribute. Finally, you can run just these marked tests.

Attributes are also used to generate reports. For example, to generate a report on the number of passed and failed tests for Engineer Craig, simply select this value from the **Pass/Fail Report** dialog box. You do not need to mark the tests or build a query in this case.

Attributes and values, as well as queries, are stored by default in `testplan.ini` which is located in the Silk Test Classic installation directory. The initialization file is specified in the **Data File for Attributes and Queries** field in the **General Options** dialog box.

Silk Test Classic ships with predefined attributes. You can also create up to 254 user-defined attributes.

Make sure that all the QA engineers in your group use the same initialization body file. You can modify the definition of an attribute.

Modifying attributes and values through the **Define Attributes** dialog box has no effect on existing attributes and values already assigned to the test plan. You must make the changes in the test plan yourself.

Predefined Attributes

The test plan editor has three predefined attributes:

Developer	Specifies the group of QA engineers who developed the test cases called by the test plan.
Component	Specifies the application modules to be tested in this test plan.
Category	Specifies the kinds of tests used in your QA Department, for example, Smoke Test.

User Defined Attributes

You can define up to 254 attributes. You can also rename the predefined attributes.

The rules for naming attributes include:

- Attribute names can be up to 11 characters long.
- Attribute and value names are not case sensitive.

Adding or Removing Members of a Set Attribute

Tests can be assigned more than one value at a time for attributes whose type is `Set`.

For example, you might have a `Set` variable called *RunWhen* with three values: `UI`, `regression`, and`smoke`. You can assign any combination of these three values to a test or group of tests. Separate each value with a semicolon.

You can use the + or – operator to add or subtract elements to what were previously assigned.

Consider the following examples:

Using + to add numbers

```
RunWhen: UI; regression Test 1    testcase: t1 RunWhen: + smoke
Test 2    testcase: t2
```

In this example, Test 1 has the values `UI` and `regression` . The statement

```
RunWhen: + smoke
```

adds the value `smoke` to the previously assigned values, so Test 2 has the values `UI`, `regression`, and `smoke`.

Using - to remove numbers

```
RunWhen: UI; regression Test 1 testcase: t1 RunWhen: -
regression Test 2
```

testcase: t2

In this example, Test 1 has the values `UI` and `regression`. The statement

```
RunWhen: - regression
```

removes the value `regression` from the previously assigned values, so Test2 has the value `UI`.

Rules for Using + and -

- You must follow the + or – with a space.
- You can add or remove any number of elements with one statement. Separate each element with a semicolon.
- You can specify + elements even if no assignments had previously been made. The result is that the elements are now assigned.
- You can specify – elements even if no assignments had previously been made. The result is that the set's complement is assigned. Using the previous example, specifying:

  ```
  RunWhen: - regression
  ```

  when no `RunWhen` assignment had previously been made results in the values `UI` and `smoke` being assigned.

Defining an Attribute and its Values

1. Click **Testplan** > **Define Attributes**, and then click **New**.

2. Name the attribute.
3. Select one of the following types, and then click **OK**.

   Normal You specify values when you define the attribute. Users of the attribute in a test plan pick one value from the list.

   Edit You don't specify values when you define the attribute. Users type their own values when they use the attribute in a test plan.

   Set Like normal, except that users can pick more than one value.
4. On the **Define Attributes** dialog box, if you:
   - have defined an `Edit` type attribute, you are done. Click **OK** to close the dialog box.
   - are defining a `Normal` or `Set` type attribute, type a value in the text box and click **Add**.

Once attributes have been defined, you can modify them.

Assigning Attributes and Values to a Test Plan

Attributes and values have no connection to a test plan until you assign them to one or more tests using an assignment statement. To add an assignment statement, you can do one of the following:

- Type the assignment statement yourself directly in the test plan.
- Use the **Testplan Detail** dialog box.

Format

An assignment statement consists of the attribute name, a colon, and a valid attribute value, in this format:

```
attribute-name: attribute value
```

For example, the assignment statement that associates the `Searching` value of the *Module* attribute to a given test would look like:

```
Module: Searching
```

Attributes of type `Set` are represented in this format:

```
attribute-name: attribute value; attribute
value; attribute value; ...
```

Placement

Whether you type an assignment statement yourself or have the **Testplan Detail** dialog box enter it for you, the position of the statement in the plan is important.

To have an assignment statement apply to	Place it directly after the
An individual test	test description
A group of tests	group description

Assigning an Attribute from the Testplan Detail Dialog Box

1. Place the cursor in the test plan where you would like the assignment statement to display, either after the test description or the group description.
2. Click **Testplan** > **Detail**, and then click the **Test Attributes** tab on the **Testplan Detail** dialog box. The arrow in the list box at the top of the dialog box identifies the test description at the cursor position in the test plan. The attribute will be added to this test description. The **Test Attributes** tab lists all your current attributes at this level of the test plan.

3. Do one of the following:
   - If the attribute is of type *Normal*, select a value from the list.
   - If the attribute is of type *Set*, select on or more values from the list.
   - If the attribute is of type *Edit*, type a value.
4. Click **OK**. Silk Test Classic closes the dialog box and places the assignment statements in the test plan.

Modifying the Definition of an Attribute

Be aware that modifying attributes and values through the **Define Attributes** dialog box has no effect on existing attributes and values already assigned to the test plan. You must make the changes in the test plan yourself.

1. Click **Testplan** > **Define Attributes**.
2. On the **Define Attributes** dialog box, select the attribute you want to modify, then:

Rename an attribute	Edit the name in the **Name** text box.
Assign a new value to the attribute	Type the value in the text box at the bottom right of the dialog box, and click **Add**. The value is added to the list of values.
Modify a value	Select the value from the **Values** list box, and click **Edit**. The value displays in the text box at the bottom right of the dialog box and the **Add** button is renamed to **Replace**. Modify the value and click **Replace**.
Delete a value	Select the value from the **Values** list box and click **Remove**. The text box is cleared and the value is removed from the **Values** list box.
Delete an attribute	Click **Delete**.

3. Click **OK**. The attributes and values are saved in the initialization file specified in the **General Options** dialog box.

Queries

This section describes how you can use a test plan query to mark all tests that match a user-selected set of criteria, or test characteristics.

Overview of Test Plan Queries

You can use a test plan query to mark all tests that match a user-selected set of criteria, or test characteristics. A query comprises one or more of the following criteria:

- Test plan execution: script file, test case name, or test data
- Test attributes and values
- Symbols and values

Test attributes and symbols must have been previously defined to be used in a query.

Named queries are stored by default in `testplan.ini`. The initialization file is specified in the **Data File for Attributes and Queries** text box in the **General Options** dialog box. The `testplan.ini` file is in the Silk Test Classic installation directory. Make sure that all the QA engineers in your group use the same initialization file.

Overview of Combining Queries to Create a New Query

You can combine two or more existing queries into a new query using the **Mark by Named Query** dialog box. The new query can represent the union of the constituent queries (logical OR) or the intersection of the constituent queries (logical AND).

Combining by union

Combining two or more queries by union creates a new named query that marks all tests that would have been marked by running each query one after the other while retaining existing marks. Since **Mark by Named Query** clears existing marks before running a query, the only way to achieve this result is to create a new query that combines the constituent queries by union.

Example

Suppose you have two queries, Query1 and Query2, that you want to combine by union.

Query1	Query2
Developer: David	Developer: Jesse
Component: Searching	TestLevel: 2

The new query created from the union of Query1 and Query2 will first mark those tests that match all the criteria in Query1 (Developer is David and Component is Searching) and then mark those tests that match all the criteria in Query2 (Developer is Jesse and TestLevel is 2).

Combining by intersection

Combining two or more queries by intersection creates a new named query that marks every test that has the criteria specified in all constituent queries.

Example

For example, combining Query1 and Query2 by intersection would create a new query that comprised these criteria: Developer is David and Jesse, Component is Searching, and TestLevel is 2. In this case, the new query would not mark any tests, since it is impossible for a test to have two different values for the attribute Developer (unless Developer were defined as type Set under Windows). Use care when combining queries by intersection.

Guidelines for Including Symbols in a Query

- Use ? (question mark) to indicate an unset value. For example, `Mysymbol = ?` in a query would mark those tests where *Mysymbol* is unset. Space around the equals sign (=) is insignificant.
- If you need to modify the symbol in the query, select it from the list box and click **Edit**. The test plan editor places it in the text box and changes the **Add** button to **Replace**. Edit the symbol or value and click **Replace**.
- To exclude the symbol from the query, select it from the list box and click **Remove**. The test plan editor deletes it from the list box.

The Differences between Query and Named Query Commands

Testplan > **Mark by Query** or **Testplan** > **Mark by Named Query** both create queries, however, **Mark by Named Query** provides extra features, like the ability to combine queries or to create a query without running it immediately. If the query-creation function and the query-running function are distinct in your company, then use **Mark by Named Query**. If you intend to run a query only once, or run a query while keeping existing marks, then use **Mark by Query**.

The following table highlights the differences between the two commands.

Mary by Query	Mark by Named Query
Builds a query based on criteria you select and runs query immediately.	Builds a new query based on criteria you select. Can run query at any time.
Name is optional, but note that only named queries are saved and can be rerun at any time in the **Mark by Named Query** dialog box.	Name is required. Query is saved.
Cannot edit or delete a query.	Can edit or delete a query.
Cannot combine queries.	Can combine queries into a new query.
Lets you decide whether or not to clear existing marks before running new query. Unmarks by default.	Clears existing marks before running new query.

Unnamed queries can be run only once. If you name the query, you can have the test plan editor run it in the same or subsequent work sessions without having to rebuild the query or manually remark the tests that you're interested in rerunning or reporting on.

Create a New Query

You can create a new query through either **Testplan** > **Mark by Query** or **Testplan** > **Mark by Named Query**. You can also create a new query by combining existing queries.

1. Open the test plan and any associated sub-plans.
2. Click **Testplan** > **Mark by Query** or **Testplan** > **Mark by Named Query**.
3. Identify the criteria you want to include in the query. To include:
   - A script, test case, or test data, use the **Test Execution** tab. Use the **Script** and **Testcase** buttons to select a script and test case, or type the full specification yourself. To build a query that marks only manual tests, enter the keyword manual in the **Testcase** text box.
   - Existing attributes and values in the query, use the **Test Attributes** tab.
   - One or more existing symbols and values, use the **Symbols** panel. Type the information and click **Add**. The symbol and value are added to the list box.

   Do not type the dollar sign ($) prefix before the symbol name. The wildcard characters * (asterisk) and ? (question mark) are supported for partial matches: * is a placeholder for 0 or more characters, and ? is a placeholder for 1 character.

   Example 1

   If you type `find_5 (*` in the **Testcase** field, the query searches all the `testcase` statements in the plan and marks those test descriptions that match, as well as all subordinate descriptions to which the matching `testcase` statement applies (those where the find_5 testcase passed in data).

   Example 2

   If you type `find.t` in the Script field, the query searches all `script` statements in the plan and marks those test descriptions that match exactly, as well as all subordinate descriptions to which the matching

`script` statement applies (those in which you had specified find.t exactly). It would not match any `script` statements in which you had specified a full path.

4. Take one of the following actions, depending on the command you chose to create the query:

Mark by Query	Click **Mark** to run the query against the test plan. The test plan editor closes the dialog box and marks the test plan, retaining the existing marks if requested.
Mark by Named Query	Click **OK** to create the query. The **New Testplan Query** dialog box closes, and the **Mark by Named Query** dialog box is once again visible. The new query displays in the **Testplan Queries** list box. If you want to: • Run the query, select it from the list box and click **Mark**. • Close the dialog box without running the query, click **Close**.

Edit a Query

1. Click **Testplan** > **Mark by Named Query** to display the **Mark by Named Query** dialog box.
2. Select a query from the **Testplan Queries** list box and click **Edit**.
3. On the **Edit Testplan Query** dialog box, edit the information as appropriate, and then click **OK** .
4. To run the query you just edited, select the query and click **Mark** . To close the dialog box without running the edited query, click **Close** .

Delete a Query

1. Click **Testplan** > **Mark by Named Query** to open the **Mark by Named Query** dialog box.
2. Select a query from the **Testplan Queries** box and click **Remove**.
3. Click **Yes** to delete the query, and then click **Close** to close the dialog box.

Combining Queries

1. Click **Testplan** > **Mark by Named Query** to display the **Mark by Named Query** dialog box.
2. Click **Combine**. The **Combine Testplan Queries** dialog box lists all existing named queries in the **Queries to Combine** list box.
3. Specify a name for the new query in the **Query Name** text box.
4. Select two or more queries to combine from the **Queries to Combine** list box.
5. Click the option button that represents the combination method to use: either **Union of Queries** or **Intersection of Queries**.
6. Click **OK** to save the new query. The **Mark by Named Query** dialog box displays with the new query in the **Testplan Queries** list box.
7. To run the query, select the query and click **Mark** or click **Close** to close the dialog box without running the query.

Designing and Recording Test Cases with the Classic Agent

This section describes how you can design and record test cases with the Classic Agent.

Hierarchical Object Recognition

When you record window declarations, Silk Test Classic records descriptions based on hierarchical object recognition of the GUI objects in your application. Silk Test Classic stores the declarations in an include file (*.inc). When you record or replay a test case, Silk Test Classic references the declarations in the include file to identify the objects named in your test scripts.

Using hierarchical object recognition compared to using dynamic object recognition

Use hierarchical object recognition to test applications that require the Classic Agent. Dynamic object recognition requires the Open Agent.

Alternatively, you can combine the advantages of INC files with the advantages of dynamic object recognition by including locator keywords in INC files. Enhancing INC files with locators facilitates a smooth transition from using hierarchical object recognition to new scripts that use dynamic object recognition. With locators, you use dynamic object recognition but your scripts look and feel like traditional, Silk Test Classic tag-based scripts that use hierarchical object recognition.

You can create tests for both dynamic and hierarchical object recognition in your test environment. You can use both recognition methods within a single test case if necessary. Use the method best suited to meet your test requirements.

Open Agent Example

For example, if you record a test to open the **New Window** dialog box by clicking **File** > **New** > **Window** in the SWT sample application, Silk Test Classic performs the following tasks:

- Records the following test:

```
testcase Test1 ()
  recording
    SwtTestApplication.WindowMenuItem.Pick()
```

- Creates window declarations in the include file for Window menu item. For example:

```
window Shell SwtTestApplication
  locator "/Shell[@caption='Swt Test Application']"
  MenuItem WindowMenuItem
    locator "//MenuItem[@caption='Window']"
```

Classic Agent Example

For example, if you record a test to open the **New Window** dialog box by clicking **File** > **New** > **Window** in a sample application, Silk Test Classic performs the following tasks:

- Records the following test:

```
testcase Test1 ()
  recording
    SwtTestApplication.File.New.xWindow.Pick()
```

- Creates window declarations in the include file for File menu, New menu item, and xWindow menu item. For example:

```
Menu File
  tag "File"
  MenuItem New
    tag "New.."
    MenuItem xWindow
      tag "Window"
```

Highlighting Objects During Recording

During recording, the active object in the AUT is highlighted by a green rectangle. As soon as a new object becomes active this new object is highlighted. If the same object remains active for more than 0.5 seconds a tool-tip will be displayed that displays the class name of the active object and also the current position of the mouse relative to the active object. This tool-tip will no longer be displayed when a new object becomes active, the user presses the mouse, or automatically after 2 seconds.

Setting Recording Preferences for the Classic Agent

Specify settings that Silk Test Classic uses during recording.

All the following settings are optional. Change these settings if they will improve the quality of your test methods.

1. Click **Options** > **Recorder**. The **Recorder Options** dialog box opens.
2. To set **Ctrl+Shift** as the shortcut key combination to use to pause recording, check the **Change hotkey to Ctrl+Shift** check box.

   By default, **Ctrl+Alt** is the shortcut key combination.
3. To record the tags that are specified in the **Record Window Declarations Options** dialog box, check the **Record multiple tags** check box.
4. To add new declarations to the INC file during recording, check the **Auto Declaration** check box.
5. To verify the test application using properties instead of attributes, check the **Verify using properties** check box.

   This option is checked automatically if you have enabled enhanced support for Visual Basic. This feature requires properties for verification. You cannot uncheck the **Verify using properties** check box without disabling enhanced support for Visual Basic.
6. To record events at a lower level for selected controls, check the corresponding check boxes in the **Recorded Events** list.

   For example, you might want to record a click in a check box, instead of recording an actual selection. If you specify that you want to record only low-level events in check boxes, Silk Test Classic records something like the following when you select a check box: `Find.CaseSensitive.Click (1, 41, 10)`. If you are using a high-level event, Silk Test Classic records something like the following: `Find.CaseSensitive.Check ()`.
7. To record absolute values for scroll events, check the **OPT_RECORD_SCROLLBAR_ABSOLUT** check box.
8. button is pressed. Typically, you leave this checked unless you are testing an application, such as a drawing application, where mouse movements themselves are significant. Default is checked.
9. To record mouse movements that cannot be built into higher-level actions and that occur while a mouse button is pressed when you select the **Record Testcase** and **Record Actions** commands, uncheck the Ignore mouse move events check box.

Leave the check box checked unless you are testing an application where mouse movements themselves are significant.

10. To record `BeginDrag` and `EndDrag` methods when you press a mouse button on an object and do a drag operation on a listview, treeview, or list box, uncheck the **Don't record BeginDrag/EndDrag** check box.
11. Click **OK**.

Test Cases

This section describes how you can use automated tests to address single objectives of a test plan.

Overview of Test Cases

A test case is an automated test that addresses one objective of a test plan. A test case:

- Drives the application from the initial state to the state you want to test.
- Verifies that the actual state matches the expected (correct) state. Your QA department might use the term baseline to refer to this expected state. This stage is the heart of the test case.
- Cleans up the application, in preparation for the next test case, by undoing the steps performed in the first stage.

In order for a test case to function properly, the application must be in a stable state when the test case begins to execute. This stable state is called the base state. The recovery system is responsible for maintaining the base state in the event the application fails or crashes, either during the execution of a test cases or between test cases.

Each test case is independent and should perform its own setup, driving the application to the state that you want to test, executing the test case, and then returning the application to the base state. The test case should not rely on the successful or unsuccessful completion of another test case, and the order in which the test case is executed should have no bearing on its outcome. If a test case relies on a prior test case to perform some setup actions, and an error causes the setup to fail or, worse yet, the application to crash, all subsequent test cases will fail because they cannot achieve the state where the test is designed to begin.

A test case has a single purpose: a single test case should verify a single aspect of the application. When a test case designed in this manner passes or fails, it is easy to determine specifically what aspect of the target application is either working or not working.

If a test case contains more than one objective, many outcomes are possible. Therefore, an exception may not point specifically to a single failure in the software under test but rather to several related function points. This makes debugging more difficult and time-consuming and leads to confusion in interpreting and quantifying results. The result is an overall lack of confidence in any statistics that might be generated. But there are techniques you can use to perform more than one verification in a test case.

Types of test cases

Silk Test Classic supports two types of test cases, depending on the type of application that you are testing. You can create test cases that use:

Hierarchical object recognition	This is a fast, easy method for creating scripts. This type of testing is supported for all application types.
Dynamic object recognition	This is a more robust and easy to maintain method for creating scripts. However, dynamic object recognition is only supported for applications that use the Open Agent.

If you are using the Open Agent, you can create tests for both dynamic and hierarchical object recognition in your test environment. Use the method best suited to meet your test requirements. You can use both recognition methods within a single test case if necessary.

Anatomy of a Basic Test Case

A test case is comprised of testcase keywords and object-oriented commands. You place a group of test cases for an application into a file called a script.

Each automated test for an application begins with the testcase keyword, followed by the name of the test case. The test case name should indicate the type of testing being performed.

The core of the test case is object-oriented 4Test commands that drive, verify, and clean up your application. For example, consider this command:

```
TextEditor.File.New.Pick
```

The first part of the command, `TextEditor.File.New`, is the name of a GUI object. The last part of the command, `Pick`, is the operation to perform on the GUI object. The dot operator (`.`) delimits each piece of the command. When this command is executed at runtime, it picks the **New** menu item from the **File** menu of the Text Editor application.

Types of Test Cases

There are two basic types of test cases:

- Level 1 tests, often called smoke tests or object tests, verify that an application's GUI objects function properly. For example, they verify that text boxes can accept keystrokes and check boxes can display a check mark.
- Level 2 tests verify an application feature. For example, they verify that an application's searching capability can correctly find different types of search patterns.

You typically run Level 1 tests when you receive a new build of your application, and do not run Level 2 tests until your Level 1 tests achieve a specific pass/fail ratio. The reason for this is that unless your application's graphical user interface works, you cannot actually test the application itself.

Test Case Design

When defining test requirements, the goal is to vigorously test each application feature. To do so, you need to decide which set of inputs to a feature will provide the most meaningful test results.

As you design your test cases, you may want to associate data with individual objects, which can then be referenced inside test cases. You may find this preferable to declaring global variables or passing parameters to your test cases.

The type of data you decide to define within a window declaration will vary, depending on the type of testing you are doing. Some examples include:

- The default value that you expect the object to have when it displays.
- The tab sequence for each of a dialog box's child objects.

The following declaration for the **Find** dialog contains a list that specifies the tab sequence of the dialog box children.

```
window DialogBox Find
   tag "Find"
   parent TextEditor
   LIST OF WINDOW lwTabOrder = {...}
      FindWhat
      CaseSensitive
      Direction
      Cancel
```

For more information about the syntax to use for lists, see *LIST data type*.

Before you begin to design and record test cases, make sure that the built-in recovery system can close representative dialogs from your application window.

Constructing a Test Case

This section explains the methodology you use when you design and record a test case.

A test case has three stages

Each test case that you record should have the following stages:

Stage 1 The test case drives the application from the initial state to the state you want to test.

Stage 2 The test case verifies that the actual state matches the expected (correct) state. Your QA department might use the term baseline to refer to this expected state. This stage is the heart of the test case.

Stage 3 The test case cleans up the application, in preparation for the next test case, by undoing the steps performed in stage 1.

Each test case is independent

Each test case you record should perform its own setup in stage 1, and should undo this setup in stage 3, so that the test case can be executed independently of every other test case. In other words, the test case should not rely on the successful or unsuccessful completion of another test case, and the order in which it is executed should have no bearing on its outcome.

If a test case relies on a prior test case to perform some setup actions, and an error causes the setup to fail or, worse yet, the application to crash, all subsequent test cases will fail because they cannot achieve the state where the test is designed to begin.

A test case has a single purpose

Each test case you record should verify a single aspect of the application in stage 2. When a test case designed in this manner passes or fails, it's easy to determine specifically what aspect of the target application is either working or not working.

If a test case contains more than one objective, many outcomes are possible. Therefore, an exception may not point specifically to a single failure in the software under test but rather to several related function points. This makes debugging more difficult and time-consuming and leads to confusion in interpreting and quantifying results. The net result is an overall lack of confidence in any statistics that might be generated.

There are techniques you can use to do more than one verification in a test case.

A test case starts from a base state

In order for a test case to be able to function properly, the application must be in a stable state when the test case begins to execute. This stable state is called the base state. The recovery system is responsible for maintaining the base state in the event the application fails or crashes, either during a test case's execution or between test cases.

DefaultBaseState

To restore the application to the base state, the recovery system contains a routine called `DefaultBaseState` that makes sure that:

- The application is running and is not minimized.
- All other windows, for example dialog boxes, are closed.

- The main window of the application is active.

If these conditions are not sufficient for your application, you can customize the recovery system.

Defining test requirements

When defining test requirements, the goal is to rigorously test each application feature. To do so, you need to decide which set of inputs to a feature will provide the most meaningful test results.

Data in Test Cases

What data does the feature expect

A user can enter three pieces of information in the **Find** dialog box:

- The search can be case sensitive or insensitive, depending on whether the **Case Sensitive** check box is checked or unchecked.
- The search can be forward or backward, depending on whether the **Down** or **Up** option button is selected.
- The search can be for any combination of characters, depending on the value entered in the **Find What** text box.

Create meaningful data combinations

To organize this information, it is helpful to construct a table that lists the possible combinations of inputs. From this list, you can then decide which combinations are meaningful and should be tested. A partial table for the **Find** dialog box is shown below:

Case Sensitive	Direction	Search String
Yes	Down	Character
Yes	Down	Partial word (start)
Yes	Down	Partial word (end)
Yes	Down	Word
Yes	Down	Group of words
Yes	Up	Character
Yes	Up	Partial word (start)
Yes	Up	Partial word (end)
Yes	Up	Word
Yes	Up	Group of words

Saving Test Cases

When saving a test case, Silk Test Classic does the following:

- Saves a source file, giving it the `.t` extension; the source file is an ASCII text file, which you can edit.
- Saves an object file, giving it the `.to` extension; the object file is a binary file that is executable, but not readable by you.

For example, if you name a test case (script file) `mytests` and save it, you will end up with two files: the source file `mytests.t`, in the location you specify, and the object file `mytests.to`.

To save a new version of a script's object file when the script file is in view-only mode, click **File** > **Save Object File**.

Recording Without Window Declarations

If you record a test case against a GUI object for which there is no declaration or if you want to write a test case from scratch against such an object, Silk Test Classic requires a special syntax to uniquely identify the GUI object because there is no identifier.

This special syntax is called a dynamic instantiation and is composed of the class and tag of the object. The general syntax of this kind of identifier is:

```
class("tag").class("tag"). ...
```

Example

If there is not a declaration for the **Find** dialog box of the **Notepad** application, the syntax required to identify the object with the Classic Agent looks like the following:

```
MainWin("Untitled - Notepad|$C:\Windows
\SysWOW64\notepad.exe").DialogBox("Find")
```

To create the dynamic tag, the recorder uses the multiple-tag settings that are stored in the **Record Window Declarations** dialog box. In the example shown above, the tag for the **Notepad** contains its caption as well as its window ID.

For the Open Agent, the syntax for the same example looks like the following:

```
FindMainWin("/MainWin[@caption='Untitled -
Notepad']").FindDialogBox("Find")
```

Overview of Application States

When testing an application, typically, you have a number of test cases that have identical setup steps. Rather than record the same steps over and over again, you can record the steps as an application state and then associate the application state with the relevant test cases.

An application state is the state you want your application to be in after the base state is restored but before you run one or more test cases. By creating an application state, you are creating reusable code that saves space and time. Furthermore, if you need to modify the Setup stage, you can change it once, in the application state routine.

At most, a test case can have one application state associated with it. However, that application state may itself be based on another previously defined application state. For example, assume that:

- The test case Find is associated with the application state Setup.
- The application state Setup is based on the application state OpenFile.
- The application state OpenFile is based on the built-in application state, DefaultBaseState.
- Silk Test Classic would execute the programs in this order:

  1. DefaultBaseState application state.
  2. OpenFile application state.
  3. Setup application state.
  4. Find test case.

If a test case is based on a single application state, that application state must itself be based on DefaultBaseState in order for the test case to use the recovery system. Similarly, if a test case is based on a chain of application states, the final link in the chain must be DefaultBaseState. In this way, the built-in recovery system of Silk Test Classic is still able to restore the application to its base state when necessary.

Behavior of an Application State Based on NONE

If an application state is based on the keyword NONE, Silk Test Classic executes the application state twice: when the test case with which it is associated is entered and when the test case is exited.

On the other hand, if an application state is based on DefaultBaseState, Silk Test Classic executes the application state only when the associated test case is entered.

The following example code defines the application state InvokeFind as based on the NONE keyword and associates that application state with the test case TestFind.

```
Appstate InvokeFind () basedon none
 xFind.Invoke ()
 print ("hello")

testcase TestFind () appstate InvokeFind
 print ("In TestFind")
 xFind.Exit.Click ()
```

When you run the test case in Silk Test Classic, in addition to opening the **Find** dialog box, closing it, and reopening it, the test case also prints:

```
hello
In TestFind
hello
```

The test case prints hello twice because Silk Test Classic executes the application state both as the test case is entered and as it is exited.

Example: A Feature of a Word Processor

For purposes of illustration, this topic develops test requirements for the searching feature of the sample Text Editor application using the **Find** dialog box. This topic contains the following:

- Determining what data the feature expects.
- Creating meaningful data combinations.
- Overview of recording the stages of a test case.

When a user enters the criteria for the search and clicks **Find Next**, the search feature attempts to locate the string. If the string is found, it is selected (highlighted). Otherwise, an informational message is displayed.

Determining what data the feature expects

A user can enter three pieces of information in the **Find** dialog box:

- The search can be case sensitive or insensitive, depending on whether the **Case Sensitive** check box is checked or unchecked.
- The search can be forward or backward, depending on whether the **Down** or **Up** option button is clicked.
- The search can be for any combination of characters, depending on the value entered in the **Find What** text box.

Creating meaningful data combinations

To organize this information, it is helpful to construct a table that lists the possible combinations of inputs. From this list, you can then decide which combinations are meaningful and should be tested. A partial table for the **Find** dialog box is shown below:

Case Sensitive	Direction	Search String
Yes	Down	Character
Yes	Down	Partial word (start)
Yes	Down	Partial word (end)
Yes	Down	Word
Yes	Down	Group of words
Yes	Up	Character
Yes	Up	Partial word (start)
Yes	Up	Partial word (end)
Yes	Up	Word
Yes	Up	Group of words

Overview of recording the stages of a test case

A test case performs the included actions in three stages. The following table illustrates these stages, describing in high-level terms the steps for each stage of a sample test case that tests whether the Find facility is working.

Setup

1. Open a new document.
2. Type text into the document.
3. Position the text cursor either before or after the text, depending on the direction of the search.
4. Select **Find** from the **Search** menu.
5. In the **Find** dialog box:
   - Enter the text to search for in the **Find What** text box.
   - Select a direction for the search.
   - Make the search case sensitive or not.
   - Click **Find Next** to do the search.
6. Click **Cancel** to close the **Find** dialog box.

Verify

Record a 4Test verification statement that checks that the actual search string found, if any, is the expected search string.

Cleanup

1. Close the document.
2. Click **No** when prompted to save the file.

After learning the basics of recording, you can record from within a test plan, which makes recording easier by automatically generating the links that connect the test plan to the test case.

Recording Test Cases with the Classic Agent

This section describes how you can record test cases with the Classic Agent.

Overview of Recording the Stages of a Test Case

A test case includes several stages. The following table illustrates these stages, describing in high-level terms the steps for each stage of a sample test case that tests whether the Find facility is working.

After learning the basics of recording, you can record from within the test plan file, which makes recording easier by automatically generating the links that connect the test plan to the test case.

Setup and Record

1. Open a new document.
2. Type text into the document.
3. Position the text cursor either before or after the text, depending on the direction of the search.
4. Click **Find** in the **Search** menu.
5. In the **Find** dialog box:
   a. Type the text to search for in the **Find What** text box.
   b. Select a direction for the search.
   c. Make the search case sensitive or not.
   d. Click **Find Next** to perform the search.
6. Click **Cancel** to close the **Find** dialog box.

Verify

Record a 4Test verification statement that checks that the actual search string found, if any, is the expected search string.

Cleanup

1. Close the document.
2. Click **No** when prompted to save the file.

Overview of Recording 4Test Components

This functionality is available only for projects or scripts that use the Classic Agent.

If you want to manually write some or most of your 4Test code, or if you want to add individual lines to an existing test case, you can use the following recording tools:

Record/Actions

For example, when you are working with a script, you might want to leave the **Record Actions** dialog box open. Any time you want to verify a GUI object, you can point to the object in your application and verify it.

You can also use the dialog box to write a syntactically correct 4Test statement based on your manual interaction with your application. This eliminates the need to search through the documentation for the correct method and its arguments. Once the statement is recorded, the **Paste to Editor** button inserts the statement to your script.

Record/Window Identifiers

Similar to the `Actions` command, **Record/Window Identifiers** records the fully qualified name of the GUI object you are pointing at, which you can then insert into your script. This eliminates the need to bring up your test frame file to find the correct identifier for the object.

Record/Window Locations

It can be useful to know the position of certain objects, for example objects that are drawn (like tools on a toolbar) or drawing regions (in a CAD/CAM package, for example). To record the location of an object, use the **Record Window Locations** dialog box. You can also add a window location to an existing window declaration.

Record/Class

If you are using ActiveX, Visual Basic, or Java classes (controls) that are not predefined, you can record the classes for use in your tests.

Recording a Test Case With the Classic Agent

When you record a test case with the Classic Agent, Silk Test Classic uses hierarchical object recognition, a fast, easy method to create scripts. However, test cases that use dynamic object recognition are more robust and easy to maintain. You can create tests for both dynamic and hierarchical object recognition in your test environment. Use the method best suited to meet your test requirements. You can use both recognition methods within a single test case if necessary.

1. Enable extensions and set up the recovery system.
2. Click **Record Testcase** on the Basic Workflow bar. If the workflow bar is not visible, click **Workflows** > **Basic** to enable it.
3. Type the name of your test case in the **Testcase name** text box of the **Record Testcase** dialog box.

   Test case names are not case sensitive; they can be any length and consist of any combination of alphabetic characters, numerals, and underscore characters.
4. Select **DefaultBaseState** in the **Application State** field to have the built-in recovery system restore the default BaseState before the test case begins executing. If you chose **DefaultBaseState** as the application state, the test case is recorded in the script file as: `testcase testcase_name ()`. If you chose another application state, the test case is recorded as: `testcase testcase_name () appstate appstate_name`.
5. If you do not want Silk Test Classic to display the status window it normally shows during playback when driving the application to the specified base state—perhaps because the status bar obscures a critical control in the application you are testing—uncheck the **Show AppState status window** check box.
6. Click **Start Recording**. Silk Test Classic:

   - Closes the **Record Testcase** dialog box.
   - Starts your application, if it was not already running.
   - Removes the editor window from the display.
   - Displays the **Record Status** window.
   - Waits for you to take further action.
7. Interact with your application, driving it to the state that you want to test.

   As you interact with your application, Silk Test Classic records your interactions in the **Testcase Code** field of the **Record Testcase** dialog box, which is not visible.
8. To review what you have recorded, click **Done** in the **Record Status** window. Silk Test Classic displays the **Record Testcase** dialog box, which contains the 4Test code that has been recorded for you.
9. To resume recording your interactions, click **Resume Recording** in the dialog box. To temporarily suspend recording, click **Pause Recording** on the **Record Status** window.
10. Verify the test case.

Verifying a Test Case

This functionality is available only for projects or scripts that use the Classic Agent.

The cornerstone of an automated test is the verification stage, in which the test verifies that the state of the application matches the expected (baseline) state. Using the recorder, you can record object-appropriate verification of your application's state, data, or appearance.

To record the verification stage:

1. Continue with these steps after you record a test case. Or, if you have previously recorded a verification statement in an existing test case, choose **Record** > **Actions** to modify it.
2. Drive your application to the state that you want to verify and position the mouse cursor over the object.
3. Look at the **Record Status** window and make sure it is listing the object you want to verify. If so, press `Ctrl+Alt`.

   The **Verify Window** dialog box opens over your application window. The **Window** field, in the top-left corner of the dialog box, displays the name of the object you were pointing at when you pressed `Ctrl+Alt`. If the name in the **Window** field is incorrect, click **Cancel** to close the dialog box and return to the application. Point to the object you want to verify and press `Ctrl+Alt` again.
4. If a script file is not the active window, Silk Test Classic prompts you for a file name. If prompted, specify the name of either a new or an existing script file and click **OK**.
5. Choose to verify any of the following:

   - Properties of an object.
   - Appearance using a bitmap.
   - An object's state using built-in verification methods or other methods in combination with the built-in `Verify` function.
6. If you are writing a complete test case, record the cleanup stage and paste the test case into the script. If you have added a verification statement to an existing test case, paste it into your script and close the **Record Actions** dialog box.

Recording the Cleanup Stage and Pasting the Recording

After performing the verification, continue to interact with your application. This is the cleanup stage. For example, in the sample test case, cleanup means closing the document window without saving it.

1. When you have finished recording your test case or just want to see what you have recorded, click **Done** on the **Record Status** window. Silk Test Classic displays the **Record Testcase** window again. The **Testcase Code** field contains your interactions written as 4Test code.
2. Review the code and take the following actions:

   - All the information in the window is complete and what you want, then click **Paste to Editor**. Silk Test Classic closes the **Record Testcase** dialog box and places the new test case in your script file. If the 4Test code is not what you expected, then edit the name in the **Testcase Name** field.
   - If the test case name is not what you want, then edit the name in the **Testcase Name** field.
   - If the application state is not the one you want, then delete the code in the **Testcase Code** field, select a new application state from the list box and click **Resume Recording** to re-record the test case.
   - If the test case is not finished, then click **Resume Recording**. The **Record Testcase** window is reopened. You can continue to record your interactions.

   Note: When you paste a recorded test case (or other recorded actions, such as when you use **Record Actions**) into a script, Silk Test Classic indents the code under a recording statement to facilitate playback. For more information, see *recording statement*.
3. Click **Paste to Editor**.

   If you have interacted with objects in your application that have not been identified in your include files, the **Update Files** dialog box opens. Choosing **Paste testcase only**, does not update any `.inc` files while it pastes to the script with dynamically instantiated new objects. **Update window declarations and testcase** will create window declarations for new objects and use the new identifiers in the resulting test case.

Note: If you edit the contents of the recorder window, then you must allow Silk Test Classic to update the window declarations. The **Paste testcase only** option will be disabled.

4. Click **File** > **Save** to save the script file.

Testing the Ability of the Recovery System to Close the Dialog Boxes of Your Application

Before you begin to design and record test cases, make sure that the built-in recovery system can close representative dialog boxes of your application. Although the recovery system is robust enough to be able to close almost any application window, some applications may have windows that close in an unconventional fashion.

Here are the three types of dialog boxes you should test:

- A modal dialog box, which is a dialog box that locks you out of the rest of your application until you dismiss it.
- A non-modal dialog box.
- A non-modal dialog box that causes the display of a confirmation dialog box.

To test the ability of the recovery system to close your the dialog boxes of your application:

1. Start Silk Test Classic.
2. If you have not already done so, record a test frame for your application.
3. Choose **Options** > **Runtime** to ensure that your application's test frame file is listed in the **Use Files** field in the **Runtime Options** dialog box.
4. Start your application and invoke a representative dialog box.
5. In Silk Test Classic, click **Run** > **Application State**.
6. On the **Run Application State** dialog box, select the **DefaultBaseState application state** and click **Run**.
7. Silk Test Classic executes the `DefaultBaseState` routine, which should close the dialog box and any open windows, then display a results file.

   If the built-in recovery system cannot close one of the three representative dialog boxes, you need to modify the recovery system so that it understands how to close the dialog box.

Linking to a Script and Test Case by Recording a Test Case

1. Place the cursor at the end of a test description or a group description.
2. Choose **Record** > **Testcase**. Silk Test Classic prompts you to name a script file to contain the test case. Silk Test Classic does not prompt you for a script file if there is a script defined at a higher level and inherited by the test case you are recording. If that script exists, Silk Test Classic puts the test case in that script.
3. If prompted, select an existing script from the list or enter the name of a new script in the **File Name** text box, then click **OK**.
4. On the **Record Testcase** dialog box, type the name for the test case and optionally select an application state to be run before the recording starts.
5. Click **Start Recording**. Silk Test Classic displays the **Recording Status** dialog box. The dialog box flashes the word `Recording` for the duration of the session.
6. When you are finished recording the actions that comprise the test case, click **Done** in the **Recording Status** dialog box.

7. On the **Record Testcase** dialog box, click **Paste to Editor**. Silk Test Classic closes the **Record Testcase** dialog box and inserts the test case into the script file. It also adds the script and test case statements to the test plan on a new line and indents them appropriately.

If the script file is inherited by the test case you are recording, only the testcase statement is pasted.

Saving a Script File

To save a script file, click **File** > **Save**. If it is a new file, Silk Test Classic prompts you for the file name and location.

If you are working within a project, Silk Test Classic prompts you to add the file to the project. Click **Yes** if you want to add the file to the open project, or **No** if you do not want to add this file to the project.

To save a new version of a script's object file when the script file is in view-only mode, choose **File** > **Save Object File**.

If you are working within a project, you can add the file to your project. If you add object files (`.to`, `.ino`) to your project, the files will display under the **Data** node on the **Files** tab. You cannot modify object files within the Silk Test Classic editor because object files are binary. To modify an object file, open the source file (`.t` or `.inc`), edit it, and then recompile.

Recording an Application State

You define an application state before recording the test cases associated with it. As with test cases, you can write an application state routine from scratch or you can use the **Application State** command on the **Record** menu.

1. Open the file in which you want to place the application state.

   This can either be the test frame file for the application or the script file where the associated test cases are defined.

   If you put the application state in the test frame file, it will be available to all test cases. If you put it in the script file, it will be available only to test cases in that script file.
2. Open the application that you want to test.
3. Choose **Record** > **Application State** to display the **Record Application State** dialog box.
4. Type the name of your new application state in the **Application State Name** text box.
5. Select an application state from the **Based On** list box.
6. Click **Start Recording**. Silk Test Classic closes the **Record Application State** dialog box and displays the **Record Status** window. The **Status** field flashes `Recording`.
7. Drive your application to the state you want to record. At any point, you can record a verification by pressing `Ctrl+Alt`.
8. When you have finished recording an application state, click **Done** on the **Record Status** window.

   Silk Test Classic redisplays the **Record Application State** dialog box. The **Application State Code** field contains the 4Test code you recorded. You can take the following actions:

All the information in the window is complete and what you expect.	Click **Paste to Editor**. Silk Test Classic closes the **Record Application State** dialog box and places the new application state in your file.
You want to alter the code.	Edit the **Application State Code** field.
The application state name is not what you want.	Edit the name in the **Application State Name** field.

The application state on which this application state is based is not the one you want.	Delete the code in the **Application State Code** field, select a new application state from the list, and click **Resume Recording** to re-record the application state.
The application state routine is not finished.	Click **Resume Recording**. Silk Test Classic opens the **Record Status** window.

Testing an Application State

Before you run a test case that is associated with an application state, make sure the application state compiles and runs without error.

1. Make the window active that contains the application state and choose **Run** > **Application State**.
2. On the **Run Application State** dialog box, select the application state you want to run and click **Run**.

   If there are compilation errors, Silk Test Classic displays an error window. Fix the errors and rerun the application state.

Recording Actions

Use the **Record Actions** dialog box to record the actions you perform to test an application. For example, you can also use the dialog box to write a syntactically correct 4Test statement based on your manual interaction with your application. This eliminates the need to search through the documentation for the correct method and its arguments. Once the statement is recorded, click **Paste to Editor** to insert the statement to your script.

This functionality is available only for projects or scripts that use the Classic Agent.

1. Click **Record** > **Actions** to open the **Record Actions** dialog box.
2. Perform the action that you want to record.

   The dialog box displays the GUI object name when you point to an object. You can click **Pause Recording** to review the object properties that you have recorded. When you click **Resume Recording**, the status bar returns.
3. Press `Ctrl+Alt` to verify the action.
4. Click **Paste to Editor** and then click **Close**.

Recording the Location of an Object

You can record the x, y locations of a graphical control, such as a toolbar. It can be useful to know the position of certain objects, for example objects that are drawn, like tools on a toolbar, or drawing regions, for example in a CAD/CAM package. A location is recorded relative to the screen, frame, and client window.

This functionality is available only for projects or scripts that use the Classic Agent.

To record the location of an object:

1. Click **Record** > **Window Locations** to open the **Record Window Locations** dialog box.
2. Position the cursor over the object that you want to record. The dialog box displays the name of the object and its x,y coordinates relative to the screen, the frame (the main window and its window decoration), and the client (the main window minus its window decoration).
3. Press `Ctrl+Alt` to verify the location.
4. Click the option button that corresponds with the object that you want to record and what you intend to do with the recording.

   For example, if you plan to add the location of the toolbar to an existing window declaration, click the **Client** option.

5. Click **Paste to Editor** and then click **Close**.

Recording Window Identifiers

This functionality is available only for projects or scripts that use the Classic Agent.

You can record window identifiers to record the fully qualified name of the GUI object you are pointing at in your test application, which you can then insert into your script.

If you are recording a test that uses hierarchical object recognition this eliminates the need to bring up your test frame file to find the correct identifier for the object.

1. Click **Record** > **Window Identifiers** to open the **Record Window Identifiers** dialog box.
2. Position the cursor over the GUI object that you want to record.

   The text box displays the GUI object name when you point to an object.
3. If necessary, press `Ctrl+Alt` to pause recording the window identifier.

   Note: For any application that uses `Ctrl+Shift` as the shortcut key combination, press `Ctrl +Shift` to pause recording.
4. Select the identifier in the text box and choose one of the following:
   - Click **Paste to Editor** to paste the window identifier into the open file.
   - Click **Copy to Clipboard** to copy the window identifier to the clipboard. Then, paste the code into a different editing window or paste it into the current window at the location of your choice.
5. Click **Close**.

Verification

This section describes how you can verify one or more characteristics, or properties, of an object.

Verifying Object Properties

You will perform most of your verifications using properties. When you verify the properties of an object, a `VerifyProperties` method statement is added to your script. The `VerifyProperties` method verifies the selected properties of an object and its children.

Each object has many characteristics, or properties. For example, dialog boxes can have the following verification properties:

- `Caption`
- `Children`
- `DefaultButton`
- `Enabled`
- `Focus`
- `Rect`
- `State`

`Caption` is the text that displays in the title bar of the dialog box. `Children` is a list of all the objects contained in the dialog box, `DefaultButton` is the button that is invoked when you press **Enter**, and so on. In your test cases, you can verify the state of any of these properties.

You can also, in the same test case, verify properties of children of the selected object. For example, the child objects in the **Find** dialog box, such as the text box **FindWhat** and the check box **CaseSensitive**,will also be selected for verification.

By recording verification statements for the values of one or more of an object's properties, you can determine whether the state of the application is correct or in error when you run your test cases.

Verifying Object Properties (Classic Agent)

This functionality is supported only if you are using the Classic Agent.

Record verification statements to verify the properties of an object.

1. Complete the steps in *Verifying a Test Case*.
2. Click the **Properties** tab and then choose the objects to verify. To verify all or most objects, click **Check All** and then uncheck individual check boxes.
3. Choose the properties to verify in one of the following ways:
4. Click **OK** to close the **Verify Window** dialog box.
5. If you are writing a complete test case, record the cleanup stage and paste the test case into the script. If you have added a verification statement to an existing test case, paste it into your script and close the **Record Actions** dialog box.

   Here are some points to note about the **Property** tab:

   - The **Windows to Verify** list box (left) displays the class and the identifier of all the objects whose properties have been captured. Indentation denotes the hierarchy. A checked check box (left margin) means that the object will be verified. By default, all objects are checked and the first object is selected.
   - The **Properties to Verify** list box (right) displays each property of the selected object and its current value. A checked check box (left margin) means that the property will be verified. By default, the properties of the selected property set (shown in the **Property Set** list box) are checked.
   - The **Property Value** field displays the value of the selected property. You can edit the value in this field if it is not what you want to verify against. The value specified in this field is the value you expect at runtime, that is, the baseline value.

Verifying an Object Using the Verify Function

This functionality is supported only if you are using the Classic Agent.

Use this procedure to verify an object's state using built-in verification methods or other methods in combination with the built-in `Verify` function.

1. Complete the steps in *Verifying a Test Case*.
2. On the **Verify Window** dialog box, click the `Method` tab. Silk Test Classic lists the methods for the selected class on the left.
3. Check the **Include Inherited** check box to see methods that the class inherits.
4. Select the method that will return the expected value and provide any needed arguments.

   You can specify a built-in method or a user-defined method (as long as it returns a value).
5. Click **OK**.
6. Silk Test Classic returns you to the test application.
7. If you are writing a complete test case, record the cleanup stage and paste the test case into the script. If you have added a verification statement to an existing test case, paste it into your script and close the **Record Actions** dialog box.
8. In the editor, wrap the `Verify` function around the method that returns the expected value as follows: Make the method call the first argument, specify the expected value as the second argument, and provide an error message string optionally as the third argument.

For example, here is a test case that verifies that the text in the `TextField Replace.FindWhat` is `myText`. It uses the built-in verification method `VerifyValue`.

```
testcase VerifyMethodTest ()
TextEditor.Search.Replace.Pick ()
Replace.FindWhat.VerifyValue
```

```
("myText")
Replace.Cancel.Click ()
```

Verifying Object Attributes

This functionality is available only for projects or scripts that use the Classic Agent.

Each kind of GUI object in an application has a variety of characteristics, called attributes. For example, a text box has the following attributes:

- `Caret position`, which is the current position of the text insertion cursor, in (line, column) format. For example, a value of {1,1} means that the text insertion cursor is positioned on line 1, column 1.
- `Enabled`, which is the current enabled status of the text box, either true or false.
- `Selected range`, which is the beginning and ending position of the text string currently selected in the field, in (line, column) format. For example a value of {1,12,1,16} means that the selected text begins on line 1, column 12 and ends on line 1, column 16.
- `Selected Text`, which is the string that is currently selected, if any, in the text box.
- `Text`, which is the entire contents of the text box.

By recording verification statements for the values of one or more of an object's attributes, you can determine whether the state of the application is correct or in error when you run your test cases. That is: did the feature you are testing have the expected result?

By selecting the **Verify All Attributes** check box, you can record a test that verifies the state, contents, and value of a GUI object and any objects it contains. This is commonly called a smoke test or a Level 1 test. A smoke test uses the `VerifyEverything` method to verify every aspect of a particular GUI object.

If you need to, you can define and add your own attributes to the built-in hierarchy of GUI classes.

Attributes have been essentially rendered obsolete and have been replaced by properties.

Verifying Attributes of an Object

This functionality is available only for projects or scripts that use the Classic Agent.

1. Click **Options** > **Recorder**. Uncheck the **Verify Using Properties** check box if necessary.

   You will not be able to uncheck this check box if you have enabled enhanced support for Visual Basic; it requires properties for verification.
2. Drive your application to the test state and press `Ctrl+Alt`. Silk Test Classic displays the **Attribute** tab of the **Verify Window** dialog box. The list box on the left shows the attributes for the current object.
3. Select an attribute from the list box or check the **Verify All Attributes** check box. In the **Attribute value** field Silk Test Classic displays the current value of the attribute (that is, the value that exists when you are recording).

   When verifying attributes during recording, the value size limit of the attribute is 256 characters. The name size limit is 32 characters. The total attribute value/name pair limit size is 4K. If the length exceeds 4K, the message `Unable to Get Windows Properties` is displayed.
4. If the current value of the attribute is not the value you want to test for at runtime, edit the **Attribute value** field.

   The value specified in this field is the value you expect at runtime, that is, the baseline value.
5. Click **OK** to accept the attribute and its value.

   Silk Test Classic closes the **Verify Window** dialog box and displays the **Record Status** window. The test case will verify that the object has the attribute value selected. If not, Silk Test Classic writes an error to the results file. With the **Verify Using Properties** check box unchecked, the next time you go to verify an object, the **Verify Window** dialog box will have an **Attribute** tab, instead of a **Property** tab.

Overview of Verifying Bitmaps

A bitmap is a picture of some portion of your application. Verifying a bitmap is usually only useful when the actual appearance of an object needs to be verified to validate application correctness. For example, if you are testing a drawing or CAD/CAM package, a test case might produce an illustration in a drawing region that you want to compare to a baseline. Other possibilities include the verification of fonts, color charts, and certain custom objects.

When comparing bitmaps, keep the following in mind:

- Bitmaps are not portable between GUIs. The format of a bitmap on a PC platform is `.bmp`.
- A bitmap comparison will fail if the image being verified does not have the same screen resolution, color, window frame width, and window position when the test case is run on a different machine than the one on which the baseline image was captured.
- Make sure that your test case sets the size of the application window to the same size it was when the baseline bitmap was captured.
- Capture the smallest possible region of the image so that your test is comparing only what is relevant.
- If practical, do not include the window's frame (border), since this may have different colors and/or fonts in different environments.

Verifying Appearance Using a Bitmap

When you are using the Classic Agent, use this procedure to compare the actual appearance of an image against a baseline image. Or, use it to verify fonts, color charts, or custom objects.

Note: To verify a bitmap when you are using the Open Agent, you can add the `VerifyBitmap` method to your script. The `VerifyBitmap` method is supported for both agents.

1. Complete the steps in *Verifying a Test Case*.
2. On the **Verify Window** dialog box, click the **Bitmap** tab and then select the region to update: **Entire Window**, **Client Area of Window** (that is, without scroll bar or title bar), or **Portion of Window**.
3. In the **Bitmap File Name** text box, type the full path of the bitmap file that will be created.

   The default path is based on the current directory. The default file name for the first bitmap is `bitmap.bmp`. Click **Browse** if you need help choosing a new path or name.
4. Click **OK**. If you selected **Entire Window** or **Client Area of Window**, Silk Test Classic captures the bitmap and returns you to your test application. If you selected **Portion of Window**, position the cursor at the desired location to begin capturing a bitmap. While you press and hold the mouse button, drag the mouse to the screen location where you want to end the capture. Release the mouse button.

   A bitmap comparison will fail if the image being verified does not have the same screen resolution, color, window frame width, and window position as the baseline image.

   Capture the smallest possible region of the image so that your test is comparing only what is relevant.
5. If you are writing a complete test case, record the cleanup stage and paste the test case into the script. If you have added a verification statement to an existing test case, paste it into your script and close the **Record Actions** dialog box.

Overview of Verifying an Objects State

Each class has a set of methods associated with it, including built-in verification methods. You can verify an object's state using one of these built-in verification methods or by using other methods in combination with the built-in `Verify` function.

A class's verification methods always begin with `Verify`. For example, a `TextField` has the following verification methods; `VerifyPosition`, `VerifySelRange`, `VerifySelText`, and `VerifyValue`.

You can use the built-in `Verify` function to verify that two values are equal and generate an exception if they are not. Typically, you use the `Verify` function to test something that does not map directly to a built-in property or method. `Verify` has the following syntax:

```
Verify (aActual, aExpected [, sDesc])
```

`aActual`	The value to verify. `ANYTYPE`.
`aExpected`	The expected value. `ANYTYPE`.
`sDesc`	*Optional:* A message describing the comparison. `STRING`.

Usually, the value to verify is obtained by calling a method for the object being verified; you can use any method that returns a value.

Example: Verify an object

This example describes how you can verify the number of option buttons in the **Direction RadioList** in the **Replace** dialog box of the Text Editor. There is no property or method you can directly use to verify this. But there is a method for **RadioList**, `GetItemCount`, which returns the number of option buttons. You can use the method to provide the actual value, then specify the expected value in the script.

When doing the verification, position the mouse pointer over the **RadioList** and press `Ctrl+Alt`. Click the **Method** tab in the **Verify Window** dialog box, and select the `GetItemCount` method.

Click **OK** to close the **Verify Window** dialog box, and complete your test case. Paste it into a script. You now have the following script:

```
testcase VerifyFuncTest ()
TextEditor.Search.Replace.Pick ()
Replace.Direction.GetItemCount ()
Replace.Cancel.Click ()
```

Now use the `Verify` function to complete the verification statement. Change the line:

```
Replace.Direction.GetItemCount ()
```

to

```
Verify (Replace.Direction.GetItemCount (), 2)
```

That is, the call to `GetItemCount` (which returns the number of option buttons) becomes the first argument to `Verify`. The expected value, in this case, 2, becomes the second argument.

Your completed script is:

```
testcase VerifyFuncTest ()
TextEditor.Search.Replace.Pick ()
Verify (Replace.Direction.GetItemCount (), 2)
Replace.Cancel.Click ()
```

Fuzzy Verification

There are situations when Silk Test Classic cannot see the full contents of a control, such as a text box, because of the way that the application paints the control on the screen. For example, consider a text box whose contents are wider than the display area. In some situations the application clips the text to fit the display area before drawing it, meaning that Silk Test Classic only sees the contents that are visible; not the entire contents.

Consequently, when you later do a `VerifyProperties` against this text box, it may fail inappropriately. For example, the true contents of the text box might be `29 Pagoda Street`, but only `29 Pagoda` displays. Depending on how exactly the test is created and run, the expected value might be `29 Pagoda` whereas the value seen at runtime might be `29 Pagoda Street`, or vice versa. So the test would fail, even though it should pass.

To work around this problem, you can use fuzzy verification, where the rules for when two strings match are loosened. Using fuzzy verification, the expected and actual values do not have to exactly match. The two values are considered to match when one of them is the same as the first or last part of the other one. Specifically, `VerifyProperties` with fuzzy verification will pass whenever any of the following functions would return `TRUE`, where `actual` is the actual value and `expected` is the expected value:

- `MatchStr (actual + "*", expected)`
- `MatchStr ("*" + actual, expected)`
- `MatchStr (actual, expected + "*")`
- `MatchStr (actual, "*" + expected)`

In string comparisons, * stands for any zero or more characters.

For example, all the following would pass if fuzzy verification is enabled:

Actual Value	Expected Value
29 Pagoda	29 Pagoda Street
oda Street	29 Pagoda Street
29 Pagoda Street	29 Pagoda
29 Pagoda Street	oda Street

Enabling fuzzy verification

You enable fuzzy verification by using an optional second argument to `VerifyProperties`, which has this prototype:

```
VerifyProperties (WINPROPTREE WinPropTree [,FUZZYVERIFY FuzzyVerifyWhich])
```

where the `FUZZYVERIFY` data type is defined as:

```
type FUZZYVERIFY is BOOLEAN, DATACLASS, LIST OF DATACLASS
```

So, for the optional `FuzzyVerifyWhich` argument you can either specify `TRUE` or `FALSE`, one class name, or a list of class names.

FuzzyVerifyWhich value

FALSE (default)	Fuzzy verification is disabled.
One class	Fuzzy verification is enabled for all objects of that class. Example `window.VerifyProperties ({…},Table)` enables fuzzy verification for all tables in window (but no other object).
List of classes	Fuzzy verification is enabled for all objects of each listed class. Example `window.VerifyProperties ({…}, {Table, TextField})` enables fuzzy verification for all tables and text boxes in window (but no other object).

TRUE

Fuzzy verification is enabled only for those objects whose `FuzzyVerifyProperties` member is `TRUE`.

To set the `FuzzyVerifyProperties` member for an object, add the following line within the object's declaration:

```
FUZZYVERIFY FuzzyVerifyProperties = TRUE
```

Example: If in the application's include file, the `DeptDetails` table has its `FuzzyVerifyProperties` member set to `TRUE`:

```
window ChildWin EmpData
. . .
   Table DeptDetails
     FUZZYVERIFY FuzzyVerifyProperties = TRUE
```

And the test has this line:

```
EmpData.VerifyProperties ({...}, TRUE)
```

Then fuzzy verification is enabled for the `DeptDetails` table (and other objects in `EmpData` that have `FuzzyVerifyProperties` set to `TRUE`), but no other object.

Fuzzy verification takes more time than standard verification, so only use it when necessary.

For more information, see the `VerifyProperties` method.

Defining your own verification properties

You can also define your own verification properties.

Verifying that a Window or Control is No Longer Displayed

1. Click **Record** > **Testcase** to begin recording a test case and drive your application to the state you want to verify. To record a verification statement in an existing test case, click **Record** > **Actions**.
2. When you are ready to record a verification statement, position the mouse cursor over the object you want to verify, and press `Ctrl+Alt`. Silk Test Classic displays the **Verify Window** dialog box over your application window.
3. Click the **Property** tab. Silk Test Classic lists the properties for the selected window or control on the right.
4. Make sure that only the `Exists` property is selected for the window or control.

   If additional properties are selected, the verification will fail because the actual list of properties will differ from the expected list.
5. Change the value in the **Property Value** field from `TRUE` to `FALSE`.
6. Click **OK** to accept the `Exists` property for the selected window or control. Silk Test Classic closes the **Verify Window** dialog box and displays the **Record Status** window. The test case will verify that the window or control has the property value of `FALSE`, verifying that the object is no longer displayed. If not, Silk Test Classic writes an error to the results file.

Data-Driven Test Cases

Data-driven test cases enable you to invoke the same test case multiple times, once for each data combination stored in a data source. The data is passed to the test case as a parameter. You can think of a data-driven test case as a template for a class of test cases. Data-driven test cases offer the following benefits:

- They reduce redundancy in a test plan.
- Writing a single test case for a group of similar test cases makes it easier to maintain scripts.
- They are reusable; adding new tests only requires adding new data.

Regardless of the technique you use, the basic process for creating a data-driven test case is:

1. Create a standard test case. It will be very helpful to have a good idea of what you are going to test and how to perform the verification.
2. Identify the data in the test case and the 4Test data types needed to store this data.
3. Modify the test case to use variables instead of hard data.
4. Modify the test case to specify input arguments to be used to pass in the data. Replace the hard coded data in the test case with variables.
5. Call the test case and pass in the data, using one of four different techniques:
   - Use a database and the **Data Driven Workflow** to run the test case, the preferred method.
   - Click **Run** > **Testcase** and type the data in the **Run Testcase** dialog box.
   - In a QA Organizer test plan, insert the data as an attribute to a test description.
   - If the data exists in an external file, write a function to read the file and use a `main()` function to run the test case.

Data-Driven Workflow

You can use the **Data Driven Workflow** to create data-driven test cases that use data stored in databases. The **Data Driven Workflow** generates much of the necessary code and guides you through the process of creating a data-driven test case.

Before you can create and run data-driven test cases, you need to perform the following actions:

1. Record a standard test case.
2. Set up or identify the existing data source with the information you want to use to run the test.
3. Configure your Data Source Name (DSN), if you are not using the default, which is *Silk DDA Excel*.

Note: When you use the **Data Driven Workflow**, Silk Test Classic uses a well-defined record format. To run data-driven test cases that were not created through the **Data Driven Workflow**, you need to convert your recordings to the new record format. To run data-driven test cases that do not follow the record format, run the tests outside of the **Data Driven Workflow**.

To enable or disable the **Data Driven Workflow**, click **Workflows** > **Data Driven**.

To create and execute a data-driven test case, sequentially click each icon in the workflow bar to perform the corresponding procedure.

Action	Description
Data Drive Testcase	Select a test case to data drive. Silk Test Classic copies the selected test case and creates a new data-driven test case by adding a `"DD_"` prefix to the original name of the test case. Silk Test Classic also writes other data-driven information to the new or existing data driven script file (`.g.t` file).
Find/Replace Values	Find and replace values in the new test case with links to the data source.
Run Testcase	Run the data-driven test case, optionally selecting the rows and tables in the data source that you want to use.

Action	Description
Explore Results	View test results.

Working with Data-Driven Test Cases

Consider the following when you are working with data-driven test cases:

- The **4Test Editor** contains additional menu selections and toolbars for you to use.
- Silk Test Classic can data drive only one test case at a time.
- You cannot duplicate test case names. Data-driven test cases in the same script must have unique names.
- The Classic 4Test editor is not available with data-driven test cases in `.g.t` files.
- You cannot create data-driven test cases from test cases in `.inc` files; you can only create data-driven test cases from test cases in `.t` or `.g.t` files. However, you can open a project, add the `*.inc`, select the test case from the test case folder of the project, and then select data drive.
- When you data drive a `[use '<script>.t']` is added to the data-driven test case. This is the link to the `.t` file where the test case originated. If you add a test case from another script file then another use line pointing to that file is added. If the script file is in the same directory as the `<script.g.t>`, then no path is given, otherwise, the absolute path is added to the use line. If this path changes, it is up to you to correct the path; Silk Test Classic will not automatically update the path.
- When you open a `.g.t` file using **File** > **Open**, Silk Test Classic automatically loads the data source information for that file. If you are in a `.g.t` file and that file's data source is edited, click **Edit** > **Data Driven** > **Reload Database** to refresh the information from the data source.
- If you add a new data-driven test case to an existing `.g.t` file that is fully collapsed, Silk Test Classic expands the previous test case, but does not edit it.

Code Automatically Generated by Silk Test Classic

When you create a data-driven test case, Silk Test Classic verifies that the DSN configuration is correct by connecting to the database, generates the 4Test code describing the DSN, and writes that information into the data-driven script.

Do not delete or change the information created by Silk Test Classic. If you do, you may not be able to run your data-driven test case.

When you click **OK** on the **Specify Data Driven Testcase** dialog box, Silk Test Classic automatically writes the following information to the top of your data driven script file.

The information is delivered "rolled up" (collapsed); in order to see the details you need to click on the plus sign to expand the code:

```
[+] // *** DATA DRIVEN ASSISTANT Section (!! DO NOT REMOVE !!) ***
```

The .inc files used by the original test cases, and the .t file indicating where the test case just came from, in this case from `Usability.t`:

```
[ ] use "datadrivetc.inc"
[ ] use "Usability.t"
```

A reference to the DSN, specifying the connect string, including username and password, for example:

```
[ ] // *** DSN ***
[ ] STRING gsDSNConnect = "DSN=SILK DDA Excel;DBQ=C:\ddatesting
\TestExcel.xls;UID=;PWD=;"
```

Each data-driven test case takes as a single argument a record consisting of a record for each table that is used in the test case. The record definition is automatically generated as shown here:

```
[+] // testcase VerifyProductDetails (REC_DATALIST_VerifyProductDetails rdVpd)
[ ] // Name: REC_<Testcase name>. Fields Types: Table record types. Field
```

```
Names: Table record
type with 'REC_' replaced by 'rec'
[-] type REC_DATALIST_VerifyProductDetails is record
  [ ] REC_Products recProducts
  [ ] REC_Customers recCustomers
  [ ] REC_CreditCards recCreditCards
```

Each table record contains the column names in the same order as in the database. Spaces in table and column names are removed. Special characters such as $ are replaced by underscores.

```
[ ] // *** Global record for each Table ***
[ ]
[-] type REC_Products_ is record
 [ ] STRING Item //Item,
 [ ] REAL Index //Index,
 [ ] STRING Name //Name,
 [ ] REAL ItemNum //ItemNum,
 [ ] STRING Price //Price,
 [ ] STRING Desc //Desc,
 [ ] STRING Blurb //Blurb,
 [ ] REAL NumInStock //NumInStock,
 [ ] INTEGER QtyToOrder //QtyToOrder,
 [ ] INTEGER OnSale //OnSale,
```

Silk Test Classic writes a sample record for each table. This is the data used if you opt to use sample data on the **Run Testcase** dialog box. A value from the original test case is inserted into the sample record, even if there are syntax errors when that column is first used to replace a value.

```
[ ] // *** Global record containing sample data for each table ***
[ ] // *** Used when running a testcase with 'Use Sample Data from Script'
checked ***
[ ]
[-] REC_Products_ grTest_Products_ = {...}
 [ ] NULL // Item
 [ ] NULL // Index
 [ ] NULL // Name
 [ ] NULL // ItemNum
 [ ] NULL // Price
 [ ] NULL // Desc
 [ ] NULL // Blurb
 [ ] NULL // NumInStock
 [ ] 2 // QtyToOrder
 [ ] NULL // OnSale
[ ]
[ ] // *** End of DATA DRIVEN ASSISTANT Section ***
```

Tips And Tricks for Data-Driven Test Cases

There are several things to know about working with data sources while you are creating data-driven test cases.

- You must have an existing data source with tables and columns defined before you data drive a test case. However, the data source does not need to contain rows of data. You cannot use the *Data Driven Workflow* to create data sources or databases.
- If you have a table in your data source that has a long name (greater than 25 characters), all of the name may not be visible in the **Find and Replace** menu bar in the **4Test Editor**. You may find it helpful to change the size of the menu bar to display more of your table name.
- You cannot change to a different data source once you have started to find and replace values in a script. If you do, you will have problems with prior replacements. If you want to change your data source, you should create a new data-driven script file.
- If you are working with a data source that requires a user name and password, you can add the username and password to the connect string in the `.g.t` file. The first example below shows how SQL Server requires a userid and password. `[ ] STRING gsDSNConnect =`

`"DSN=USER.SQL.DSN;UID=SA;PWD=sesame;"` where UID=<your user ID> ("SA" in the example above) and where `PWD=<your password>` ("sesame" in the example above). On the other hand, the example below shows how the Connect string for a MS Excel DSN does not require user IDs or passwords: `[ ] STRING gsDSNConnect = "DSN=Silk DDA Excel;DBQ=C:\TestExcel.xls;UID=;PWD="`

- You can choose to run with a sample record if the table is empty; however, this record is not inserted into the database. The sample record is created by Silk Test Classic when it replaces values from the test case by the table and columns in your database.
- Real numbers should be stored as valid 4Test Real numbers with format: `[-]ddd.ddd[e[-]ddd]`, even though databases such as MS Excel allow a wider range of formats – for example, currencies and fractions.
- There are no restrictions on how you name your tables and columns within your data source. Silk Test Classic automatically removes spaces, and converts dollar signs and other special characters to underscores when it creates the sample record and writes other code to your data-driven test case. Silk Test Classic handles MS Excel and MS Access table names without putting quotation marks around them. This means that your table and column names will look familiar when you go to find and replace values.
- If you encounter the error "ODBC Excel Driver numeric field overflow" while running a test case, check the Excel workbook that you are using as your data source. You may have some columns that are defined as STRING columns but contain numeric values in some of the rows. If you have a column that you want to treat as numeric strings rather than as numbers, either format the column as 'Text' or begin the number strings with a single-quote character. For example: '1003 instead of: 1003
- If modifying data sources in an existing Excel data sheet, use the **remove column** option to delete any data to be removed, as simply deleting from the cell, using clear contents, or copy/pasting content will not register correctly with the DDS file in Silk Test Classic and may lead to a data source mismatch error: `*** Error: Incompatible types -- Number of list elements exceeds number of fields.`

Formatting MS Excel worksheets for use as a data source

Use the 'General' format for the columns of your worksheets. Here are specific suggestions for column formats based on the intended data type of the column:

Intended Data Type of Column	Excel Column Format
STRING	If the column contains only text, no numbers, dates or booleans, then apply the 'General' format. If the column contains text and numbers, then you can still apply the 'General' format if you begin the number strings with a single-quote character. For example: '1003 instead of: 1003. Otherwise, apply the 'Text' format.
INTEGER or REAL	‘General' or 'Number' format.
BOOLEAN	‘General' format. Use only the values TRUE and FALSE.
DATETIME	‘Custom' format: yyyy-mm-dd hh:mm:ss. That agrees with the ISO format used by Silk Test Classic DATETIME values.

Testing an Application with Invalid Data

This topic assumes that you are familiar with data driving test cases.

To thoroughly test an application feature, you need to test the feature with invalid as well as valid data.

For example, the sample Text Editor application displays a message box if a user specifies a search string in the **Find** dialog box that doesn't exist in the document. To account for this, you can create a data-driven test case, like the following, that verifies that the message box displays and has the correct message:

```
type SEARCHINFO is record
   STRING  sText      // Text to type in document window
   STRING  sPos       // Starting position of search
   STRING  sPattern   // String to look for
   BOOLEAN bCase      // Case-sensitive or not
   STRING  sDirection // Direction of search
   STRING  sExpected  // The expected match
   STRING  sMessage   // The expected message in message box

testcase FindInvalidData (SEARCHINFO Data)
   TextEditor.File.New.Pick ()
   DocumentWindow.Document.TypeKeys (Data.sText + Data.sPos)
   TextEditor.Search.Find.Pick ()
   Find.FindWhat.SetText (Data.sPattern)
   Find.CaseSensitive.SetState (Data.bCase)
   Find.Direction.Select (Data.sDirection)
   Find.FindNext.Click ()

   MessageBox.Message.VerifyValue (Data.sMessage)
   MessageBox.OK.Click ()

   Find.Cancel.Click ()
   TextEditor.File.Close.Pick ()
   MessageBox.No.Click ()
```

The `VerifyValue` method call in this test case verifies that the message box contains the correct string. For example, the message should be `Cannot find Ca` if the user enters `Ca` into the **Find** dialog box and the document editing area does not contain this string.

Enabling and Disabling Workflow Bars

Only one workflow bar can be enabled at a time.

To enable or disable a workflow bar, click **Workflows** and then select the workflow bar that you want to turn on or off. For example, click **Workflows** > **Basic**.

You can select one of the following workflows:

Workflow	Description
Basic workflow	Guides you through the process of creating a test case.
Data Driven workflow	Guides you through the process of creating a data-driven test case.

Data Source for Data-Driven Test Cases

When you install Silk Test Classic, the `SILK DDA EXCEL` DSN is copied to your installation computer. This is the default DSN that Silk Test Classic uses. This DSN uses a MS Excel 8.0 driver and does not have a particular workbook (`.xls` file) associated with it.

The **Select Data Source** dialog box allows you to choose the data source:

- For new data-driven test cases, choose *Silk DDA Excel*.
- For backward compatibility, choose *Segue DDA Excel*. This allows existing `.g.t` files that reference *Segue DDA Excel* to continue to run.

You do not have to use the default DSN. For additional information when using a different DSN, see *Configuring Your DSN*.

You may use any of the following types of data sources:

- Text files and comma separated value files (*.txt and *.csv files)
- Microsoft Excel
- Microsoft SQL Server
- Microsoft Access
- Oracle
- Sybase SQL Anywhere

For information about new features, supported platforms and versions, known issues, and work-arounds, refer to the *Release Notes*.

Configuring Your DSN

The default DSN for data-driven test cases, *Silk DDA Excel*, is created during the installation of Silk Test Classic. To use the default DSN you do not need to configure your DSN.

The **Select Data Source** dialog box allows you to choose the data source:

- For new data-driven test cases, choose *Silk DDA Excel*.
- For backward compatibility, choose *Segue DDA Excel*. This allows existing `.g.t` files that reference *Segue DDA Excel* to continue to run.

The following instructions show how to configure a machine to use a different DSN than the *Silk DDA Excel* default.

1. Click **Start** > **Control Panel** > **System and Security** > **Administrative Tools** > **Data Sources (ODBC)**.
2. On the **ODBC Data Source Administrator**, click either the **System DSN** tab or the **User DSN** tab, depending on whether you want to configure this DSN for one user or for every user on this machine.
3. Click **Add**.
4. On the **Create New Data Source** dialog box, select the driver for the data source and click **Finish**.

   To restore the default DSN for Silk Test Classic, select the driver for Microsoft Excel Driver (*.xls).
5. On the setup dialog box of the data source, enter a name for the data source.

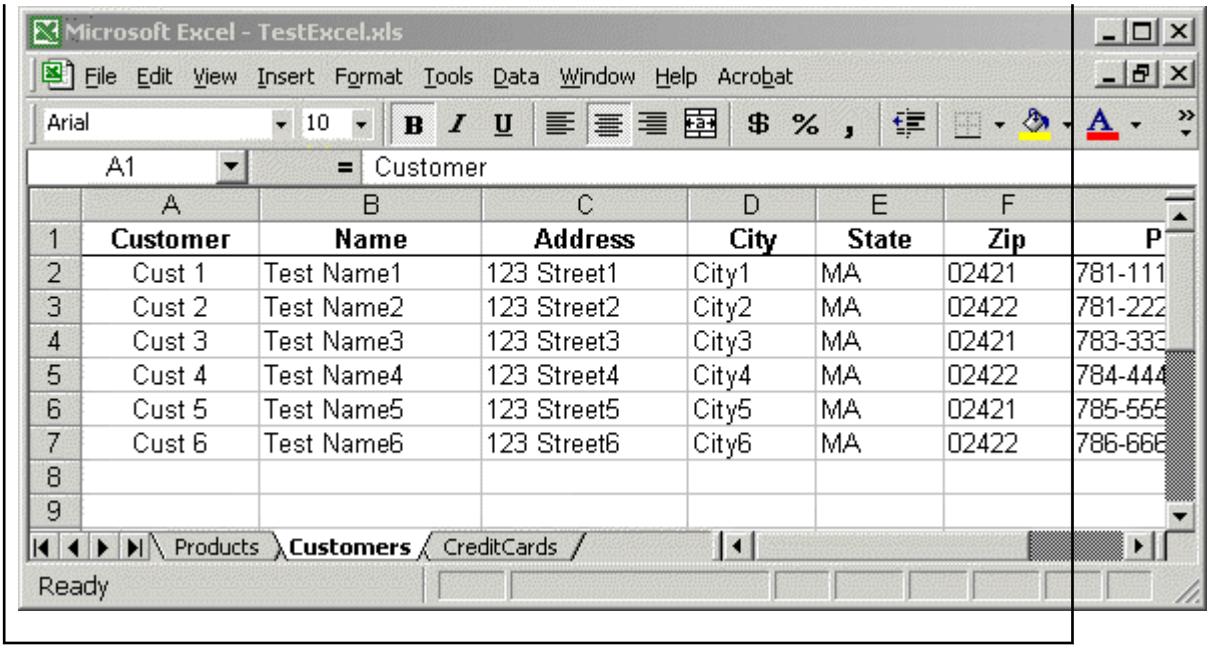
   To restore the default for Silk Test Classic, enter `Silk DDA Excel`. For additional information about the dialog box, refer to the database documentation or contact your database administrator.
6. Click **OK**.

Setting Up a Data Source

Before you can run a data-driven test case you must set up a file that contains the tables, which are called *worksheets* in Microsoft Excel (Excel), and the columns that you want to use. The tables do not have to be populated with data, but it might help to have at least one complete record filled out.

1. Open one of the data sources for data-driven test cases, for example Excel.
2. Name at least one table, or worksheet if you are using Excel, and create column names for the table.
3. Save the data source.

> **Example**
>
> The Excel file `TestExcel.xls` can be used as a data source for a data-driven test case and includes the three worksheets *Products*, *Customers*, and *CreditCards*. The Customers worksheet includes the columns *Customer*, *Name*, *Address*, and so on.



	A	B	C	D	E	F	
1	Customer	Name	Address	City	State	Zip	P
2	Cust 1	Test Name1	123 Street1	City1	MA	02421	781-111
3	Cust 2	Test Name2	123 Street2	City2	MA	02422	781-222
4	Cust 3	Test Name3	123 Street3	City3	MA	02421	783-333
5	Cust 4	Test Name4	123 Street4	City4	MA	02422	784-444
6	Cust 5	Test Name5	123 Street5	City5	MA	02421	785-555
7	Cust 6	Test Name6	123 Street6	City6	MA	02422	786-666
8							
9							

Using an Oracle DSN to Data Drive a Test Case

To use an Oracle DSN to data drive a test case, select the test case to data drive, let Silk Test Classic generate code into the new test case file, and then make the following manual modifications to the DSN:

1. Find out which columns are included in the table of your schema.

   Different schemas may contain tables with the same name. The table lists for the **Find/Replace Values** dialog box, the re-sizable menu bar, and the **Specify Rows** dialog box will list the same table name once for each schema without indicating the schema. For each of those list items the column list will contain the names of the columns in all of the tables with that name.
2. After finding and replacing values, split each table record into separate records according to the schema. Do that for the sample record as well.

   The record names should have the form: `<Record prefix><schema>_<table>`. For example, if the schema is `STUser` and the table is `Customers`, the name of the table record type will be `REC_STUser_Customers` and the declaration for the field in the test case record for the table will be `REC_STUser_Customers recSTUser_Customers // Customers`.
3. Run the test case from a test plan, unless you are running all rows for all tables. Use the **Specify Rows** dialog box to build the *ddatestdata* value, then modify that value to include the schema name in the query.

   Note: Specify a query for every table, even if you want to run all rows for a table. To run all rows, leave the where clause blank.

Creating the Data-Driven Test Case

This section describes how you can create a data-driven test case.

Selecting a Test Case to Data Drive

For information on the steps that you need to complete before you can select a test case to data drive, see *Data-Driven Workflow*.

While you are in a script, choose one of the following to select a test case for data driving:

- Click **Tools** > **Data Drive Testcase**.
- Right-click into the script and select **Data Drive Testcase**.

When you select a test case, Silk Test Classic copies the selected test case and creates a new data-driven test case by adding a `DD_` prefix to the original name of the test case. Silk Test Classic also writes other data-driven information to the new or existing data-driven script file `script.g.t`.

Finding and Replacing Values

For information on the steps that you need to complete before you can find and replace values in a test case, see *Data-Driven Workflow*.

Values are text strings, numbers, and booleans (true/false) that exist in your original test cases. One of the steps in creating a data-driven test case is to find these values and replace them with references to columns in your data source.

Silk Test Classic checks to make sure that each value you select is appropriate for replacement by the column in your test case. You can turn off this validation by clicking **Edit** > **Data Driven** > **Validate Replacements** while you are in a `.g.t` file. This means that the **Find** aspect of **Find and Replace** works as usual, but that the values that you replace are not validated. By turning off this checking, you suppress the error messages that Silk Test Classic would have otherwise displayed. Any 4Test identifier or fragment of a string is considered an invalid value for replacement unless **Validate Replacements** is turned off.

If you are new to creating data-driven test cases, we recommend that you keep this validation turned on.

Find and replace values in a test case using either the **Find/Replace Values** dialog box or the **Find and Replace** re-sizable menu bar in the **4Test Editor**. You can access the **Find/Replace Values** dialog box in one of the following ways:

- Right-click into a test case in a `.g.t` file and select **Data Drive Testcase**. Specify the data source, the data-driven script, and the data-driven test case. When you complete the **Specify Data Driven Testcase** dialog box and the data-driven script opens in the **4Test Editor**, the **Find/Replace Values** dialog box opens automatically.
- After you have highlighted a value in a `.g.t` file, choose **Edit** > **Data Driven** > **Find** > **Replace Values**, or right-click the value and select **Find** > **Replace Values**.

When you are using **Find and Replace**, sometimes a method requires a data type that does not match the column that you want to replace. For example, `SetText` requires a string, but you may want to set a number instead, or perhaps the database does not store data in the 4Test type that you would like to use. Silk Test Classic can handle these kinds of conversions, with a few exceptions.

Running a Data-Driven Test Case

Once you have selected a test case to data drive, and found and replaced values, choose one of the following ways to run the test case:

- Click **Run** > **Run** while in a `.g.t` file. This command runs `main()`, or if there is no `main()`, the command runs all test cases. For each test case, this command runs all rows for all tables used by the test case.
- Click **Run** > **Testcase** and select the data-driven test case from the list of test cases on the **Run Testcase** dialog box, for all tables used by the test case.
- Click **Run** > **Testcase** > **Run** to run the test case for all rows for all tables used by the test case.

Running a Test Case Using a Sample Record for Each Table Used by the Data-Driven Test Case

This is useful if you want to do a quick test or are not connected to your data source. The sample record is created as you replace values in the test case. When you first use a column to replace a test case value, that value is inserted into the table record in the field for that column.

1. On the **Run Testcase** dialog box, click **Use Sample Data from Script**.

By default, Silk Test Classic runs every combination of rows in your tables. The number of test cases that runs is:

```
# of rows selected for Table 1 X the # of rows selected for
Table 2 X the number of rows for Table 3
... and so on
```

For example, if your test case uses 3 tables with 5 rows each, Silk Test Classic will run 125 test cases.

2. To select the rows you want to run on a table-by-table basis, click **Specify Rows** on the **Run Testcase** dialog box to use the **Specify Rows** dialog box to create a query.
3. Specify arguments, if necessary, in the **Arguments** text box. Remember to separate multiple arguments with commas.
4. To wait one second after each interaction with the application under test is executed, check the **Animated Run Mode (Slow-Motion)** check box.

   Typically, you will only use this check box if you want to watch the test case run. For instance, if you want to demonstrate a test case to someone else, you might want to check this check box. Executions of the default base state and functions that include one of the following strings are not delayed:

   - `BaseStateExecutionFinished`
   - `Connecting`
   - `Verify`
   - `Exists`
   - `Is`
   - `Get`
   - `Set`
   - `Print`
   - `ForceActiveXEnum`
   - `Wait`
   - `Sleep`
5. To view results using the Silk TrueLog Explorer, check the **Enable TrueLog** check box. Click **TrueLog Options** to set the options you want to record.
6. Click **Run**. Silk Test Classic runs the test case and generates a results file.

Passing Data to a Test Case

Once you have defined your data-driven test case, you pass data to it, as follows:

- If you are not using the test plan editor, you pass data from a script's main function.
- If you are using the test plan editor, you embed the data in the test plan and the test plan editor passes the data when you run the test plan.

Example Setup for Forward Case-Sensitive Search

Here is a sample application state that performs the setup for all forward case-sensitive searches in the **Find** dialog box:

```
appstate Setup () basedon DefaultBaseState
TextEditor.File.New.Pick ()
DocumentWindow.Document.TypeKeys ("Test Case<Home>")
TextEditor.Search.Find.Pick ()
Find.CaseSensitive.Check ()
Find.Direction.Select ("Down")
```

Building Queries

Before you define a query to access certain data in a data-driven test case, there are several steps you need to complete. for additional information, see *Using the Data Driven Workflow* for more information.

Respond to the prompts on the **Specify Rows** dialog box to create a query for a table. The following are examples of simple queries:

- To find and run the records of customers whose customer ID number is 1001: `(CUSTID = 1001)`
- To find and run the records of customers whose names begin with the letters "F" or "G": `(CUST_NAME LIKE 'F%') OR (CUSTNAME LIKE 'G%')`.

See the description of the enter values area in the **Specify Rows** dialog box to see examples of more complex queries.

Adding a Data-Driven Test Case to a Test Plan

You can run a data-driven test case from a test plan as either a data-driven test case or as a regular test case. To distinguish between the two cases, there are two keywords for you to use:

- ddatestcase specifies the name of a test case that runs as a data-driven test case.
- ddatestdata specifies the list of rows that will be run with the data-driven test case.

If the test case is specified with the keyword ddatestcase, it is run as a data-driven test case. Use this keyword only with data-driven test cases.

To specify a data-driven test case in a test plan

- Add keyword ddatestcase in front of the test case name.
- Add the keyword ddatestdata as a list of queries that specify the particular rows you want the test case to run with. The list of queries is represented as a single `LIST OF STRING` parameter.

Rules for using data-driven keywords

- The ddatestdata keyword requires simple select queries. To specify the row you want to run a test case with, use the ddatestdata keyword with the format: `select * from <table> where ....`
- The keyword ddatestcase cannot be a level above the script file and still work. The script file has to be at the same level or above it.
- A test plan needs to specify a test case using either the keyword testcase or the keyword ddatestcase. Using both causes a compiler error.
- If the ddatestdata keyword is present, then the ddatestcase is run using the ddatestdata value as the rows to run.
- The default is to run all rows for all tables. The value for ddatestdata for this is `ALL_ROWS_FOR_ALL_TABLES`.
- Using the keyword testdata in a test item with keyword ddatestcase will cause a compiler error.
- If the test case is specified with the keyword testcase, then the test case is run as a regular test case and the testdata keyword or symbols must be present to specify the value that will be passed as the regular argument. This value must be a record of the type defined for the ddatestcase, in other words of type `REC_DATALIST_<Testcase name>`.

You can add a data-driven test case to a test plan by using the **Testplan Detail** dialog box or by editing the test plan directly. However, if you edit the test plan directly, then the keywords are not automatically validated and it is your responsibility to make sure that the keywords, which are testcase versus ddatestcase and testdata versus ddatestdata, are appropriate for the intended execution of the test case.

Whenever you use the **Test Detail** dialog box, be sure to click the **Testcases** button and select the test case from the list. That will ensure that the proper keywords are inserted into the test plan.

Using sample records data within test plans

To run a test case with the sample record within a test plan, you must manually input the test data, in the format `ddatestdata: {"USE_SAMPLE_RECORD_<tablename>"}`

For example:

```
script: example.t
ddatestcase: sampletc
ddatestdata: {"USE_SAMPLE_RECORD_SpaceTable$"}
```

You must put the USE_SAMPLE_RECORD_ prefix in front of each table name that you want to run against. If you are using two tables, you need to input the prefix twice, as shown below with two tables named "Table1" and "Table2":

```
ddatestdata: {"USE_SAMPLE_RECORD_Table1","USE_SAMPLE_RECORD_Table2"}
```

Using a main Function in the Script

Although most of the script files you create contain only test cases, in some instances you need to add a function named main to your script. You can use the main function to pass data to test cases as well as control the order in which the test cases in the script are executed.

When you run a script file by clicking **Run** > **Run**:

- If the script file contains a main function, the main function is executed, then execution stops. Only test cases and functions called by main will be executed, in the order in which they are specified in main.
- If the script does not contain a main function, the test cases are executed from top to bottom.

Example

The following template shows the structure of a script that contains a main function that passes data to a data-driven test case:

```
main ()
// 1. Declare a variable to hold current record
// 2. Store all data for test case in a list of records
// 3. Call the test case once for each record in the list
```

Using this structure, the following example shows how to create a script that defines data records and then calls the sample test case once for each record in the list:

```
type SEARCHINFO is record
   STRING  sText      // Text to type in document window
   STRING  sPos       // Starting position of search
   STRING  sPattern   // String to look for
   BOOLEAN bCase      // Case-sensitive or not
   STRING  sDirection // Direction of search
   STRING  sExpected  // The expected match

main ()
   SEARCHINFO Data
   list of SEARCHINFO lsData = {...}
      {"Test Case", "<END>", "C", TRUE, "Up", "C"}
      {"Test Case", "<END>", "Ca", TRUE, "Up", "Ca"}
      // additional data records can be added here
   for each Data in lsData
      FindTest (Data)

testcase FindTest (SEARCHINFO Data)
   TextEditor.File.New.Pick ()
   DocumentWindow.Document.TypeKeys (Data.sText + Data.sPos)
   TextEditor.Search.Find.Pick ()
   Find.FindWhat.SetText (Data.sPattern)
   Find.CaseSensitive.SetState (Data.bCase)
   Find.Direction.Select (Data.sDirection)
   Find.FindNext.Click ()
   Find.Cancel.Click ()
   DocumentWindow.Document.VerifySelText ({Data.sExpected})
   TextEditor.File.Close.Pick ()
     MessageBox.No.Click ()
```

When you click **Run** > **Run**, the main function is called and the `FindTest` test case will be executed once for every instance of Data in lsData (the list of SEARCHINFO records). In the script shown above, the test case will be run twice. Here is the results file that is produced:

```
Script findtest.t - Passed
Passed:  2 tests (100%)
Failed:  0 tests (0%)
Totals:  2 tests, 0 errors, 0 warnings

Testcase FindTest ({"Test Case", "<END>", "C", TRUE, "Up", "C"}) - Passed
Testcase FindTest ({"Test Case", "<END>", "Ca", TRUE, "Up", "Ca"}) - Passed
```

Note: With data-driven test cases, Silk Test Classic records the parameters that are passed in, in the results file.

In this sample data-driven test case, the test case data is stored in a list within the script itself. It is also possible to store the data externally and read records into a list using the `FileReadValue` function.

Using do...except to Handle an Exception

The `VerifyValue` method, like all 4Test verification methods, raises an exception if the actual value does not match the expected (baseline) value. When this happens, Silk Test Classic halts the execution of the test case and transfers control to the recovery system. The recovery system then returns the application to the base state.

However, suppose you don't want Silk Test Classic to transfer control to the recovery system, but instead want to trap the exception and handle it yourself. For example, you might want to log the error and continue executing the test case. To do this, you can use the 4Test `do...except` statement and related statements, which allow you to handle the exception yourself.

Property Sets

This functionality is supported only if you are using the Classic Agent.

This section describes how properties are organized into sets to make your testing easier.

Verifying Properties as Sets

This functionality is supported only if you are using the Classic Agent.

To make your testing easier, properties are organized into sets. A property set consists of a list of properties and the class associated with each property. A number of property sets is predefined for your convenience.

Properties and attributes in this context are similar—they both are used to verify a characteristic of an object. However, properties are more encompassing, more flexible, and easier to use. For example, using attributes you can only verify one attribute at a time or verify every attribute for an object and all its children; using properties you can verify selected properties of an object and any or all of its children at the same time.

All property sets reside in the file named in the **Data file for property sets** text box in the **Recorder Options** dialog box. The default file location is your Silk Test Classic installation directory. To make sure that all testers in your group have access to the same property sets file, place the file on a shared drive and specify the full path in the **Data file for property sets** text box.

If you selected enhanced support for Visual Basic, your property set file is `vbprpset.ini`. If you did not select enhanced support for Visual Basic, then your property set file is `propset.ini`.

You can configure property sets to suit your needs, even combining frequently used property sets into a new larger property set.

Creating a New Property Set

This functionality is supported only if you are using the Classic Agent.

1. Click **Options** > **Property Sets**. Silk Test Classic displays the **Property Sets** dialog box, which lists all existing property sets. You can also click **Define** on the **Verify Window** dialog box.
2. On the **Property Sets** dialog box, click **New**.
3. Specify a name for the new property set in the **Name** text box.

   Property set names are not case sensitive. They can be any length and consist of any combination of alphabetic characters, numerals, and underscore characters.
4. Specify a class in the **Class** text box and then a property of that class in the **Property** text box.
5. Click **Add**. Silk Test Classic adds the class-property pair to the list box. The class or property name is not validated here, so type carefully. Invalid names are ignored at runtime. If you make a mistake, select the class-property pair and click **Edit**.
6. Repeat steps 4 and 5 for as many class-property pairs as you want to add. Delete any class-property pairs you don't want to include by selecting them and clicking **Remove**.
7. Once the list of classes and properties is correct, click **OK**. Silk Test Classic closes the **New Property Set** dialog box and displays the new property set in the **Property Sets** list box.
8. Click **Close**.

Combining Property Sets

This functionality is supported only if you are using the Classic Agent.

1. Click **Options** > **Property Sets** to display the **Property Sets** dialog box.

   You can also click **Define** on the **Verify Window** dialog box.
2. Click **Combine**.
3. On the **Combine Property Sets** dialog box, specify a name for the new property set in the **Name** text box.
4. Select at least two property sets from the **Property sets to combine** list box and click **OK**. Silk Test Classic closes the **Combine Property Sets** dialog box and displays the new property in the **Property Sets** list box, along with the constituent sets.

If you modify any of the constituent sets, the combined set will be modified as well.

Deleting a Property Set

This functionality is supported only if you are using the Classic Agent.

1. Click **Options** > **Property Sets**.

   You can also click **Define** on the **Verify Window** dialog box.
2. Select the name of the property set you want to delete from the **Property Sets** list box and then click **Remove**.
3. Silk Test Classic prompts you are to confirm the deletion. Click **Yes** to delete the property set.
4. Click **Close**.

Editing an Existing Property Set

This functionality is supported only if you are using the Classic Agent.

1. Click **Options** > **Property Sets** to display the **Property Sets** dialog box.

   You can also click **Define** on the **Verify Window** dialog box.

2. On the **Property Sets** dialog box, select a property set from the **Property Sets** list box and click **Edit**.
3. Take any of the following actions:

Edit the property set name	Edit the name in the **Name** text box.
Add a class-property pair	1. Specify a class in the **Class** text box. 2. Specify a property for the class in the **Property** text box. 3. Click **Add**.
Delete a class-property pair	Select a class-property pair and click **Remove**. The pair is deleted from the list box.
Edit a class-property pair	1. Select a class-property pair and click **Edit**. The class and property display in the text boxes at the bottom of the dialog box and the **Add** pushbutton becomes **Replace**. 2. Modify the class, property, or both, and click **Replace**. Silk Test Classic displays the class-property pair in the list box.

4. When you finish editing, click **OK**.
5. Click **OK** to close the **Property Set** dialog box.

Specifying a Class-Property Pair

This functionality is supported only if you are using the Classic Agent.

You can specify class-property pairs in the following ways:

- You can specify them as a full class name and a full property name. For example, to specify the **State** property for the `CheckBox` class, enter `CheckBox` in the **Class** text box and `State` in the **Property** text box.
- You can use the * wildcard character for partial matches. The asterisk matches zero or more characters. For example, specifying * as a class name matches all classes. Specifying Text* as a class name matches all classes that begin with the string "Text".
- You can apply the rule of inheritance to property sets; that is, the properties of a class are inherited by its child classes. For example, specifying the **Enabled** property and the `Control` class as a pair means that the **Enabled** properties of all classes, which are descended from `Control`, are also implicitly included in the property set.

Predefined Property Sets

This functionality is supported only if you are using the Classic Agent.

The predefined built-in property sets include:

This property set	Includes properties that describe
Children	Objects within the selected object, such as pushbuttons in a dialog box.
Control State	The state of controls, for example, whether a control is enabled.
Menu State	The state of a menu, for example, whether it's enabled or checked.
Moveable Window State	The state of a moveable window, for example, whether it's enabled or the control that has focus.

This property set	Includes properties that describe
Selection	The currently selected row or current selection in an editable field.
Style	Style variations for controls and objects.
Value Range	Information that governs the range of possible values for controls and objects.
Values (default)	The current value of a control or object, for example, the text in a text box.
Window Location and Size	The position and size of objects on the screen.

If you have enabled an extension to provide enhanced support for testing an application built with a particular development environment, there might be additional property sets. For additional information, refer to the online Help for the extension.

If you are testing a Web application, there are additional property sets.

Characters Excluded from Recording and Replaying

The following characters are ignored by Silk Test during recording and replay:

Characters	Control
...	`MenuItem`
tab	`MenuItem`
&	All controls. The ampersand (&) is used as an accelerator and therefore not recorded.

Testing in Your Environment with the Classic Agent

This section describes how you can test applications in your environment with the Classic Agent.

Distributed Testing with the Classic Agent

This section describes how you can run tests on multiple machines.

Configuring Your Test Environment

This topic contains information about configuration tasks that you can perform on your test environment to test on multiple machines.

When you are working with ...	Configure the following ...
PC-Class Platforms	Explicitly assign a unique network name to remote agents so that Silk Test Classic can identify the agent when your test case connects to that machine.
TCP/IP	On PCs. Windows machines generally come with TCP/IP. Silk Test Classic on Microsoft Windows can use any TCP/IP software package that supports the Windows Sockets Interface Standard, Version 1.1, (WINSOCK), and supplies `WINSOCK.DLL`.
LAN Manager or Windows for Workgroups	• This functionality is supported only if you are using the Classic Agent. • Increase the SESSIONS value, the default is 6, to a higher value. This variable is defined in the `protocol.ini` file, which is typically located in your Windows directory. • Increase the NCBS value in `protocol.ini` to twice the SESSIONS value. • The LAN Manager network environment and Windows for Workgroups have the ability to use more than one protocol driver at a time. NetBEUI is the protocol driver frequently used by LAN Manager. In order for Silk Test Classic and the agent to run, the NetBEUI protocol must be the first protocol loaded. The LANABASE option under the [NETBEUI_XIF] section of `protocol.ini` must be set to 0 (zero). If additional protocols are loaded, they must have a sequentially higher LANABASE setting. For example, if you are running both NetBEUI and TCP/IP, the LANABASE setting for NetBEUI is (as always) 0 (zero), and the value for TCP/IP is1 (one).
NetBIOS on PCs	• This functionality is supported only if you are using the Classic Agent. • Under Windows, install NetBEUI with NetBIOS. • In the Network control panel, set NetBEUI as the default protocol. • On Windows, NetBIOS is started automatically. • Explicitly assign a unique network name to remote agents so that Silk Test Classic can identify the agent when your test case issues a `Connect` function for that machine. This step is not necessary for agents using TCP/IP because Silk Test Classic automatically uses the workstation's TCP/IP name. The name must be from 1 to 16 alphanumeric characters long and must not be the standard name you use for your machine itself or the name of any other distributed agent. On some systems, using the same name can cause a system crash. A safe alternative is to

When you are working with ...	Configure the following ...

derive the agent name from the machine name. For example, if a machine is called *Rome*, call the Agent *Rome_QAP*.

- Your NetBIOS adapter may be configured as any host adapter number, including adapter 0. Check with your network administrator if you are not sure how to do this or need to change your configuration.

Client/Server Testing Configurations

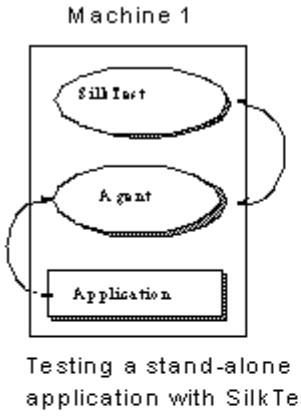
The processes that participate in a client/server testing scenario are logically associated with three different computers:

1. System A runs Silk Test Classic, which processes test scripts and sends application commands to the agent.
2. System B runs the client application and the agent, which submits the application commands to the client application.
3. System C runs the server software, which reacts to requests submitted by the client application.

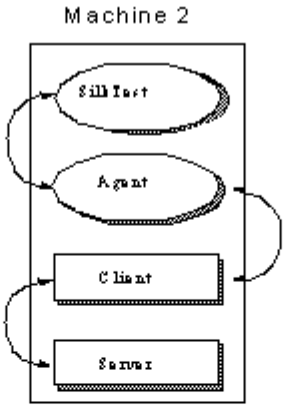
The following sections describe different hardware/software configurations that can support Silk Test Classic testing.

Configuration 1

Machine 1 shows the software configuration you would have when testing a stand-alone application. Machine 2 shows Silk Test Classic and a client/server application with all of your software running on one machine. This configuration allows you to do all types of functional testing other than testing the behavior of the connection between a client and a remote server.



Testing a stand-alone application with SilkTest



Testing a client/server application with software all in one machine

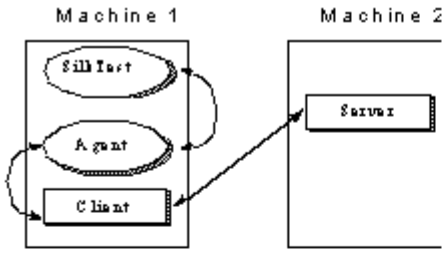
During your initial test development phase, you can reduce your hardware needs by making two (and possibly all) of these systems the same. If you write tests for an application running on the same system as

Silk Test Classic, you can implement the tests without consideration of any of the issues of remote testing. You can then expand your testing program incrementally to take your testing into each new phase.

Configuration 2

A testing configuration in which the client application runs on the same machine as Silk Test Classic and the server application runs on a separate machine.

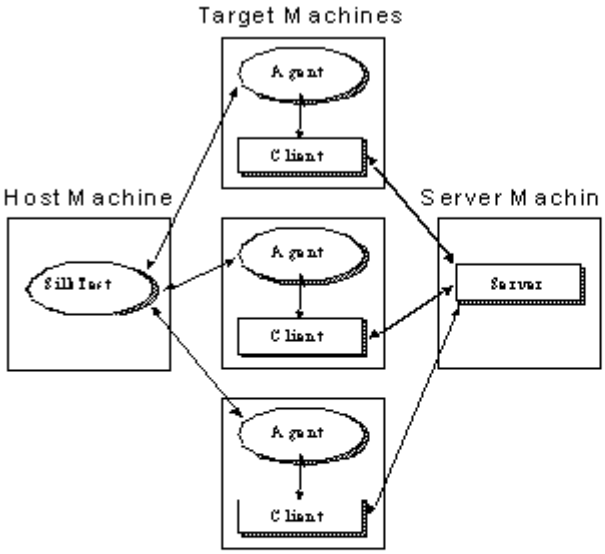
Note: In this configuration, as with Machine 2 in Configuration 1, there is no communication between Silk Test Classic and the server. This means that you must manage the work of starting and initializing the server database manually. For some kinds of testing this is appropriate.



This configuration lets you test the remote client-to-server connection and is appropriate for many stress tests. It allows you to do volume load testing from the point of view of the client application, but not the server.

Configuration 3

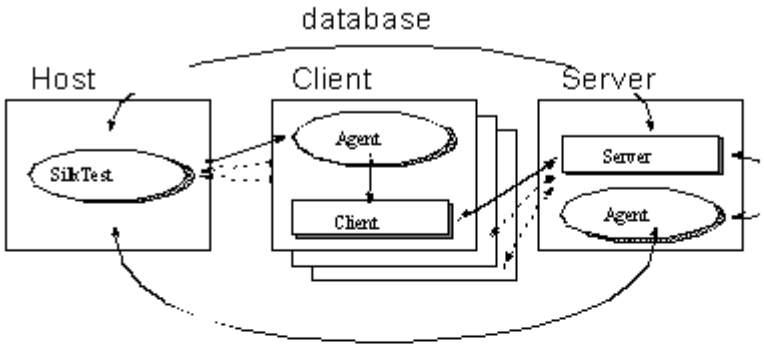
Multiple copies of the client application running on separate machines, with Silk Test Classic driving the client application by means of the agent process on each client machine, and the client application driving the server application. This is just the multi-client version of the previous configuration. You could run a fourth instance of the client application on the Silk Test Classic machine. The actual number of client machines used is your choice.



This configuration is appropriate for load testing and configuration testing if you have no need to automatically manipulate the server. You must have at least two clients to test concurrency and mutual-exclusion functionality.

Configuration 4

Once you are running Silk Test Classic, it makes sense to have your script initialize your server automatically. Configuration 4 uses the same hardware configuration as Configuration 3, but Silk Test Classic is also driving the server directly. This figure shows Silk Test Classic using an agent on the server machine to drive the server's GUI (the lower connecting arrow); this approach can be used to start the server's database and sometimes can be used to initialize it to a base state. The upper arrow shows Silk Test Classic using SQL commands to directly manipulate the server database; use this approach when using the agent is not sufficient. After starting the database with the agent, use SQL commands to initialize it to a base state. The SQL commands are submitted by means of Silk Test Classic's database functions, which do not require the services of the agent.



Configuration 4 is the most complete testing configuration. It requires the database tester. You can use it for all types of Silk Test Classic testing, including volume load testing of the server, peak load testing, and performance testing.

The special features that allow Silk Test Classic to provide rigorous testing for client/ server applications are the following:

- Automatic control of multiple applications.
- Multithreading for automatic control of concurrent applications.
- Reporting results by thread ID.
- Testing across networks using a variety of protocols.

The added value that the database tester provides for the client/server environment is direct database access from the test script.

Networking Protocols Used by the Classic Agent

The Classic Agent uses only three different protocols, although Silk Test Classic runs on many platforms. This means that a Silk Test Classic script on one platform can drive the agent on a target platform, as long as both the host and the agent platforms are running the same appropriate protocol for the platform, regardless of the protocols used by the applications under test. The following table lists the protocols available for each platform:

Platform	TCP/IP	NetBIOS	NetBEUI
Windows	•	•	•
AIX	•		
IRIX	•		

There is no limit on the protocol or API that an application under test may use. Just make sure that the protocol required by Silk Test Classic and the protocol required by your application are running at the same time.

Example

Suppose you are running Silk Test Classic under Windows and you are testing an application that requires TCP/IP communications in order to communicate with a server on a Sun Sparc station. The Windows machine on which Silk Test Classic is running can run NetBIOS for the host and the Windows machine with the application under test must then run NetBIOS for the agent and TCP/IP for the application under test. Running NetBIOS has no impact on your TCP/ IP connections but allows Silk Test Classic to communicate with the agent. Alternatively, since the application is already running TCP/IP, you can choose to use TCP/IP for Silk Test Classic and the Silk Test Classic agents as well.

Single Local Applications

In a single-application test environment, if the application is local, you do not have to determine an agent name or issue a connection command. When you start an agent on the local machine, Silk Test Classic automatically connects to it and directs all agent commands to it.

Remote Applications

When you have one or more remote agents in your testing network, you enable networking by specifying the network type.

For projects or scripts that use the Classic Agent, if you are not using TCP/IP, you have to assign to each agent the unique name that your scripts use to direct test operations to the associated application. For additional information, see *Enabling Networking and Assigning the Classic Agent Name and Port*.

You can use Silk Test Classic to test two applications on the same target from one host machine.

Single Remote Applications

In a single-application test environment, if the application is remote, specify the agent name in the **Runtime Options** dialog box. This causes Silk Test Classic to automatically connect to that machine and to direct all agent commands to that machine. This contrasts with the multi-application case, in which you explicitly connect to the target machines and explicitly specify which machines are to receive which sections of code.

Multiple Remote Applications

When you enable networking by selecting the networking type in the **Runtime Options** dialog box on the host, do not set the **Agent Name** text box to an agent name if you have multiple remote agents. This field only accepts a single agent name and using it prevents you from handling all your client machines the same way.

If you specify one agent name from your set of agents, then you cannot issue a `Connect` call to that agent and thus do not receive the machine handle that the `Connect` function returns. Since you have to issue some `Connect` calls, be consistent and avoid writing exception code to handle a machine that is automatically connected.

For projects or scripts that use the Classic Agent, you can specify multiple agents from within your script file by adding the following command line to the agent:

```
agent –p portNumber
```

Configuring a Network of Computers

To configure a network of computers so that they can run Silk Test Classic and the Silk Test Classic agents, perform the following steps:

1. Install, or have already running, networking protocols supported by Silk Test Classic.
2. Install Silk Test Classic on the host machine and the agent software on all target machines.
3. Establish connectability between host and agents.

   This may be automatic or may require some setup, depending on the circumstances.
4. Enable networking on any target machines.

   Use the **Agent** window, as described in *Enabling Networking and Assigning the Classic Agent Name and Port*.
5. Enable networking on the host machine.

   Use the **Runtime Options** dialog box. Details may vary, depending on your configuration.
6. Gather the information that your test scripts need when making explicit connections.

   For example, you can edit the agent names into a list definition and have your test plan pass the list variable name as an argument for test cases controlled by that plan. The test cases then pass each agent name to a `Connect` or `SetUpMachine` function and that function makes the explicit host-to-agent connection.

Configuration details are specific to the different protocols and operating systems you are using. In general, set up your Agents and make all adjustments to the `partner.ini` file or environment variables before enabling networking on the host machine.

Enabling Networking and Assigning the Classic Agent Name and Port

This describes the process for enabling the Classic Agent on Windows. You only need to execute this procedure the first time you run the Agent.

1. Start the Classic Agent.
2. Right-click the **Agent** icon and choose **Network**. The **Agent Network** dialog box displays.
3. Select the network type from the **Network** list box.
4. For TCP/ IP, the default port number displays in the **Port number** text box. Typically, you accept the default.
5. For NetBIOS, type the Agent name in the **Agent name** text box in the format `hostname:#` where hostname is a unique host name on your network and # is the host adapter number (for example, dwc: 3).

   Your NetBIOS adapter may be configured as any host adapter number. In the past, Silk Test Classic could only be configured as adapter 0, but this is no longer the case. Check with your network administrator if you are not sure how to do this or need to change your configuration.
6. Click **OK**.

Enabling Networking on NetBIOS Host

This functionality is supported only if you are using the Classic Agent.

Once the protocol has been selected for all the Agents and they are named, you can enable networking on the host. Do this by choosing **Options** > **Runtime** and selecting the NetBIOS network type. Then fill in the Agent name if you have a single-remote-application configuration.

Your NetBIOS adapter may be configured as any host adapter number. In the past, Silk Test Classic could only be configured as adapter 0, but this is no longer the case. Check with your network administrator if you are not sure how to do this or need to change your configuration.

Enabling Networking on an Agent

This functionality is supported only if you are using the Classic Agent.

You assign the selected name to the Agent when you enable networking for each Agent PC.

For each Agent, the Agent name that you specify in the `Connect` function is the name of that Agent host, stored in the network host database. You can find the host name in the name given to the Agent icon. The name takes the form `Agent [TCP/IP <Host Name> <Port Number>]`.

For Windows, move the mouse pointer over the Agent icon and wait two seconds; the icon name displays.

You must enable networking for each Agent PC by selecting the protocol type to be used in the Agent window. When you select TCP/IP as the protocol, the port number field is displayed with the default TCP/IP port number. When you click **OK**, the selection is accepted if the default port is available.

Enabling Networking on a Remote Host

Once the protocol has been picked for any PC agents and the port settings are consistent, you can enable networking on the host.

Do this by choosing **Options** > **Runtime** and setting the port number and/or agent name. You can skip this step if you do not have to change the default port number and you are not specifying an agent name for a single-remote-application configuration.

Running Test Cases in Parallel

A concurrent, or multithreaded, script is one in which multiple statements can execute in parallel. Concurrency allows you to more effectively test distributed systems, by permitting multiple client applications to submit requests to a server simultaneously.

The 4Test language fully supports the development of concurrent scripts which enables a script to:

- Create and coordinate multiple concurrent threads.
- Protect access to variables, which are global to all threads.
- Synchronize threads with semaphores.
- Protect critical sections of code for atomic operations.
- Recover from errors in the event of script deadlock.

Concurrency

For Silk Test Classic, concurrent processing means that Agents on a specified set of machines drive the associated applications simultaneously. To accomplish this, the host machine interleaves execution of the sets of code assigned to each machine. This means that when you are executing identical tests on several machines, each machine can be in the process of executing the same operation. For example, select the Edit.FindChange menu item.

At the end of a set of concurrent operations, you will frequently want to synchronize the machines so that you know that all are ready and waiting before you submit the next operation. You can do this easily with 4Test.

There are several reasons for executing test cases concurrently:

- You want to save testing time by running your functional tests for all the different platforms at the same time and by logging the results centrally, on the host machine.
- You are testing cross-network operations.
- You need to place a multi-user load on the server.
- You are testing the application's handling of concurrent access to the same database record on the server.

To accomplish testing concurrent database accesses, you simply set all the machines to be ready to make the access and then you synchronize. When all the machines are ready, you execute the operation that commits the access operation—for example, clicking **OK**. Consider the following example:

```
// [A] Execute 6 operations on all machines concurrently
for each sMachine in lsMachine
  spawn
    SixOpsFunction (sMachine)
```

```
rendezvous                          // Synchronize

// [B] Do one operation on each machine
for each sMachine in lsMachine
  spawn
    [sMachine]MessageBox.OK.Click () // One operation
rendezvous                              // Synchronize
```

In code fragment [A], the six operations defined by the function `SixOpsFunction` are executed simultaneously on all machines in a previously defined list of Agent names. After the parallel operation, the script waits for all the machines to complete; on completion, they will present a message box, unless the application fails. In code fragment [B], the message box is dismissed. By putting the message dismissal operation into its own parallel statement block instead of adding it to the `SixOpsFunction`, you are able to synchronize and all machines click at almost the same instant.

In order to specify that a set of machines should execute concurrently, you use a 4Test command that starts concurrent threads. In the fragments above, the spawn statement starts a thread for each machine.

Global Variables

Suppose the code for each machine is counting instances of some event. You want a single count for the whole test and so each machine adds its count to a global variable. When you are executing the code for all your machines in parallel, two instances of the statement iGlobal = iGlobal + iCount could be executing in parallel. Since the instructions that implement this statement would then be interleaved, you could get erroneous results. To prevent this problem, you can declare a variable shareable. When you do so, you can use the access statement to gain exclusive access to the shared variable for the duration of the block of code following the access statement. Make variables shareable whenever the potential for conflict exists.

Recovering Multiple Tests

There are three major categories of operations that an Agent executes on a target machine:

- Setup operations that bring the application to the state from which the next test will start.
- Testing operations that exercise a portion of the application and verify that it executed correctly.
- Cleanup operations that handle the normal completion of a test plus the case where the test failed and the application is left in an indeterminate state. In either case, the cleanup operations return the application to a known base state.

When there are multiple machines being tested and more than one application, the Agent on each machine must execute the correct operations to establish the appropriate state, regardless of the current state of the application.

Remote Recording

Once you establish a connection to a target machine, any action you initiate on the host machine, which is the machine running Silk Test Classic, is executed on the target machine.

With the Classic Agent, one Agent process can run locally on the host machine, but in a networked environment, the host machine can connect to any number of remote Agents simultaneously or sequentially. You can record and replay tests remotely using the Classic Agent. If you initiate a Record/Testcase command on the host machine, you record the interactions of the user manipulating the application under test on the target machine. In order to use the Record menu's remote recording operations, you must place the target machine's name into the **Runtime Options** dialog box. Choose **Options** > **Runtime**.

With the Open Agent, one Agent process can run locally on the host machine. In a networked environment, any number of Agents can replay tests on remote machines. However, you can record only on a local machine.

Threads and Concurrent Programming

Silk Test Classic can run test cases in parallel on more than one machine. To run test cases in parallel, you can use parallel threads within `main( )` or in a function called by `main( )`. If you attempt to run test cases in parallel on the same machine, you will generate a runtime error.

A more elegant alternative to parallel threads is to use a multitestcase function, which provides a robust multi-machine recovery system. For additional information on multitestcase code templates, see *Using the Client/Server Template* and *Using the Parallel Template*.

In the 4Test environment, a thread is a mechanism for interleaving the execution of blocks of client code assigned to different Agents so that one script can drive multiple client applications simultaneously. A thread is part of the script that starts it, not a separate script. Each thread has its own call stack and data stack. However, all the threads that a script spawns share access to the same global variables, function arguments, and data types. A file that one thread opens is accessible to any thread in that script.

While the creation of a thread carries no requirement that you use it to submit operations to a client application, the typical reason for creating a multithread script is so that each thread can drive test functions for one client, which allows multiple client application operations to execute in parallel.

When a script connects to a machine, any thread in that script is also connected to the machine. Therefore, you must direct the testing operations in a thread to a particular Agent machine. Threads interleave at the machine instruction level; therefore, no single 4Test statement is atomic with respect to a statement in another thread.

Driving Multiple Machines

When you want to run tests on multiple machines simultaneously, you connect to all the machines and then you direct specific test operations to particular machines. This enables you to drive different applications concurrently. For example, you can test the intercommunication capabilities of two different applications or you can drive both a client application and its server.

To do this, at the beginning of a test script you issue for each machine an explicit connection command. This can be either `Connect`(*agent_name*) or `SetMachine`(*agent_name*). This connection lasts for the duration of the script unless you issue a `Disconnect`(*agent_name*) command. In the body of the script you can specify that a particular portion of code is to be executed on a particular machine. The `SetMachine`(*agent_name*) command specifies that the following statements are directed to that Agent. You can specify that just one statement is directed to a particular Agent by using the bracket form of the machine handle operator. For example `["Client_A"]SYS_SetDir ("c:\mydir")`.

Since 4Test allows you to pass variables to these functions, you can write a block of code that sends the same operations to a particular set of target machines and you can pass the `SetMachine` function in that block of code a variable initialized from a list that specifies the machines in that set. Thus, specifying which machines receive which operations is very simple.

Protecting Access to Global Variables

When a new thread is spawned, 4Test creates a new copy of all local variables and function arguments for it to use. However, all threads have equal access to global variables. To avoid a situation in which multiple threads modify a variable simultaneously, you must declare the variable as shareable. A shareable variable is available to only one thread at a time.

Instances where threads modify variables simultaneously generate unpredictable results. Errors of this kind are difficult to detect. Make variables shareable wherever the potential for conflict exists.

A declaration for a shareable variable has the following form:

```
[scope] share data-type name [= expr] {, name [= expr]}
```

- *scope* can be either public or private. If omitted, the default is public.

- *data-type* is a standard or user-defined data type.
- *name* is the identifier that refers to the shareable variable.
- *expr* is an expression that evaluates to the initial value you want to give the variable. The value must have the same type you gave the variable. If you try to use a variable before its value is set, 4Test raises an exception.

At any point in the execution of a script, a shared variable can only be accessed from within the block of code that has explicitly been granted access to it. You request access to shareable variables by using the access statement.

An access statement has the following form:

```
access name1, name2, ...
  statement
```

where *name1*, *name2*, ... is a list of identifiers of optional length, each of which refers to a shareable variable and `statement` is the statement to be executed when access to the variables can be granted.

If no other thread currently has access to any of the shareable variables listed, 4Test executes the specified statement. Otherwise, 4Test blocks the thread where the `access` statement occurs until access can be granted to all the shareable variables listed. At that point, 4Test blocks competing threads and executes the blocked thread.

Example

```
share INTEGER iTestNum = 0
public share STRING asWeekDay [7]
share ANYTYPE aWhoKnows

void IncrementTestNum ()
  access iTestNum
    iTestNum = iTestNum + 1
```

Synchronizing Threads with Semaphores

Use semaphores to mutually exclude competing threads or control access to a resource. A semaphore is a built-in 4Test data type that can only be assigned a value once. The value must be an integer greater than zero. Once it is set, your code can get the semaphore's value, but cannot set it.

Example

The following code example shows legal and illegal manipulations of a variable of type *SEMAPHORE*:

```
SEMAPHORE semA = 10                          // Legal
semA = 20                                    // Illegal -
existing semaphore
                                             // cannot be
reinitialized
if (semA == [SEMAPHORE]2)...                 // Legal - note the
typecast
Print ("SemA has {semA} resources left.")  // Legal
SEMAPHORE semB = 0                           // Illegal - must be
greater than 0
```

To compare an integer to a semaphore variable, you must typecast from integer to semaphore using `[SEMAPHORE]`.

Note: You cannot cast a semaphore to an integer.

To use a semaphore, you first declare and initialize a variable of type *SEMAPHORE*. Thereafter, 4Test controls the value of the semaphore variable. You can acquire the semaphore if it has a value greater than

zero. When you have completed your semaphore-protected work, you release the semaphore. The `Acquire` function decrements the value of the semaphore by one and the `Release` function increments it by one. Thus, if you initialize the semaphore to 5, five threads can simultaneously execute semaphore-protected operations while a sixth thread has to wait until one of the five invokes the `Release` function for that semaphore.

The `Acquire` function either blocks the calling thread because the specified semaphore is zero, or "acquires" the semaphore by decrementing its value by one. `Release` checks for any threads blocked by the specified semaphore and unblocks the first blocked thread in the list. If no thread is blocked, `Release` "releases" the semaphore by incrementing its value by one so that the next invocation of `Acquire` succeeds, which means it does not block.

A call to `Acquire` has the following form:

```
void Acquire(SEMAPHORE semA)
```

Where `semA` s the semaphore variable to acquire.

A call to `Release` has the following form:

```
void Release(SEMAPHORE semA)
```

Where `semA` s the semaphore variable to release.

If more than one thread was suspended by a call to `Acquire`, the threads are released in the order in which they were suspended.

A semaphore that is assigned an initial value of 1 is called a binary semaphore, because it can only take on the values 0 or 1. A semaphore that is assigned an initial value of greater than one is called a counting semaphore because it is used to count a number of protected resources.

Example: Application only supports three simultaneous users

Suppose you are running a distributed test on eight machines using eight 4Test threads. Assume that the application you are testing accesses a database, but can support only three simultaneous users. The following code uses a semaphore to handle this situation:

```
SEMAPHORE DBUsers = 3
...
Acquire (DBUsers)
  access database
Release (DBUsers)
```

The declaration of the semaphore is global; each thread contains the code to acquire and release the semaphore. The initial value of three ensures that no more than three threads will ever be executing the database access code simultaneously.

Testing In Parallel but Not Synchronously

This topic illustrates a method for running test functions in parallel on multiple clients, but with different tests running on each client. This provides a realistic multi-user load as opposed to a load in which all clients perform the same operations at roughly the same time.

Example

This example suggests a method by which each client, operating in a separate thread, executes a test that is assigned by a random number. The `RandSeed` function is called first so that the random number sequence is the same for each iteration of this multi-user test scenario. This enables you to subsequently repeat the test with the same conditions.

The example reads a list of client machines from a file, `clients.txt`, and receives the test count as in input argument. These external variables make the example scalable as to the number of machines being tested and the number of tests to be run on each. The number of different testcases is twelve in this example, but could be changed by modifying the `SelectTest` function and adding further test functions. For each machine in the client machine list, the example spawns a thread in which the specified client executes a randomly selected test, repeating for the specified number of tests.

Note: You can execute this test as it is written because it sets its own application states. However, when you use multi-application support, this is automatic. And if you want to use this approach to drive different applications or to initialize a server before starting the testing, you must add multi-application support.

```
testcase ParallelRandomLoadTest (INTEGER iTestCount)
  LIST OF STRING lsClients
  RandSeed (3)

  // list of client names
  ListRead (lsClients, "clients.txt")

  STRING sClientName

  for each sClientName in lsClients
    spawn
      // Connect to client, which becomes current machine
      Connect (sClientName)
      SetAppState ("MyAppState")          // Initialize
application
      TestClient (iTestCount)
      Disconnect (sClientName)
    rendezvous

  TestClient (INTEGER iTestCount)
    for i = 1 to iTestCount
      SelectTest ()

  SelectTest ()
    INTEGER i = RandInt (1, 12)

    // This syntax invokes Test1 to Test12, based on i
    @("Test{i}") ()

  // Define the actual test functions
  Test1 ()
    // Do the test . . .

  Test2 ()
    // Do the test . . .
  . . .
  Test12 ()
    // Do the test . . .
```

Statement Types

This section describes the statement types that are available for managing distributed tests.

Parallel Processing Statements

You create and manage multiple threads using combinations of the 4Test statements parallel, spawn, rendezvous, and critical.

In 4Test, all running threads, which are those not blocked, have the same priority with respect to one another. 4Test executes one instruction for a thread, then passes control to the next thread. The first thread called is the first run, and so on.

All threads run to completion unless they are deadlocked. 4Test detects script deadlock and raises an exception.

Note: The 4Test exit statement terminates all threads immediately when it is executed by one thread.

Using Parallel Statements

A parallel statement spawns a statement for each machine specified and blocks the calling thread until the threads it spawns have all completed. It condenses the actions of spawn and rendezvous and can make code more readable.

The parallel statement executes a single statement for each thread. Thus if you want to run complete tests in parallel threads, use the invocation of a test function, which may execute many statements, with the parallel statement, or use a block of statements with spawn and rendezvous.

To use the parallel statement, you must specify the machines for which threads are to be started. You can follow the parallel keyword with a list of statements, each of which specifies a different Agent name. For example:

```
parallel
  DoSomething ("Client1")
  DoSomething ("Client2")
```

The `DoSomething` function then typically issues a `SetMachine(sMachine)` call to direct its machine operations to the proper Agent.

Using a Spawn Statement

A spawn statement begins execution of the specified statement or block of statements in a new thread. Since the purpose of spawn is to initiate concurrent test operations on multiple machines, the structure of a block of spawned code is typically:

- A `SetMachine` command, which directs subsequent machine operations to the specified agent.
- A set of machine operations to drive the application.
- A verification of the results of the machine operations.

You can use spawn to start a single thread for one machine, and then use successive spawn statements to start threads for other machines being tested. Silk Test Classic scans for all spawn statements preceding a rendezvous statement and starts all the threads at the same time. However, the typical use of spawn is in a loop, like the following:

```
for each sMachine in lsMachine
  spawn        // start thread for each sMachine
    SetMachine (sMachine)
    DoSomething ()
  rendezvous
```

The preceding example achieves the same result when written as follows:

```
for each sMachine in lsMachine
  spawn
    [sMachine]DoSomething ()
  rendezvous
```

To use a spawn statement in tests that use TrueLog, use the `OPT_PAUSE_TRUELOG` option to disable TrueLog. Otherwise, issuing a spawn statement when TrueLog is enabled causes Silk Test Classic to hang or crash.

Using Templates

This section describes how you can use templates for distributed testing.

Using the Parallel Template

This template is stored as `parallel.t` in the `Examples` subdirectory of the Silk Test Classic installation directory. The code tests a single application that runs on an externally defined set of machines.

This multi-test-case template accepts a list of machine names. The application whose main window is MyMainWin is invoked on each machine. The same operations are then performed on each machine in parallel. If any test case fails, the multi-test-case will be marked as having failed; however, a failed test case within a thread does not abort the thread.

You can use this template by doing three edits:

- Include the file that contains your window declarations.
- Substitute the MainWin name of your application, which is defined in your MainWin window declaration, with the Mainwin name of the template, `MyMainWin`.
- Insert the calls to one or more tests, or to the main function, where indicated.

Use `myframe.inc`.

```
use "myframe.inc"
multitestcase MyParallelTest (LIST of STRING lsMachines)

  STRING sMachine

  // Connect to all machines in parallel:
  for each sMachine in lsMachines
    spawn
      SetUpMachine (sMachine, MyMainWin)
    rendezvous

  // Set app state of each machine, invoking if necessary:
  SetMultiAppStates()

  // Run testcases in parallel
  for each sMachine in lsMachines
    spawn
      SetMachine (sMachine)
      // Call testcase(s) or call main()
    rendezvous
```

Client/Server Template

This template is stored as `multi_cs.t` in the `Examples` subdirectory of the Silk Test Classic installation directory. This test case invokes the server application and any number of client applications, based on the list of machines passed to it, and runs the same function on all clients concurrently, after which the server will perform end-of-session processing.

You can use this template by doing the following edits:

- Include the files that contain your window declarations for both the client application and the server application.
- Substitute the MainWin name of your server application, which is defined in your MainWin window declaration, with the MainWin name of the template, `MyServerApp`.
- Substitute the MainWin name of your client application, which is defined in your MainWin window declaration, with the Mainwin name of the template, `MyClientApp`.
- Replace the call to `PerformClientActivity` with a function that you have written to perform client operations and tests.

- Replace the call to `DoServerAdministration` with a function that you have written to perform server administrative processing and/or cleanup.

```
use "myframe.inc"
multitestcase MyClientServerTest (STRING sServer, LIST of STRING lsClients)
  STRING sClient

  // Connect to server machine:
  SetUpMachine (sServer, MyServerApp)

  // Connect to all client machines in parallel:
  for each sClient in lsClients
    spawn
      SetUpMachine (sClient, MyClientApp)
  rendezvous

  // Set app state of each machine, invoking if necessary:
  SetMultiAppStates()

  // Run functions in parallel on each client:
  for each sClient in lsClients
    spawn
      // Make client do some work:
      [sClient] PerformClientActivity()
    rendezvous

  // Perform end-of-session processing on server application:
  [sServer] DoServerAdministration()
```

Testing Multiple Machines

This section describes strategies for testing multiple machines.

Running Tests on One Remote Target

Use one of the following methods to specify that you want a script, suite, or test plan to run on a remote target instead of the host:

- Enter the name of the target Agent in the **Runtime Options** dialog box of the host. You also need to select a network protocol in the dialog box. If you have been testing a script by running Silk Test Classic and the Agent on the same system, you can then test the script on a remote system without editing your script by using this method.
- Specify the target Agent's name by enclosing it within brackets before the script or suite name. For example `[Ohio]myscript.t`.
- You can select `(none)` in the **Runtime Options** dialog box of the host and then specify the name of the target Agent in a call to the `Connect` function in your script. For example, to connect to a machine named Ontario:

  ```
  testcase MyTestcase ()
    Connect ("Ontario")
    // Call first testcase
    DoTest1 ()
    // Call second testcase
    DoTest2 ()
    Disconnect ("Ontario"
  ```

When you are driving only one remote target, there is no need to specify the current machine; all test case code is automatically directed to the only connected machine.

When you use the multi-application support functions, you do not have to make explicit calls to `Connect`; the support functions issue these calls for you.

Running Tests Serially on Multiple Targets

To run your scripts or suites serially on multiple target machines, specify the name of the target Agent within the suite file. For example, the following code runs a suite of three scripts serially on two target machines named Ohio and Montana:

```
[Ohio] script1.t
[Ohio] script2.t
[Ohio] script3.t
[Montana] script1.t
[Montana] script2.t
[Montana] script3.t
```

Any spaces between the name of the target Agent and the script name are not significant.

Alternatively, to run test cases serially on multiple target machines, switch among the target machines from within the script, by using the `Connect` and `Disconnect` functions of 4Test. For example, the following script contains a function named `DoSomeTesting` that is called once for each machine in a list of target machines, with the name of the target Agent as an argument:

```
testcase TestSerially ()
  STRING sMachine
  // Define list of agent names
  LIST OF STRING lsMachines = {...}
    "Ohio"
    "Montana"

  // Invoke test function for each name in list
  for each sMachine in lsMachines
    DoSomeTesting (sMachine)

  // Define the test function
  DoSomeTesting (STRING sMachine)
    Connect (sMachine)
    Print ("Target machine: {sMachine}")
    // do some testing...
    Disconnect (sMachine)
```

You will rarely need to run one test serially on multiple machines. Consider this example a step on the way to understanding parallel testing.

Specifying the Target Machine Driven By a Thread

While the typical purpose for a thread is to direct test operations to a particular test machine, you have total flexibility as to which machine is being driven by a particular thread at any point in time. For example, in the code below, the `spawn` statement starts a thread for each machine in a predefined list of test machines. The `SetMachine` command directs the code in that thread to the Agent on the specified machine. But the `["server"]` machine handle operator directs the code in the `doThis` function to the machine named *server*. The code following the `doThis` invocation continues to be sent to the *sMachine* specified in the `SetMachine` command.

```
for each smachine in lsMachine
  spawn // start thread for each sMachine
  SetMachine (sMachine)
  // ... code executed on sMachine
  ["server"]doThis() // code executed on "server"
  // ...continue with code for sMachine
rendezvous
```

While the machine handle operator takes only a machine handle, 4Test implicitly casts the string form of the Agent machine's name as a machine handle and so in the preceding example the machine name is effectively the same as a machine handle.

Specifying the Target Machine For a Single Command

To specify the target machine for a single command, use the machine handle operator on the command. For example, to execute the `SYS_SetDir` function on the target machine specified by the *sMachine1* variable, type `sMachine1->SYS_SetDir (sDir)`.

To allow you to conveniently perform system related functions (SYS_) on the host, you can preface the function call with the machine handle operator, specifying the globally defined constant *hHost* as the argument to the operator. For example, to set the working directory on the host machine to `c:\mydir`, type `hHost->SYS_SetDir ("c:\mydir")`.

You can use this syntax with a method call, for example `sMachine-> TextEditor.Search.Find.Pick`, but when invoking a method, this form of the machine handle must be the first token in the statement.

If you need to use this kind of statement, use the alternative form of the machine handle operator described below.

You can use the SetMachine function to change target machines for an entire block of code.

The hHost constant cannot be used in simple handle compares like hMyMachineHandle== hHost. This will never be TRUE. A better method is to use GetMachineName(hHost) and compare names. If hHost is used as an argument, it will refer to the "(local)" host not the target host.

Example

The following example shows valid and invalid syntax:

```
// Valid machine handle operator use
for each sMachine in lsMachine
  sMachine-> TextEditor.Search.Find.Pick

// Invalid machine handle operator use with method
if (sMachine->ProjX.DuplicateAlert.Exists())
  Print ("Duplicate warning on {sMachine} recipient.")
```

If you need to use this kind of statement, use the alternative form of the machine handle operator described below.

You can use the `SetMachine` function to change target machines for an entire block of code.

The *hHost* constant cannot be used in simple handle compares, like `hMyMachineHandle== hHost`. This will never be `TRUE`. A better method is to use `GetMachineName`(*hHost*) and compare names. If *hHost* is used as an argument, it will refer to the local host, not the target host.

Reporting Distributed Results

You can view test results in each of several formats, depending on the kind of information you need from the report. Each format sorts the results data differently, as follows:

Elapsed time	Sorts results for all threads and all machines in event order. This enables you to see the complete set of results for a time period and may give you a sense of the load on the server during that time period or may indicate a performance problem.
Machine	Sorts results for all threads running on one machine and presents the results in time-sorted order for that machine before reporting on the next machine.
Thread	Sorts results for all tests run under one thread and presents the results in time-sorted order for that thread before reporting on the next thread.

Alternative Machine Handle Operator

An alternative syntax for the machine handle operator is the bracket form, like the following example shows.

```
[hMachine] Any4TestFunctionCall ()
```

Example

To execute the `SYS_SetDir` function on the target machine specified by the string `sMachineA`, you do this:

```
[sMachineA] SYS_SetDir (sDir)
```

The correct form of the invalid syntax shown above is:

```
// Invalid machine handle operator use
if ([sMachine]ProjX.DuplicateAlert.Exists())
  Print ("Duplicate warning on {sMachine} recipient.")
```

To execute the `SYS_SetDir` function on the host machine, you can do the following:

```
[hHost] SYS_SetDir (sDir)
```

You can also use this form of the machine handle operator with a function that is not being used to return a value or with a method.

Example

```
for each sMachine in lsMachine
     [sMachine] FormatTest7 ()
```

Example

```
for each sMachine in lsMachine
  [sMachine] TextEditor.Search.Find.Pick
```

Testing Clients Concurrently

In concurrent testing, Silk Test Classic executes one function on two or more clients at the same time. This topic demonstrates one way to perform the same tests concurrently on multiple clients.

For example, suppose you want to initiate two concurrent database transactions on the same record, and then test how well the server performs. To accomplish this, you need to change the script presented in *Testing Clients Plus Server Serially* to look like this:

```
testcase TestConcurrently ()
  Connect ("server")
  Connect ("client1")
  Connect ("client2")
  DoSomeSetup ("server")    // initialize server first
  Disconnect ("server")     // testcase is thru with server

  spawn                     // start thread for client1
    UpdateDatabase ("client1")
  spawn                     // start thread for client2
    UpdateDatabase ("client2")

  rendezvous                // synchronize
  Disconnect ("client1")
  Disconnect ("client2")
```

```
  DoSomeSetup (STRING sMachine)   // define server setup
    HTIMER hTimer
    hTimer = TimerCreate ()
    TimerStart (hTimer)
    SetMachine (sMachine)
    // code to do server setup goes here
    TimerStop (hTimer)
    Print ("Time on {sMachine} is: {TimerStr (hTimer)}")
    TimerDestroy (hTimer)

  UpdateDatabase (STRING sMachine)   // define update test
    HTIMER hTimer
    hTimer = TimerCreate ()
    TimerStart (hTimer)
    SetMachine (sMachine)
    // code to update database goes here
    TimerStop (hTimer)
    Print ("Time on {sMachine} is: {TimerStr (hTimer)}")
    TimerDestroy (hTimer)
```

An alternative but equivalent approach is to use the parallel statement in place of the spawn and rendezvous:

```
testcase TestConcurrently2 ()
  Connect ("server")
  Connect ("client1")
  Connect ("client2")

  DoSomeSetup ("server")
  Disconnect ("server")

  parallel                      // automatic synchronization
    UpdateDatabase ("client1") // thread for client1
    UpdateDatabase ("client2") // thread for client2

  Disconnect ("client1")
  Disconnect ("client2")

  DoSomeSetup (STRING sMachine)
    HTIMER hTimer
    hTimer = TimerCreate ()
    TimerStart (hTimer)
    SetMachine (sMachine)
    // code to do server setup goes here
    TimerStop (hTimer)
    Print ("Time on {sMachine} is: {TimerStr (hTimer)}")
    TimerDestroy (hTimer)

  UpdateDatabase (STRING sMachine)
    HTIMER hTimer
    hTimer = TimerCreate ()
    TimerStart (hTimer)
    SetMachine (sMachine)
    // code to update database goes here
    TimerStop (hTimer)
    Print ("Time on {sMachine} is: {TimerStr (hTimer)}")
    TimerDestroy (hTimer)
```

If you use variables to specify different database records for each client's database transactions, you can use the above techniques to guarantee parallel execution without concurrent database accesses.

Testing Clients Plus Server Serially

In a client/server application, the server and its clients typically run on different target machines. This topic explains how to build tests that test the server and its clients in a serial fashion. In this scenario, the `SetMachine` function switches among the target machines on which the server and its clients are running. The following script fragment tests a client/server database application in the following steps:

1. Connect to three target machines, which are server, client1, and client2.
2. Call the `DoSomeSetup` function, which calls `SetMachine` to make "server" the current target machine, and then perform some setup.
3. Call the `UpdateDatabase` function once for each client machine. The function sets the target machine to the specified client, then does a database update. It creates a timer to time the operation on this client.
4. Disconnect from all target machines.

Example

This example shows how you direct sets of test case statements to particular machines. If you were doing functional testing of one application, you might want to drive the server first and then the application. However, this example is not realistic because it does not show the support necessary to bring the different machines to their different application states and to recover from a failure on any machine.

```
testcase TestClient_Server ()
  Connect ("server")
  Connect ("client1")
  Connect ("client2")
  DoSomeSetup ("server")
  UpdateDatabase ("client1")
  UpdateDatabase ("client2")
  DisconnectAll ()

DoSomeSetup (STRING sMachine)
  HTIMER hTimer
  hTimer = TimerCreate ()
  TimerStart (hTimer)
  SetMachine (sMachine)
  // code to do server setup goes here
  TimerStop (hTimer)
  Print ("Time on {sMachine} is: {TimerStr (hTimer)}")
  TimerDestroy (hTimer)

UpdateDatabase (STRING sMachine)
  HTIMER hTimer
  hTimer = TimerCreate ()
  TimerStart (hTimer)
  SetMachine (sMachine)
  // code to update database goes here
  TimerStop (hTimer)
  Print ("Time on {sMachine} is: {TimerStr (hTimer)}")
  TimerDestroy (hTimer)
```

Testing Databases

You may be testing a distributed application that accesses a database or you may be directly testing database software. In either of these cases, you might want to manipulate the database directly from Silk Test Classic for several purposes:

- To exercise certain database functions that are present in a GUI that runs directly on the server machine and is not a client application. For example, administrative functions used for setting up the database.
- To set the server database to a known state.
- To verify an application's database results without using the application.
- To read information from the database to use as input to a test case.

Silk Test Classic can drive a server application's GUI by means of the Silk Test Classic Agent exactly as it drives a client application. In addition, the database tester provides direct access, using SQL, from a test script to any database supported by ODBC drivers. These database functions enable you to read and write database records without using the client application. Thus, you can verify client test results without assuming the ability of the client to do that verification.

In addition to using the SQL functions in your tests, you can also use these functions to help manage your testing process as follows:

- Maintain a bug database, updating it with the results of your testing.
- Manage your test data in a database instead of in a text file.

The database functions, among other things, allow you to connect to a database, submit an SQL statement, read data from the selected record(s) if the SQL statement was SELECT, and subsequently disconnect from the database. About a dozen of these functions allow you to access your database's catalog tables.

The functions that support these operations begin with the letters "DB_".

Multi-Machine Testing in a Terminal Server Environment

This functionality is supported only if you are using the Classic Agent.

Note: This functionality is tested only for C++ applications. Whether any other extension will work is unconfirmed. If a non-C++ environment does not work, then it is still considered a non-supported environment and is not something that will be addressed by technical support.

Terminal Server

Terminal Server is an optional setup of Windows where the server is used in a similar fashion as a Unix server. In a network of this type you have a server with a lot of memory to serve many workstations. Each workstation has its own operating system and is connected through TCP/IP to the server machine. Each client is required to have only one program installed; the terminal client.

The terminal client is a program that displays a complete Windows desktop, including a taskbar, just like the one you see when you run Windows. Using the mouse and keyboard you are able to use this Desktop to start and use Windows Applications like Word, PowerPoint and Silk Test Classic. When these applications are running, they are not using the CPU or memory of the client machines, but are running on the server machines. The display of the desktop however is being set to the terminal client programs. They do this by sending many compressed images in cartoon fashion through TCP/IP from the server to the terminal emulators, which in turn display the images and make it appear as though the Desktop and the applications are running on the client machine.

Each terminal emulator has its own virtual mouse and keyboard port. You can have several different Terminal Clients running on the same machine and each window will have its own mouse pointer. When you use the physical mouse on the terminal client, the virtual mouse reacts to the commands.

Installing Silk Test Classic on the Controller Machine

In a Terminal Server environment the copy of Silk Test Classic is only installed on the server machine. Only one copy is needed. When installing the product, you must follow the Terminal Server instructions for installing the application in 'multi user' mode. This is done from the Add/Remove Programs feature found in the Control Panel. When this is complete, you can start Silk Test Classic from a terminal client and create and run tests as you do when Silk Test Classic is installed on one machine.

Setting Up the Terminal Server Clients

In order to have multiple terminal clients running Silk Test Classic or other Agents, some configuration must take place. If you want to be able to run test scenarios in different terminal client windows from a single point of control, each client needs to start its own Agent process. Each Agent must be configured for network use. You can either use the NetBios protocol and give each Agent instance a separate name, or use the TCP/IP protocol and assign each Agent instance a different port.

Silk Test Classic does not support multiple-user sessions for a single target system. Using standalone Silk Test Classic you can only service one Agent on a particular machine. However, using Silk Performer as controller for Silk Test Classic scripts, you can also service multiple user/Agent sessions on a single system (Silk Performer Gui level testing).

Starting Agents

Silk Test Classic has a -p option for `Agent.exe` that lets you specify a port number for the Agent. Therefore, it is recommended that you use TCP/IP protocol and start Agents using `agent -p <unique port number>`.

Testing Multiple Applications

This section describes testing multiple applications.

Overview of Multi-Application Testing

Silk Test Classic can easily drive multiple different applications simultaneously. Thus you can bring a server's database to a known state at the same time you are bringing multiple instances of the client application to their base state window. Likewise, you can drive a server database with several different client applications at the same time.

The essential difference between single-application and multi-application testing is clearly the difference between "one" and "many." When the following entities in a test case are greater than one, they need special consideration and support functions found in Silk Test Classic:

- Agent names.
- Application main window names.
- Sets of application states associated with each main window name.

Multi-machine testing requires that you map both the name of an application and all application states for that application to the machine on which it will be tested. This makes it possible for you to direct test operations to the right machines, and it enables Silk Test Classic to automatically set the machines to the proper application state before a test is run, and to clean up after a test has failed.

Test Case Structure in a Multi-Application Environment

This topic describes Silk Test Classic components that enable concurrent testing of more than one application. For example, there are functions that make it possible to drive both the client application and the client's server from Silk Test Classic, to set each to its base state, and to recover each if it fails. Compare with the test case structure of a single-application environment.

The multi-application environment uses the same `defaults.inc` file as does the single-application environment. However, when you define a function as a multitestcase, 4Test uses functions defined in the `cs.inc` file to invoke functions in `defaults.inc`. Thus, it can pass the appropriate application states or base states to these functions, depending on the requirements of a particular test machine.

Instead of preceding the test case function declaration with the keyword testcase, you must use the keyword multitestcase to give your test case the multi-application recovery system.

`cs.inc` is an automatically included file that contains functions used only in the multi-application environment. For additional information about this file and the functions that it contains, see *cs.inc*. You may need to include other files also.

Invoking a Test Case in a Multi-Application Environment

The keyword for a test case declaration is different when you are performing distributed testing. In the single-application environment, you invoke a test case with no arguments or you may specify an application state function. However, in a multi-application environment, instead of preceding the test case function declaration with the keyword testcase, you must use the keyword multitestcase to give your test case the multi-application recovery system.

Declaring a function as a multitestcase gives that function the ability to invoke functions declared with the keyword testcase. A multitestcase thus can be viewed as a wrapper for stand-alone test cases; it provides a means of assigning tests to particular machines and lets you invoke previously written test cases from the multi-test case file by simply adding a use statement to the file to include the test case definitions.

When you are using multi-application environment support, you can pass the test case the names of the machines to be tested during that execution of the test case, but not the application state function. In a multi-application environment, one test case can use multiple application states; you specify these in the required code at the beginning of the test case.

Test Case Structure in a Single-Application Environment

The code that implements a test case for a single application is similar to that of a test case for applications on multiple separate machines in a client/server environment.

This topic summarizes the structure of the single-application version and some Silk Test Classic components used to implement it. You can compare the structure with the support code needed for running multiple applications.

The include file `defaults.inc` implements the recovery system for a single application test. For information about the `DefaulBaseState` function and the functions that are contained within `defaults.inc`, see *defaults.inc*.

Your test case needs certain definitions that other test cases in your testing program will also need. These include:

- Window declarations
- Application states
- Utility functions

Placing these general purpose definitions in an include file, or several smaller files, saves repetitive coding. When you use Silk Test Classic to record window declarations and application states, Silk Test Classic names the generated file `frame.inc`.

Window Declarations for Multi-Application Testing

In the client/server environment, unlike the stand-alone environment, you can test two or more different applications at the same time. For example, you could run the functional tests for application "A" on multiple machines at the same time that you are running the functional tests for application "B" on the same machines. The include files that you must generate may therefore have to take into consideration different platforms and/or different applications.

When you are driving two or more applications from Silk Test Classic, you need separate window declarations for each different application. You must be certain that your main window declaration for each separate application is unique. If the same application is running on different platforms concurrently, you may need to use GUI specifiers to specialize the window declarations. 4Test will identify a window declaration statement, that is preceded by a GUI specifier, as being true only on the specified GUI.

In addition, you may find that the operations needed to establish a particular application state are slightly different between platforms. In this case, you just record application states for each platform and give them names that identify the state and the GUI for your convenience.

Recording window declarations on a client machine that is not the host machine, requires that you operate both Silk Test Classic on the host machine and the application on its machine at the same time. You record

window declarations and application states in much the same way for a remote machine as for an application running in the Silk Test Classic host machine. The primary difference is that you start the recording operation by selecting Test Frame in Silk Test Classic on the host system and you do the actual recording of application operations on the remote system.

If you have two or more applications being tested in parallel, you need to have two or more sets of window declarations. You must have window declarations, and application states, if needed, for each different application. When recording window declarations and application states on a remote machine, you will find it convenient to have the machine physically near to your host system.

Remote Recording

This functionality is supported only if you are using the Classic Agent.

Concurrency Test Example Code

The concurrency test example is designed to allow any number of test machines to attempt to access a server database at the same time. This tests for problems with concurrency, such as deadlock or out-of-sequence writes.

This example uses only one application. However, it is coded in the style required by the multi-application environment because you will probably want to use an Agent to start and initialize the server during this type of test. There is no requirement in the client/server environment that you use the single-application style of test case just because you are driving only one application. For consistency of coding style, you will probably find it convenient to always use the multi-application files and functions.

For detailed information on the code example, see *Concurrency Test Explained*.

```
const ACCEPT_TIMEOUT = 15
multitestcase MyTest (LIST OF STRING lsMachine)
  STRING sMachine
  INTEGER iSucceed
  STRING sError

  for each sMachine in lsMachine
    SetUpMachine (sMachine, Personnel)
    SetMultiAppStates ()

  /*** HAVE EACH MACHINE EDIT THE SAME EMPLOYEE ***/
  for each sMachine in lsMachine
    spawn

      /*** SET THE CURRENT MACHINE FOR THIS THREAD ***/
      SetMachine (sMachine)

      /*** EDIT THE EMPLOYEE RECORD "John Doe" ***/
      Personnel.EmployeeList.Select ("John Doe")
      Personnel.Employee.Edit.Pick ()

      /*** CHANGE THE SALARY TO A RANDOM NUMBER BETWEEN
      50000 AND 70000 ***/
      Employee.Salary.SetText ([STRING] RandInt (50000, 70000))
  rendezvous

  /*** ATTEMPT TO HAVE EACH MACHINE SAVE THE EMPLOYEE RECORD ***/
  for each sMachine in lsMachine
    spawn

      /*** SET THE CURRENT MACHINE FOR THIS THREAD ***/
      SetMachine (sMachine)

      /*** SELECT THE OK BUTTON ***/
      Employee.OK.Click ()
```

```
      /*** CHECK IF THERE IS A MESSAGE BOX ***/
      if (MessageBox.Exists (ACCEPT_TIMEOUT))
        SetMachineData (NULL, "sMessage",
          MessageBox.Message.GetText ())
        MessageBox.OK.Click ()
        Employee.Cancel.Click ()
      else if (Employee.Exists ())
        AppError ("Employee dialog not
          dismissed after {ACCEPT_TIMEOUT} seconds")
  rendezvous

  /*** VERIFY THE OF NUMBER OF MACHINES WHICH SUCCEEDED ***/
  iSucceed = 0
  for each sMachine in lsMachine
    sError = GetMachineData (sMachine, "sMessage")
    if (sMessage == NULL)
      iSucceed += 1
    else
      Print ("Machine {sMachine} got message '{sMessage}'")

  Verify (iSucceed, 1, "number of machines that succeeded")
```

Concurrency Test Explained

Before you record and/or code your concurrency test, you record window declarations that describe the elements of the application's GUI. These are placed in a file named `frame.inc`, which is automatically included with your test case when you compile. Use Silk Test Classic to generate this file because Silk Test Classic does most of the work.

The following code sample gives just those window declarations that are used in the *Concurrency Test Example*:

```
window MainWin Personnel
  tag "Personnel"
  PopupList EmployeeList
  Menu Employee
    tag "Employee"
  MenuItem Edit
    tag "Edit"
  // ...

window DialogBox Employee
  tag "Employee"
  parent Personnel
  TextField Salary
    tag "Salary"
  PushButton OK
    tag "OK"
  // ...
```

The following explanation of the *Concurrency Test Example* gives the testing paradigm for a simple concurrency test and provides explanations of many of the code constructs. This should enable you to read the example without referring to the Help. There you will find more detailed explanations of these language constructs, plus explanations of the constructs not explained here. The explanation of each piece of code follows that code.

```
const ACCEPT_TIMEOUT = 15
```

The first line of the testcase file defines the timeout value (in seconds) to be used while waiting for a window to display.

```
multitestcase MyTest (LIST OF STRING lsMachine)
```

The test case function declaration starts with the multitestcase keyword. It specifies a LIST OF STRING argument that contains the machine names for the set of client machines to be tested. You can implement and maintain this list in your test plan, by using the test plan editor. The machine names you use in this list are the names of the Agents of your target machines.

```
for each sMachine in lsMachine
  SetUpMachine (sMachine, Personnel)
```

To prepare your client machines for testing, you must connect Silk Test Classic to each Agent and, by means of the Agent, bring up the application on each machine. In this example, all Agents are running the same software and so all have the same MainWin declaration and therefore just one test frame file. This means you can initialize all your machines the same way; for each machine being tested, you pass to SetUpMachine the main window name you specified in your test frame file. The SetUpMachine function issues a Connect call for each machine. It associates the main window name you specified (Personnel) with each machine so that the DefaultBaseState function can subsequently retrieve it.

```
SetMultiAppStates ()
```

The SetMultiAppStates function reads the information associated with each machine to determine whether the machine needs to be set to an application state. In this case no application state was specified (it would have been a third argument for SetUpMachine). Therefore, SetMultiAppStates calls the DefaultBaseState function for each machine. In this example, DefaultBaseState drives the Agent for each machine to open the main window of the Personnel application. This application is then active on the client machine and 4Test can send test case statements to the Agent to drive application operations.

```
 for each sMachine in lsMachine
  spawn
    // The code to be executed in parallel by
    // all machines... (described below)
rendezvous
```

Because this is a concurrency test, you want all client applications to execute the test at exactly the same time. The spawn statement starts an execution thread in which each statement in the indented code block runs in parallel with all currently running threads. In this example, a thread is started for each machine in the list of machines being tested. 4Test sends the statements in the indented code block to the Agents on each machine and then waits at the rendezvous statement until all Agents report that all the code statements have been executed.

The following is the code defined for the spawn statement:

```
// The code to be executed in parallel by
// all machines:
SetMachine (sMachine)
Personnel.EmployeeList.Select ("John Doe")
Personnel.Employee.Edit.Pick ()
Employee.Salary.SetText
[STRING] RandInt (50000, 70000))
```

Each thread executes operations that prepare for an attempt to perform concurrent writes to the same database record. The SetMachine function establishes the Agent that is to execute the code in this thread. The next two statements drive the application's user interface to select John Doe's record from the employee list box and then to pick the **Edit** option from the **Employee** menu. This opens the **Employee** dialog box and displays John Doe's employee record. The last thread operation sets the salary field in this dialog box to a random number. At this point the client is prepared to attempt a write to John Doe's employee record. When this point has been reached by all clients, the rendezvous statement is satisfied, and 4Test can continue with the next script statement.

```
for each sMachine in lsMachine
  spawn
    SetMachine (sMachine)
    Employee.OK.Click ()
    if (MessageBox.Exists (ACCEPT_TIMEOUT))
      SetMachineData (NULL, "sMessage",
      MessageBox.Message.GetText ())
```

```
      MessageBox.OK.Click ()
      Employee.Cancel.Click ()
    else if (Employee.Exists ())
      AppError ("Employee dialog not dismissed
        after {ACCEPT_TIMEOUT} seconds")
rendezvous
```

Now that all the clients are ready to write to the database, the script creates a thread for each client, in which each attempts to save the same employee record at the same time. There is only one operation for each Agent to execute: `Employee.OK.Click`, which clicks the **OK** button to commit the edit performed in the previous thread.

The test expects the application to report the concurrency conflict with message boxes for all but one client and for that client to close its dialog box within 15 seconds. The if-else construct saves the text of the message in the error message box by means of the SetMachineData function. It then closes the message box and the **Employee** window so that the recovery system will not report that it had to close windows. This is good practice because it means fewer messages to interpret in the results file.

The "else if" section of the if-else checks to see whether the **Employee** window remains open, presumably because it is held by a deadlock condition; this is a test case failure. In this case, the AppError function places the string "***ERROR:" in front of the descriptive error message and raises an exception; all Agents terminate their threads and the test case exits.

```
iSucceed = 0
for each sMachine in lsMachine
  sMessage = GetMachineData (sMachine, "sMessage")
  if (sMessage == NULL)
    iSucceed += 1
  else
    Print ("Machine {sMachine} got message '{sMessage}'")
Verify (iSucceed, 1, "number of machines that succeeded")
```

The last section of code evaluates the results of the concurrency test in the event that all threads completed. If more than one client successfully wrote to the database, the test actually failed.

GetMachineData retrieves the message box message (if any) associated with each machine. If there was no message, iSucceed is incremented; it holds the count of "successes." The Print function writes the text of the message box to the results file for each machine that had a message box. You can read the results file to verify that the correct message was reported. Alternatively, you could modify the test to automatically verify the message text.

The `Verify` function verifies that one and only one machine succeeded. If the comparison in the Verify function fails, Verify raises an exception. All exceptions are listed in the results file.

Notification Test Example Code (1 of 2)

This functionality is supported only if you are using the Classic Agent.

This topic contains the complete test case file for a single-user notification test. It shows a testing technique for a type of communication frequently used in client/server applications. *Notification Test Example Code (2 of 2)* shows a notification test between two users running their own copies of the client application. This illustrates doing the simplest case first and then adding the next level of complexity when you go from one user to two users. For additional information on the testing technique, see *Notification Test Example Explained (1 of 2)*.

```
// ccmail.t
use "ccmail.inc"
LogMeIn()
  LogInUser(GetMachineData( NULL, "Username" ),
    GetMachineData( NULL, "Password" ) )

//-------------------------------------------------------------
multitestcase SingleUserNotification ( STRING sMachine1 optional )
```

```
  if( sMachine1 == NULL )
    sMachine1 = "(local)"

  //=== MULTI-APPLICATION SETUP SECTION ==================//
  SetUpMachine( sMachine1, CcMail, "EnsureInBoxIsEmpty" )
  SetMachineData( sMachine1, "Username", "QAtest1" )
  SetMachineData( sMachine1, "Password", "QAtest1" )
  SetMultiAppStates()

  //=== TEST BEGINS HERE ================================//
  SetMachine( sMachine1 )
  SimpleMessage( "QAtest1", "Message to myself", "A message to myself" )
  Verify( CcMailNewMailAlert.Exists( NOTIFICATION_TIMEOUT ), TRUE )
  Verify( CcMailNewMailAlert.IsActive(), TRUE, "ALERT" )
  CcMailNewMailAlert.OK.Click()
  CcMail.xWindow.GoToInbox.Pick ()
  Verify( CcMail.Message.DeleteMessage.IsEnabled(), TRUE,
    "MESSAGE WAITING" )
```

Utility function

```
void SimpleMessage (STRING sRecipient, STRING sSubject,
  STRING sBody)
CcMail.Message.NewMessage.Pick()

NewMessage.MailingLabel.Recipient.SetText (sRecipient)
NewMessage.MailingLabel.Recipient.TypeKeys ("<Enter>")
NewMessage.MailingLabel.Recipient.TypeKeys ("<Enter>")
NewMessage.MailingLabel.SubjectField.SetText (sSubject)
NewMessage.MailingLabel.SubjectField.TypeKeys ("<Enter>")
NewMessage.EditBody.Body.TypeKeys (sBody)
NewMessage.EditBody.Body.TypeKeys ("<Ctrl-s>")
```

This function uses standard methods on Ccmail window components, defined in `ccmail.inc`, to do the following:

1. Pick the NewMessage item from the Message menu.
2. Enter the string in argument one into the Recipient field and press the **Enter** key twice to move to the Subject field.
3. Enter the string in argument two into the Subject field and press **Enter** to move to the message body portion of the window (EditBody.Body).
4. Type the string in argument three into the Body field and type **Ctrl + s** to send the message.

The following block of code verifies the results of the test.

```
Verify(CcMailNewMailAlert.Exists(NOTIFICATION_TIMEOUT),
  TRUE )
Verify(CcMailNewMailAlert.IsActive(), TRUE, "ALERT")
CcMailNewMailAlert.OK.Click()
CcMail.xWindow.GoToInbox.Pick ()
Verify(CcMail.Message.DeleteMessage.IsEnabled(), TRUE,
  "MESSAGE WAITING")
```

The above code does the following:

1. Verifies that the sent message was received, as indicated by the **NewMailAlert** message box. The `NOTIFICATION_TIMEOUT` value causes the `Verify` function to wait for that period of time for the window to exist. If the timeout value is reached, the `Verify` raises an exception.
2. Verify that the dialog box **CcMailNewMailAlert** is active.
3. If the `Verify` executes without an exception, click on the **OK** button in the **CcMailNewMailAlert** dialog box.
4. Pick the **GoToInbox** menu item from the Window menu.

5. Verify that a message exists in the Inbox by checking to see that the Message menu has its **DeleteMessage** menu item enabled. If the menu item is not enabled, there is no message in the Inbox and the `Verify` function raises an exception.
- This script continues in *Notification Test Example Code (2 of 2)*.

Notification Test Example Explained (1 of 2)

This functionality is supported only if you are using the Classic Agent.

The first line in the test case file is a comment that lists the name of the file holding this code.

```
// csmail.t
```

The next line is an include statement. The explanations for each fragment of code follow that code.

```
use "ccmail.inc"
```

The `ccmail.inc` file is defined for this test case. It contains the window declarations for the application in addition to application state definitions and definitions for general-purpose utility functions also needed by other test cases designed for this application. You can find the `ccmail.inc` file in the Silk Test Classic Examples directory. Code fragments from that file are included as needed in this discussion.

```
LogMeIn()
  LogInUser(GetMachineData(NULL, "Username"),
    GetMachineData(NULL, "Password") )
```

The utility function LogMeIn is called by the invoke method for the CC Mail main window, called `CcMail`. The LogInUser function is defined in `ccmail.inc`. The machine data that LogInUser retrieves for its arguments gets established by each test before the application state function for each machine is invoked.

```
multitestcase SingleUserNotification (STRING sMachine1)
```

The function declaration for the test case passes in the name of the Agent for the machine on which the application is running.

```
if(sMachine1 == NULL)
  sMachine1 = "(local)"
```

This if statement if statement allows you to invoke the test case without specifying a machine name when you want to run on the local machine.

```
SetUpMachine(sMachine1, CcMail, "EnsureInBoxIsEmpty")
```

The SetUpMachine function provides the name of the main application window, `CcMail`, and the application state (`EnsureInBoxIsEmpty`) to be established by Silk Test Classic. `EnsureInBoxIsEmpty` is defined in `ccmail.inc`. This statement is part of the standard setup code for multi-application tests. The standard multi-application setup code is documented in *template.t Explained* and *Concurrency Test Example Code*. The setup code in this test case is essentially the same.

This is a single-user test case and therefore does not actually need the setup methodology required by a multi-application test. However, since client/server testing is frequently multi-application testing, all the example test cases use the multi-application coding methods. We recommend that you also follow this practice, since consistency of testing styles reduces coding errors in your test cases.

One difference for this test case is that this is an application that requires the user to log in. Therefore the following code fragment provides the user name and password for the application under test:

```
SetMachineData (sMachine1, "Username", "QAtest1")
SetMachineData (sMachine1, "Password", "QAtest1")
```

These statements associate two pieces of information, named "Username" and "Password," with the specified machine. In both cases the value of the associated information is the same, "QAtest1." Now that this information is available to the application state function, that function can log the user in. This will happen as a result of the next statement.

```
SetMultiAppStates()
```

In this test, `SetMultiAppStates` function will actually only set the application state for the one machine.

```
SimpleMessage ("QAtest1", "Message to myself",
  "A message to myself")
```

The above line invokes the utility function from `ccmail.inc`, which sends the short message to the local machine.

Notification Test Example Code (2 of 2)

This functionality is supported only if you are using the Classic Agent.

This is the complete test case file for a two-user notification test. It shows the next level of complexity in testing client/server notification operations. For additional information on the testing technique, see *Notification Example 2 Explained*.

```
//-------------------------------------------------------------
// This testcase logs in as user QAtest1 on the first machine,
// and logs in as user QAtest2 on the second machine; then
// sends a message from the user on the first machine to the
// user on the second machine; it then switches to the second
// machine and waits to be notified that new mail has arrived.
//

multitestcase TwoUserNotification ( STRING sMachine1, STRING sMachine2 )

  //=== MULTI-APPLICATION SETUP SECTION ==================//
  SetUpMachine( sMachine1, CcMail )
  SetUpMachine( sMachine2, CcMail, "EnsureInBoxIsEmpty" )
  SetMachineData( sMachine1, "Username", "QAtest1" )
  SetMachineData( sMachine1, "Password", "QAtest1" )
  SetMachineData( sMachine2, "Username", "QAtest2" )
  SetMachineData( sMachine2, "Password", "QAtest2" )
  SetMultiAppStates()

  //=== TEST BEGINS HERE ================================//
  //-------------------------------------------------------
  // Switch to the first machine:
  SetMachine( sMachine1 )

  // Send mail from user 1 to user 2
  SimpleMessage("QAtest2", "Message to user 2", "Message from me to you.")

  //-------------------------------------------------------
  // Switch to the second machine:
  SetMachine( sMachine2 )

  // Wait for notification to occur, then acknowledge it:
  Verify( CcMailNewMailAlert.Exists( NOTIFICATION_TIMEOUT ), TRUE )
  Verify( CcMailNewMailAlert.IsActive(), TRUE, "ALERT" )
  CcMailNewMailAlert.OK.Click()

  // Refresh the In box and check that a message is waiting there:
  CcMail.xWindow.GoToInbox.Pick ()
  Verify( CcMail.Message.DeleteMessage.IsEnabled(), TRUE,
    "MESSAGE WAITING" )
```

Notification Test Example Explained (2 of 2)

This functionality is supported only if you are using the Classic Agent.

The code in this two-user notification test is much the same as the code in the single-user example, except that the test is distributed across two CcMail applications. Thus the primary differences in this example are in the program flow.

Note: The described actions are carried out sequentially rather than concurrently.

The following actions are carried out by the code in the two-user notification test:

Before the test starts

1. The SetUpMachine function is carried out on two machines; the first machine defaults to the base state, but the second machine specifies an application state that ensures that its InBox is empty.
2. The Username and Password values for both machines are set.
3. The SetMultiAppStates function is invoked for both machines.

Note: This function will set different application states for the two machines.

On Machine 1:

1. The SetMachine function specifies that Machine 1 should receive the next operation.
2. A simple message is sent to Machine 2.

On Machine 2:

1. Verify that the **Alert** dialog box exists.
2. Verify that the **Alert** dialog box is active. If it is not, the exception's error message will be `Verify ALERT failed...`.
3. If the **Alert** dialog box has opened, dismiss it by clicking **OK**.
4. Refresh the InBox by picking the Inbox choice from CcMail's Window menu.
5. Verify that the Message menu's DeleteMessage menu item is enabled, proving that the message is in the Inbox. If Verify fails, which means the menu item is not enabled, the exception's error message will read, `Verify MESSAGE WAITING failed...`.

Code for template.t

This fragment of an example test case shows the required code with which you start a multi-application test case. It connects Silk Test Classic to all the machines being tested and brings each to its first screen. This is just a template; you must tailor your code to fit your actual needs. For information on the significance of each line of code, see *Template.t Explained*.

```
multitestcase MyTest (STRING sMach1, STRING sMach2)
  SetUpMachine (sMach1, MyFirstApp, "MyFirstAppState")
  SetUpMachine (sMach2, MySecondApp, "MySecondAppState")
  SetMultiAppStates ()
    spawn
      SetMachine (sMach1)
      // Here is placed code that drives test operations

    spawn
      SetMachine (sMach2)
      // Here is placed code that drives test operations

  rendezvous
  // "..."
```

template.t Explained

The following line of code in *Code for template.t* is the first required line in a multi-application test case file. It is the test case declaration.

Note: The code does not pass an application state as in the stand-alone environment.

```
multitestcase MyTest (STRING sMach1, STRING sMach2)
```

In the multi-application environment the arguments to your test case are names of the machines to be tested; you specify application states inside the test case. You can code the machine names arguments as you like. For example, you can pass a file name as the only argument, and then, in the test case, read the names of the machines from that file. Or you can define a LIST OF HMACHINE data structure in your test plan, if you are using the test plan editor, to specify the required machines and pass the name of the list, when you invoke the test case from the test plan. This template assumes that you are using a test plan and that it passes the Agent names when it invokes the test case. For this example, the test plan might specify the following:

```
Mytest ("Client1", "Client2")
```

The next two code lines are the first required lines in the test case:

```
SetUpMachine (sMach1, MyFirstApp, "MyFirstAppState")
SetUpMachine (sMach2, My2ndApp, "My2ndAppState")
```

You must execute the `SetUpMachine` function for every client machine that will be tested. For each `SetUpMachine` call, you specify the application to be tested, by passing the name of the main window, and the state to which you want the application to be set, by passing the name of the application state if you have defined one.

The `SetUpMachine` function issues a `Connect` call for a machine you want to test and then configures either the base state or a specified application state.

It does this as follows:

- It associates the client application's main window name with the specified machine so that the `DefaultBaseState` function can subsequently retrieve it to set the base state.
- It associates the name of the application's base state, if one is specified, with the specified machine so that the `SetMultiAppStates` function can subsequently retrieve it and set the application to that state at the start of the test case.

The first argument for `SetUpMachine` is the machine name of one of your client machines. The second argument is the name you supply in your main window declaration in your test frame file, `frame.inc`. For this example, the `frame.inc` file specifies the following:

```
window MainWin MyFirstApp
```

Because this template specifies two different applications, it requires two different test frame files.

The third argument is the name you provide for your application state function in your appstate declaration for this test. For this example, the appstate declaration is the following:

```
appstate MyFirstAppState () based on MyFirstBaseState
```

The appstate declaration could also be of the form:

```
appstate MyFirstBaseState ()
```

Although the `DefaultBaseState` function is designed to handle most types of GUI-based applications, you may find that you need to define your own base state. It would be the application state that all your tests for this application use. In this case, you would still pass this application state to `SetUpMachine` so that your application would always be brought to this state at the start of each test case.

This template specifies two application states for generality. You would not use an application state if you wanted to start from the main window each time. If you have a number of tests that require you to bring the application to the same state, it saves test-case code to record the application state once, and pass its name to `SetUpMachine`. You will probably place your application state declarations in your test frame file.

```
SetMultiAppStates ()
```

The `SetMultiAppStates` function must always be called, even if the test case specifies no application state, because `SetMultiAppStates` calls the `DefaultBaseState` function in the absence of an

appstate declaration. `SetMultiAppStates` uses the information that `SetUpMachine` associated with each connected machine to set potentially different application states or base states for each machine.

```
 spawn
  SetMachine (sMach1)
  // Here is placed code that drives test operations
```

The spawn statement starts an execution thread, in which each statement in the indented code block below it runs in parallel with all currently running threads. There is no requirement that your test case should drive all your test machines at the same time, however, this is usually the case. The `SetMachine` function directs 4Test to execute this thread's code by means of the Agent on the specified machine. This thread can then go on to drive a portion, or all, of the test operations for this machine.

```
 spawn
  SetMachine (sMach2)
  // Here is placed code that drives test operations
rendezvous
// "..."
```

The second spawn statement starts the thread for the second machine in this template. The rendezvous statement blocks the execution of the calling thread until all threads spawned have completed. You can use the rendezvous statement to synchronize machines as necessary before continuing with the test case.

defaults.inc

The `defaults.inc` file is provided by Silk Test Classic and implements the recovery system for a single application test. That is, it contains the `DefaultBaseState` function that performs any cleanup needed after an operation under test fails and returns the application to its base state.

You can define a base state function to replace the `DefaultBaseState` function by defining an application state without using the basedon keyword. This creates an application state that 4Test executes instead of the `DefaultBaseState` function.

The `defaults.inc` file contains six other functions that 4Test automatically executes unless you define functions that replace them:

DefaultScriptEnter	A null function, allows you to define a `ScriptEnter` function, as discussed below.
DefaultScriptExit (BOOLEAN bException)	Logs an exception to the results file when a script exits because of an exception.
DefaultTestcaseEnter	Executes the `SetAppState` function. If you have specified an application state for this test case, the `SetAppState` function brings your test application to that state. If you have no application state defined, `SetAppState` brings the application to the base state (if necessary).
DefaultTestcaseExit (BOOLEAN bException)	Logs an exception to the results file when a test case exits because of an exception. The function then executes the `SetBaseState` function, which calls a base state function that you have defined or the `DefaultBaseState` function.
DefaultTestPlanEnter	A null function, allows you to define `TestPlanEnter`, as discussed below, to allow logging of results.
DefaultTestPlanExit (BOOLEAN bException)	A null function, allows you to define `TestPlanExit`, as discussed below, to allow logging of results.

The word "Default" in each of the above function names signifies that you can define alternative functions to replace these. If, for example, you define a function called TestcaseEnter, 4Test will invoke your function before executing any of the code in your test case and will not invoke `DefaultTestcaseEnter`.

`TestPlanEnter()` is not called until the first test case in the plan is run. Or the first marked test case, if you are only running marked test cases. Similarly, `TestPlanExit()` is called at the completion of the last marked test case. `TestPlanExit()` is only called if the last marked test description contains an executable test case, which means not a manual test case or a commented out test case specifier.

cs.inc

`cs.inc` is an automatically included file that contains functions used only in the multi-application environment. The following functions provide a recovery system for managing automated testing of client/server applications:

SetMultiAppStates	Sets an application state for each connected machine, if the "AppState" machine data lists one; if not, it calls the `DefaultBaseState` function, which sets the application to its main window.
SetMultiBaseStates	Sets the application to the lowest state in the application state hierarchy for each connected machine, if the "AppState" machine data lists an application state. The lowest application state is one in which the appstate declaration did not use the basedon keyword. If there is no "AppState" information associated with this machine, `SetMultiBaseStates` calls the `DefaultBaseState` function, which sets the application to its main window, invoking it beforehand if necessary.
SetUpMachine	Connects Silk Test Classic to an agent on the specified machine. It provides a way to associate a main window declaration and an application state function with a machine name. These parameters are stored as data accessible by means of the `GetMachineData` function. Both of these names (the second and third arguments to the function) are optional; however, if you omit both arguments, you will have no recovery system.
DefaultMultiTestCaseEnter	Executes at the beginning of a multi-test case. It invokes a `DisconnectAll` function. The invocation of the `SetAppState` function is performed by the `SetMultiAppStates` function because the `DefaultTestCaseEnter` function is not executed for a multi-test case.
DefaultMultiTestCaseExit	Executes just before a multi-test case terminates. It logs any pending exception, then invokes `SetMultiBaseStates` and `DisconnectAll`.

Include File Size

The maximum size of an include file is approximately 65536 lines. If your include file is very large, split it into two files and continue with your testing.

Troubleshooting Distributed Testing

This section provides troubleshooting information for testing on multiple machines.

Handling Limited Licenses

By default, Silk Test Classic starts up an unplanned Agent on the local workstation. If you do not want to use the local workstation as a test machine, set the **Agent Name** field in the **Runtime Options** dialog box to `(none)` instead of `(local)`. This will free up one license for a remote Agent.

Resolving Port-Number Conflicts

This functionality is supported only if you are using the Classic Agent.

Silk Test Classic connects to each Agent through a TCP/IP port that has a 4-digit or 5-digit ID. Typically, all the machines in your testing network automatically use the same default port number. This allows the

`Connect` function to automatically specify the port number for all connections. If some other application on one of your machines has already used the default port number, you have a port number conflict.

If you start an Agent on a PC and the default port is already in use, an error message is displayed.

In either case, you can use a different port number just for this machine, while using the default number for the rest, or you can have all your machines use the same available port number. When you have an Agent that uses a port number which is different than the default port number, you must specify the port number in every reference to that Agent. The syntax is `AgentName:nnnn` where `AgentName` is the target machine name and `nnnn` is the port number. Since you typically use a file or a list variable to hold your Agent names, you can add the `:nnnn` where needed.

If there are no port conflicts, you do not have to specify ports. If you have a conflict, the port number used for that machine must change. You can choose to change the port numbers used by all your PCs and workstations so that all use the same number.

Setting-Up Extensions for Distributed Testing

This functionality is supported only if you are using the Classic Agent.

If you are testing non-Web applications, you must disable browser extensions on your host machine. This is because the recovery system works differently when testing Web applications than when testing non-Web applications.

Furthermore, when you select one or both of the Internet Explorer (IE) extensions on the host machine's **Extension** dialog box, Silk Test Classic automatically selects the correct version of the host machine's IE application in the **Runtime Options** dialog box. If the target machine's version of IE is not the same as the host machine's, you must remember to change the target machine's version.

Testing ActiveX/Visual Basic Controls

The topics in this section describe how Silk Test Classic provides built-in support for testing ActiveX controls and Visual Basic native controls with the Classic Agent.

The Open Agent does not provide dedicated support for testing ActiveX controls and native Visual Basic controls. However, the Open Agent supports mapping Visual Basic native controls to custom controls, by using the Win32 technology domain and the **Custom Controls** dialog box.

Overview of ActiveX/Visual Basic Support

Visual Basic 5 and 6 with the Classic Agent

Silk Test Classic provides built-in support for testing ActiveX controls and Visual Basic 5 and 6 native controls with the Classic Agent. These controls can be embedded in:

- Visual Basic 5 and 6 applications
- Other 32-bit Windows applications
- HTML Web pages

In addition, you can test more than one application at a time.

Visual Basic 4 with the Classic Agent

If you are testing Visual Basic 4 applications with the Classic Agent, you do not have access to properties and methods in native controls, just ActiveX controls. Since most Visual Basic 4 native controls map to Windows native controls, you can use Silk Test Classic's class mapping feature to test native controls.

Visual Basic with the Open Agent

The Open Agent does not provide dedicated support for testing ActiveX controls and native Visual Basic controls. However, the Open Agent supports mapping Visual Basic native controls to custom controls, by using the Win32 technology domain and the **Custom Controls** dialog box.

Enabling ActiveX/Visual Basic Support

This functionality is supported only if you are using the Classic Agent.

Before testing Visual Basic and ActiveX controls in a stand-alone application or in Internet Explorer, you need to enable extensions. If you are testing ActiveX controls in Internet Explorer, you must complete set up for testing ActiveX controls or Java applets in the browser.

Predefined Classes for ActiveX/Visual Basic Controls

This functionality is supported only if you are using the Classic Agent.

In addition to the 4Test classes provided in Silk Test Classic, support for Visual Basic with ActiveX controls includes predefined class definitions for:

- The native Visual Basic controls included in Microsoft Windows Visual Basic 6.0 Professional Edition.
- The ActiveX controls bundled with Visual Basic 6.0 Professional Edition.

These definitions are provided in a file as a convenience to help you quickly get started testing your applications.

Property names that begin with the prefix VB, for example, `rVBHeight` of the `OLEAniPushButton` class, are available only in Visual Basic applications. These properties are added to an ActiveX control by the Visual Basic environment. They are not available in C/C++ environments.

Do you need to record additional classes?

If your application contains only those controls shipped with the Microsoft Windows Visual Basic Professional Edition, then you do not need to record additional classes. If you are not sure, review the controls in your application and compare them to the table in the list of controls.

- If your application uses only these types of controls, you do not need to record additional classes. Go to *Testing ActiveX/Visual Basic controls*.
- If your application uses controls other than those listed in the table, for example, third-party ActiveX controls, you must record classes for these additional controls, as described in Recording new classes for ActiveX/Visual Basic controls. After you record the class, you can retrieve information about any number of instances (objects) of that class.

Predefined Class Definition File for Visual Basic

This functionality is supported only if you are using the Classic Agent.

The `vbclass.inc` file provides 4Test class definitions for the native Visual Basic controls and bundled ActiveX controls supported in included in Microsoft Windows Visual Basic 6.0 Professional Edition. Each class in the file lists the prototypes of all properties and methods for the class.

If you did not install Visual Basic/ActiveX support but later want to enable it to test your applications, you must manually edit `startup.inc` in order to use the predefined class definitions; otherwise, you will have to record all class definitions yourself. The procedure for manually enabling Visual Basic/ActiveX support is described in *Enabling the ActiveX/Visual Basic Support*.

List of Predefined ActiveX/Visual Basic Controls

This functionality is supported only if you are using the Classic Agent.

The following table lists each of the ActiveX and native Visual Basic controls for which classes are provided, the enhanced Visual Basic- and ActiveX-specific 4Test class each control is associated with, and the standard 4Test class from which that class is derived. For example, the `3D Check Box` control is associated with the `OLESSCheck` class, which is derived from the 4Test class `CheckBox`.

Native Visual Basic or ActiveX Control	4Test Class for the Control	Standard 4Test Class Derived From
3D Check Box Control	OLESSCheck	CheckBox
3D Command Button Control	OLESSCommand	PushButton
3D Frame Control	OLESSFrame	StaticText
3D Option Button Control	OLESSOption	RadioButton
3D Panel Control	OLESSPanel	Control
3D Group Push Button Control	OLESSRibbon	Control
Animation Control	OLEAnimation	Control
Animated Button Control	OLEAniPushButton	PushButton
CheckBox Control	VBCheckBox	CheckBox
ComboBox Control	VBComboBox	ComboBox
CommandButton Control	VBCommandButton	PushButton
Data Control	VBData	Control
DBCombo Control	OLEDBCombo	ComboBox
DBGrid Control	OLEDBGrid	Control
DBList Control	OLEDBList	ListBox
DirListBox Control	VBDirListBox	ListBox
DriveListBox Control	VBDriveListBox	PopupList
FileListBox Control	VBFileListBox	ListBox
Form	VBMainForm	MainWin
Form	VBChildForm	ChildWin
Form	VBForm	DialogBox
Frame Control	VBFrame	StaticText
Gauge Control	OLEGauge	Control
Graph Control	OLEGraph	ControlMultiWin
Grid Control	OLEGrid	Control
HScrollBar Control	VBHScrollBar	ScrollBar
Image Control	VBImage	Control
Key State Control	OLEMhState	Control
Label Control	VBLabel	StaticText
ListBox Control	VBListBox	ListBox
ListView Control	OLEListView	ListView
Masked Edit Control	OLEMaskEdBox	TextField

Native Visual Basic or ActiveX Control	4Test Class for the Control	Standard 4Test Class Derived From
MDIForm	VBMDIForm	MainWin
MSChart Control	OLEMSChart	Control
MSFlexGrid Control	OLEMSFlexGrid	Control
Multimedia MCI Control	OLEMMControl	ControlMultiWin
OLE Container Control	VBOLE	Control
OptionButton Control	VBOptionButton	RadioButton
Outline Control	OLEOutline	Control
PictureBox Control	VBPictureBox	Control
ProgressBar Control	OLEProgressBar	Control
RichTextBox Control	OLERichTextBox	TextField
Shape Control	VBShape	Control
Slider Control	OLESlider	Scale
SpinButton Control	OLESpinButton	Control
SSTab Control	OLESSTab	PageList
StatusBar Control	OLEStatusBar	StatusBar
TabStrip Control	OLETabStrip	Control
TextBox Control	VBTextBox	TextField
Toolbar Control	OLEToolbar	ToolBar
TreeView Control	OLETreeView	TreeView
UpDown Control	OLEUpDown	UpDown
VScrollBar Control	VBVScrollBar	ScrollBar

Access to VBOptionButton Control Methods

This functionality is supported only if you are using the Classic Agent.

To access the ActiveX methods and properties of a control of class `VBOptionButton`, you must set the **Don't Group Radio Buttons Into a List** option in the **Agent Options** dialog box. There are several ways to do this:

Locally Enter the following statement in your script(s): `Agent.SetOption (OPT_RADIO_LIST, FALSE)`. The advantage of setting the option locally is that you can still treat a group of buttons as a radio list, for example, for selection purposes.

Globally Open the **Agent Options** dialog box, click the **Compatibility** tab, and check the **Don't Group Radio Buttons into a List** check box.

0-Based Arrays

This functionality is supported only if you are using the Classic Agent.

When you access arrays using the methods provided in the Visual Basic and ActiveX extension, the arrays are 0-based; that is, the first value is stored in slot 0. In contrast, `GetArrayProperty` and

`SetArrayProperty`, two methods provided for backward compatibility with previous releases, used 1-based arrays. The following example illustrates the current and old syntax:

```
testcase GetColWidthForGrid () appstate none

   INTEGER iWidth1, iWidth2

   //Get width of col. 1 in MyGrid, using current syntax
   //Note that index (passed to GetColWidth method) is 0
   iWidth1 = MyApp.MyGrid.GetColWidth(0)

   //Changes the width of col. 1, using current syntax
   MyApp.MyGrid.SetColWidth (0, 555)

   //Gets width of col. 1 in MyGrid, using old syntax
   //Note that index of the ColWidth property arrow
   //(passed to GetArrayProperty method) is 1
   iWidth2 = MyApp.MyGrid.GetArrayProperty ("ColWidth", 1)
```

Passing an index of 0 to `GetArrayProperty` causes an error at runtime.

Dependent Objects and Collection Objects

This functionality is supported only if you are using the Classic Agent.

Active X controls can be composed of objects which themselves expose properties and methods. An example is the Data control, which contains a Recordset control. Such contained objects are often referred to as dependent objects because they don't exist outside the context of the containing control. In many cases, dependent objects are arranged into groups, or collections. For example, a TreeView control contains a collection of Node objects.

Users of the ActiveX control need a way to get at dependent objects. In the case of a simple dependent object, the outer control typically exposes a property that provides access to the contained object. In the case of a collection, the outer control provides access to the items in a collection through an intermediate object called a collection object.

You can get programmatic access to dependent objects by having the relevant control class inherit from a special class provided for this purpose: the `CompoundControl` class. This class provides methods for accessing the properties and methods of a simple dependent object, a collection object, or the individual items within a collection.

Working with Dynamically Windowed Controls

This functionality is supported only if you are using the Classic Agent.

In Internet Explorer, ActiveX controls may be dynamically windowed, which means their windows may come and go, or change location as the object is scrolled in and out of view. Consequently, recording declarations and actions against such objects can be tricky. You can achieve the most consistent results by bringing the ActiveX control into full view when recording declarations, classes, or actions. If you don't bring the ActiveX control into full view, Silk Test Classic might not recognize it correctly.

Window Timeout

This functionality is supported only if you are using the Classic Agent.

When you install Visual Basic and ActiveX support in Silk Test Classic, the **Window Timeout** setting in the **Agent Options** dialog is initially set to 20 seconds. This setting determines how long the recovery system waits when checking to see if your application exists. If you choose to change the **Window Timeout** setting to a low number or 0, you may encounter a timing problem, where rapidly exiting and restarting an application may generate "Windows not found" errors.

Setting this option is not required when testing ActiveX controls in Internet Explorer, but is recommended as a general practice.

Conversion of BOOLEAN Values

This functionality is supported only if you are using the Classic Agent.

In some cases, when you record a class declaration, the result is a method prototype that takes a SHORT data type as a parameter even though the associated Visual Basic property takes a BOOLEAN parameter.

For example, in the following Visual Basic prototype, the syntax of the `TabEnabled` property of the `SSTab` class is:

```
object.TabEnabled (tab) [= boolean]
```

where boolean, the return value, is a BOOLEAN value: TRUE for enabled, FALSE for disabled. The prototype for its method equivalent in the SSTab class is shown in `vbclass.inc` as:

```
ole VOID SetTabEnabled (SHORT Index, SHORT Arg1)
```

Note: The `SetTabEnabled` method takes a value of type SHORT as its second argument, which is equivalent to the BOOLEAN argument of the property. However, because 4Test and ActiveX define the BOOLEAN type differently, attempting to pass TRUE or FALSE as you would in Visual Basic will generate an argument type mismatch error in Silk Test Classic. Use the following constant values instead, which are defined in `oleclass.inc`:

- `OLE_TRUE (which equals –1)`
- `OLE_FALSE (which equals 0)`

Testing Controls: 4Test Versus ActiveX Methods

This functionality is supported only if you are using the Classic Agent.

The most effective way to test a control depends on the type of the control. For some controls, the inherited 4Test methods are more efficient; for others, the ActiveX methods or properties are more efficient. In general, we recommend that you begin by trying the 4T4Testest methods associated with the class from which the control is derived. For example, for the `OLERichTextBox` use the 4Test methods and properties for `TextField`.

Control Access is Similar to Visual Basic

This functionality is supported only if you are using the Classic Agent.

You can access a control's internal properties and methods using the dot operator and a syntax similar to standard Visual Basic.

Example

To retrieve the number of rows in a Grid ActiveX control, called MyGrid, you might use the `iRows` property, as follows:

```
INTEGER iNumRows
iNumRows = MyApp.MyGrid.iRows
```

Example

To set the number of rows in the same grid to 10, you might use:

```
MyApp.MyGrid.iRows = 10
```

Prerequisites for Testing ActiveX/Visual Basic Controls

This functionality is supported only if you are using the Classic Agent.

You are ready to begin testing your application. First, please read the information on designing and recording test cases in *Overview of Test Cases*. Become familiar with the basic concepts of test case design, including the setup, verification, and cleanup stages, and the **Record** > **Testcase** and **Record** > **Action** menu items.

Additionally, to test Visual Basic/Active X controls with Silk Test Classic, you must fulfil the following prerequisites:

- You are familiar with routine Silk Test Classic tasks.
- You understand properties and methods as they relate to ActiveX controls or native Visual Basic controls.
- You have access to the documentation for any ActiveX controls that are embedded in your test application, and whose properties or methods you want to call.

ActiveX/Visual Basic Exception Values

This functionality is supported only if you are using the Classic Agent.

Silk Test Classic generates exception values under given error conditions. These values are described in the Help. In addition, the ActiveX/Visual Basic support defines the following set of exception values:

E_SPY_NOT_RESPONDING	Unable to connect to the Visual Basic or Windows application with embedded OLE controls.
E_OBJ_CALL_FAILED	The method or property call returned an error.
E_OBJ_NOT_FOUND	The Visual Basic or OLE control object could not be found.
E_ARG_TYPE_MISMATCH	One of the arguments has the wrong type.
E_BAD_ARG_COUNT	Wrong number of arguments for this method or property call.
E_ARG_VAL_OUT_OF_RANGE	One of the arguments had a value that was out of range.

Recording New Classes for ActiveX/Visual Basic Controls

This functionality is supported only if you are using the Classic Agent.

The process of recording a class involves querying the objects in your application, retrieving information on properties and methods, and then translating the information into 4Test-style prototypes. Silk Test Classic does this automatically for you when you select **Record** > **Class**.

We recommend that you create an include file (for example, `userclass.inc`) for your new class definitions, instead of entering them in `vbclass.inc`. This way you will not have to update `vbclass.inc` each time you record new class definitions.

1. Start your application. If you are recording classes for controls on a web page, navigate to that Web page.
2. Open the include file you created for your new class definitions. For information on how to load class definition files, see *Loading Class Definition Files*.
3. Click **Record** > **Class** > **Scripted** to open the **Record Class** dialog box.
4. Position the mouse pointer over the control for which you want to record a class.
5. When the correct name displays in the Window field, press `Ctrl+Alt`. Properties and methods for that class are displayed in the **Record Class** dialog. Do not edit the tag name in the Tag field.

6. Click the **Derived From** drop-down menu to see the list of available 4Test classes. Then proceed as follows:
   - If there is a class type available that maps directly to your object, choose it. For example, if your object is a `SuperListBox`, you might choose `ListBox` (note that similarly named objects might not behave as expected). Your object will inherit all the standard 4Test methods and properties defined for a list box.
   - If there is not a class type that maps directly to your object, choose **Control**, which is a generic class.

   See winclass declaration and derived class for more details.

   The Agent provides special handling for certain classes of objects. If your object is one of these types, but does not work correctly while you are testing your application, you will also need to class map the object after completing this procedure. For more information, see *Options for Non-Graphical Custom Controls*.
7. Click **Paste to Editor** to paste the new class into the include file.
8. Repeat this process for every type of control in your application that does not appear in the list of classes provided. When you are done recording classes, and then click **Close**.

Loading Class Definition Files

If you record classes into a file other than `vbclass.inc`, which we recommend, you must have Silk Test Classic load the file in one of the following ways:

- In the **Use Path** fields of the **Runtime Option** dialog box, enter the path of the directory that contains the class include file. By default, these files are installed in the Silk Test Classic installation directory. Then, in the **Use Files** field, specify the name and extension of the include file.
- Insert a use statement in each script that needs to manipulate objects of the classes you just declared. Use the following format:

  ```
  use  "directory\file-name.inc"
  ```

  For example, to have Silk Test Classic load the class include file for third-party ActiveX controls, `thirdpty.inc`, which resides in the default directory `c:\mydir`, insert the following statement:

  ```
  use  "c:\mydir\thirdpty.inc"
  ```
- If most of your work involves testing Visual Basic applications, you may want to add a use statement to your `startup.inc` file.

Disabling ActiveX/Visual Basic Support

This functionality is supported only if you are using the Classic Agent.

You disable ActiveX/Visual Basic support for a particular application on the host machine, rather than in general.

1. On the host machine, click **Options** > **Extensions** to open the **Extensions** dialog box.
2. For the application you want to disable, uncheck the **ActiveX** check box.
3. Click **OK** to close the **Extensions** dialog box.

   When you're done testing the application, you may want to remove it from the **Extensions** dialog box as well as from the **Extension Enabler** dialog box.

Ignoring an ActiveX/Visual Basic Class

This functionality is supported only if you are using the Classic Agent.

If you are using the ActiveX/Visual Basic extension and you want to ignore a class, you must edit the `axext.ini` configuration file.

1. Open the `axext.ini` file, which is installed by default in the `<SilkTest installation directory>/extend` folder.
2. In the `[OmitClasses]` section, enter either the class names or class ids, separated by commas. For example:

```
[OmitClasses]
RawClassName=
CLSID=00010001-0000-0000-0000-111111111111
```

3. Save and close `axext.ini`. The next time you open Silk Test Classic, Silk Test Classic ignores the class (or classes) you've listed.

Setting ActiveX/Visual Basic Extension Options

This functionality is supported only if you are using the Classic Agent.

The ActiveX/VB options you can set are:

- `SetAdvSink` which sets the Advise Sink.
- `ShowAllWin` which controls whether Silk Test Classic optimizes window hierarchy by skipping some windows.
- `MsgTimeout` which controls the timeout for messaging in the extension.

You can set the ActiveX/VB extension options by manually editing the `axext.ini` file. The settings go in the `[VBOptions]` section of `axext.ini` and are optional. You do not have to include them in your `axext.ini` if you want the default behavior.

SetAdvSink option

Default is TRUE. Setting the advise sink may cause certain applications to crash if they frequently destroy and recreate ActiveX/VB controls. Try setting this option to FALSE if your application under test crashes. Setting the option to FALSE disables the code in the ActiveX/VB extension that sets the advise sink.

ShowAllWin option

Default is FALSE. `ShowAllWin` lets you control whether or not Silk Test Classic recognizes the full window hierarchy in ActiveX applications. The default FALSE setting causes the ActiveX extension to construct a simplified window hierarchy that filters out windows perceived to be containers. Setting the option to TRUE causes the ActiveX extension to recognize the full window hierarchy. Try setting this option to TRUE if a control that you need to test is not recognized because it has the same position and size (in other words, the same rectangle) as another control that is recognized.

MsgTimeout option

Default is 1000. Setting the `MsgTimeout` option controls the number in milliseconds of the timeout used for messaging in the extension. 1000 milliseconds corresponds to 1 second. Try increasing the timeout to 2000 or 3000 (2 or 3 seconds) if you notice the following symptoms appearing at apparently random intervals:

- The VB/ActiveX extension recognizes certain controls as CustomWin rather than as 4Test or recorded classes.
- The application under test crashes while a test script is running.

Increasing the value of MsgTimeout may slow down the performance of Silk Test Classic when interacting with an application.

To edit the ActiveX/VB extension options

1. Close Silk Test Classic and your application under test, if they are open.

2. Open `axext.ini`, located in the extend subdirectory of the directory where you installed Silk Test Classic. If you are using a Silk Test Classic Project, the applicable `axext.ini` file is still in the Silk Test Classic install directory.
3. Go to the `[VBOptions]` section and change the value of the option. You may need to add a line containing the `[VBOptions]` section name, if it does not already exist.
4. Save and close the `axext.ini` file.
5. Restart Silk Test Classic.

Setup for Testing ActiveX Controls or Java Applets in the Browser

This functionality is supported only if you are using the Classic Agent.

To test ActiveX controls in Internet Explorer, enable the extension for the version of the browser that you are using. When you enable the extension, be sure to check the **ActiveX** check box for the extension. This must be done manually whether you enable the extension manually or automatically.

To test Java applets in the browser, you must check the Java check box for the browser extension. In most cases, Silk Test Classic detects the applet and automatically checks the check box.

Client/Server Application Support

Silk Test Classic provides built-in support for testing client/server applications including:

- .NET WinForms
- Java AWT applications
- Java SWT/RCP application
- Java Swing applications
- Windows-based applications

In a client/server environment, Silk Test Classic drives the client application by means of an Agent process running on each application's machine. The application then drives the server just as it always does. Silk Test Classic is also capable of driving the GUI belonging to a server or of directly driving a server database by running scripts that submit SQL statements to the database. These methods of directly manipulating the server application are intended to support testing in which the client application drives the server. For additional information on this capability, see *Testing Databases*.

Client/Server Testing Challenges

Silk Test Classic provides powerful support for testing client/server applications and databases in a networked environment. Testing multiple remote applications raises the level of complexity of QA engineering above that required for stand-alone application testing. Here are just a few of the testing methodology challenges raised by client/server testing:

- Managing simultaneous automatic regression tests on different configurations and platforms.
- Ensuring the reproducibility of client/server tests that modify a server database.
- Verifying the server operations of a client application independently, without relying on the application under test.
- Testing the concurrency features of a client/server application.
- Testing the intercommunication capabilities of networked applications.
- Closing down multiple failed applications and bringing them back to a particular base state (recovery control).
- Testing the functioning of the server application when driven at peak request rates and at maximum data rates (peak load and volume testing).

- Automated regression testing of multi-tier client/server architectures.

Verifying Tables in ClientServer Applications

This functionality is supported only if you are using the Classic Agent.

When verifying a table in a client/server application, that is, an object of the `Table` class or of a class derived from `Table`, you can verify the value of every cell in a specified range in the table using the **Table** tab in the **Verify Window** dialog box. For additional information on verifying tables in Web applications, see *Working with Borderless Tables*.

Specifying the range

You specify the range of cells to verify in the **Range** text boxes using the following syntax for the starting and ending cells in the range:

```
row_number : column_name
```

or

```
row_number : column_number
```

> **Example**
>
> Specifying the following in the **Range** text boxes of the **Verify Window** dialog box causes the value of every cell in rows 1 through 3 to be verified, starting with the column named ID and ending with the column named Company_Name:
>
> From field: 1 : id
>
> To field: 3 : company_name
>
> After you specify a cell range in the **Verify Window** dialog box, you can click **Update** to display the values in the specified range.

Specifying a file to store the values

You specify a file to store the current values of the selected range in the **Table File Name** text box.

What happens

When you dismiss the **Verify Window** dialog box and paste the code into your script, the following occurs:

- The values that are currently in the table's specified cell range are stored in the file named in the **Table File Name** text box in the **Verify Window** dialog box.
- A `VerifyFileRangeValue` method is pasted in your script that references the file and the cell range you specified.

For example, the following `VerifyFileRangeValue` method call would be recorded for the preceding example:

```
table.VerifyFileRangeValue ("file.tbl", {{"1",
"id"}, {"3", "company_name"}})
```

When you run your script, the values in the range specified in the second argument to `VerifyFileRangeValue` are compared to the values stored in the file referenced in the first argument to `VerifyFileRangeValue`.

For additional information, see the `VerifyFileRangeValue` method.

Evolving a Testing Strategy

There are several reasons for moving your QA program from local to remote testing:

- You may have a stand-alone application that runs on many different platforms and now you want to simultaneously drive testing on all the platforms from one Silk Test Classic host system.
- You may have been testing a client/server application as a single local application and now you want to drive multiple instances of the application so as to apply a heavier load to the server.
- You may want to upgrade your client/server testing so that your test cases can automatically initialize the server and recover from server failures— in addition to driving multiple application instances.
- You may need to test applications that have different user interfaces and that communicate as peers.

If you are already a Silk Test Classic user, you will find that your testing program can evolve in any of these directions while preserving large portions of your existing tests. This topic and related topics help you to evolve your testing strategy by showing the incremental steps you can take to move into remote testing.

Incremental Functional Test Design

Silk Test Classic simplifies and automates the classic QA testing methodology in which testing proceeds from the simplest cases to the most complex. This incremental functional testing methodology applies equally well in the client/ server environment, where testing scenarios typically proceed from the simplest functional testing of one instance of a client application, to functional and performance testing of a heavily loaded, multi-client configuration. Therefore, we recommend the following incremental progression for client/server testing:

- Perform functional testing on a single client application that is running on the same system as Silk Test Classic, with the server application on the same system (if possible).
- Perform functional testing on a single remote client application, with the server application on a separate system.
- Perform functional and concurrency testing on two remote client applications.
- Perform stress testing on a single client application running locally or remotely.
- Perform volume load testing on a configuration large enough to stress the server application.
- Perform peak load testing on a large configuration, up to the limits of the server, if possible.
- Perform performance testing on several sets of loads until you can predict performance.

Network Testing Types

Software testing can be categorized according to the various broad testing goals that are the focus of the individual tests. At a conceptual level, the kinds of automated application testing you can perform using Silk Test Classic in a networked environment are:

- Functional
- Configuration
- Concurrency

The ordering of this list conforms to the incremental functional testing methodology supported by Silk Test Classic. Each stage of testing depends for its effectiveness on the successful completion of the previous stage. Functional, configuration, and concurrency testing are variations of regression testing, which is a prerequisite for any type of load testing. You can use Silk Performer for load testing, stress testing, and performance testing.

You can perform functional testing with a single client machine. You can perform the first four types of test with a testbed containing only two clients. The last two testing types require a heavy multi-user load and so need a larger testbed.

Concurrency Testing

Concurrency testing tests two clients using the same server. This is a variation of functional testing that verifies that the server can properly handle simultaneous requests from two clients. The simplest form of concurrency testing verifies that two clients can make multiple non-conflicting server requests during the same period of time. This is a very basic sanity test for a client/server application.

To test for problems with concurrent access to the same database record, you need to write specific scripts that synchronize two clients to make requests of the same records in your server's databases at the same time. Your goal is to encounter faulty read/write locks, software deadlocks, or other concurrency problems.

Once the application passes the functional tests, you can test the boundary conditions that might be reached by large numbers of transactions.

Configuration Testing

A client/server application typically runs on multiple different platforms and utilizes a server that runs on one or more different platforms. A complete testing program needs to verify that every possible client platform can operate with every possible server platform. This implies the following combinations of tests:

- Test the client application and the server application when they are running on the same machine—if that is a valid operational mode for the application. This testing must be repeated for each platform that can execute in that mode.
- Test with the client and server on separate machines. This testing should be repeated for all different platform combinations of server and client.

Functional Testing

Before you test the multi-user aspects of a client/server application, you should verify the functional operation of a single instance of the application. This is the same kind of testing that you would do for a non-distributed application.

Once you have written scripts to test all the operations of the application as it runs on one platform, you can modify the scripts as needed for all other platforms on which the application runs. Testing multiple platforms thus becomes almost trivial. Moreover, many of the tests you script for functional testing can become the basis of your other types of testing. For example, you can easily modify the functional tests (or a subset of them) to use in load testing.

Peak Load Testing

Peak load testing is placing a load on the server for a short time to emulate the heaviest demand that would be generated at peak user times—for example, credit card verification between noon and 1 PM on Christmas Eve. This type of test requires a significant number of client systems. If you submit complex transactions to the server from each client in your test network, using minimal user setup, you can emulate the typical load of a much larger number of clients.

Your testbed may not have sufficient machines to place a heavy load on your server system — even if your clients are submitting requests at top speed. In this case it may be worthwhile to reconfigure your equipment so that your server is less powerful. An inadequate server configuration should enable you to test the server's management of peak server conditions.

Volume Testing

Volume testing is placing a heavy load on the server, with a high volume of data transfers, for 24 to 48 hours. One way to implement this is to use one set of clients to generate large amounts of new data and another set to verify the data, and to delete data to keep the size of the database at an appropriate level. In such a case, you need to synchronize the verification scripts to wait for the generation scripts. The 4Test script language makes this easy. Usually, you would need a very large test set to drive this type of server load, but if you under-configure your server you will be able to test the sections of the software that handle the outer limits of data capacity.

How 4Test Handles Script Deadlock

It is possible for a multi-threaded 4Test script to reach a state in which competing threads block one another, so that the script cannot continue. This is called a script deadlock. When the 4Test runtime environment detects a deadlock, it raises an exception and halts the deadlocked script.

Example

The following script will never exit successfully.

```
share INTEGER iIndex1 = 0
share INTEGER iIndex2 = 0

main ()
  parallel
    access iIndex1
      Sleep (1)
        access iIndex2
          Print ("Accessed iIndex1 and iIndex2")
    access iIndex2
      Sleep (1)
        access iIndex1
          Print ("Accessed iIndex2 and iIndex1")
```

Troubleshooting Configuration Test Failures

The test of your application may have failed for one of the reasons below. If the following suggestions do not address the problem, you can enable your extension manually.

Note: Unsupported and embedded browsers, other than AOL, are recognized as client/server applications.

The application may not have been ready to test

1. Click **Enable Extensions** on the **Basic workflow** bar.
2. On the **Enable Extensions** dialog box, select the application for which you want to enable extensions.
3. Close and restart your application. Make sure the application has finished loading, and then click **Test**.

Embedded browsers, other than AOL, are recognized as Client/Server applications

If you want to work with a web browser control embedded within an application, you must enable the extension manually.

Testing .NET Applications with the Classic Agent

Silk Test Classic provides built-in support for testing .NET applications with the Classic Agent.

For information about new features, supported platforms and versions, known issues, and work-arounds, refer to the *Release Notes*.

Enabling .NET Support

This functionality is supported only if you are using the Classic Agent.

Before testing .NET controls, you need to enable extensions.

Do you need to record additional classes?

- If your application contains only the standard .NET WinForms controls, then you do not need to record additional classes. If you are not sure, see .NET classes.
- If your application uses other controls, for example, third-party controls, and the Classic Agent, you must record classes for these additional controls. After you record the class, you can retrieve information about any number of instances (objects) of that class.

Tips for Working with .NET

This topic discusses common problems when testing .NET applications with Silk Test Classic and how to solve them.

This functionality is supported only if you are using the Classic Agent.

DefaultBaseState() enters an infinite loop when trying to close a dialog box

This problem is probably caused by the tag of the dialog box. If you declared the dialog box with multitags, make sure that the first tag in the multitag is correct. Usually the first tag is the caption of the dialog box. If the dialog box can have multiple captions, use wildcards to create a caption that covers all of them. Alternatively, if the dialog box has a valid window ID, you may want to use that as the tag or first multitag instead of the caption. If you must rely on a multitag for the dialog box, then be sure to close the dialog box explicitly as part of your test case.

This is not a problem for just the .NET extension; it can occur with any application. However, .NET applications are one of the few for which dialog boxes have valid Window IDs. Other dialog boxes only have valid captions, so they usually only have a single tag instead of a multitag. Therefore if the tag is wrong, it needs to be corrected in order to run the test case.

Predefined class definition file for .NET

The `dotnet.inc` file provides definitions for the supported .NET classes. Each class in the file lists the prototypes of all properties and methods for the class.

If you did not install .NET support, but later want to enable it to test your applications, you must manually edit `dotnet.inc` in order to use the predefined class definitions; otherwise, you will have to record all class definitions yourself.

DefaultBaseState does not work

After you remove or change the security string for the Framework, the DefaultBaseState might not work when you run Silk Test Classic. Follow the instructions described in *Setting Your Machine Zone Security* and use the instructions that begin with *Open your Control Panel…*. Be sure to set the security for the Framework.

Silk Test Classic only records the first character when you press and hold a key in a .NET text box or combobox

If the cursor is in a text box or a combobox and you enter text by pressing down and holding a key, so that multiple characters are entered, only the first character is recorded. For example, if you press and hold the "x" key causing many x's to display in the text box, Silk Test Classic does not record "xxxxxxxxxxxxxx"; Silk Test Classic only records a single "x". You have to manually edit the argument to `SetText` in the script if you want Silk Test Classic to play back multiple characters.

Windows Forms Applications

Silk Test Classic provides built-in support for testing .NET Windows Forms (Win Forms) applications using the Open Agent as well as built-in support for testing .NET standalone and No-Touch Windows Forms (Win Forms) applications using the Classic Agent. However, side-by-side execution is supported only on standalone applications.

For information about new features, supported platforms and versions, known issues, and work-arounds, refer to the *Release Notes*.

Using the Classic Agent to Test Windows Forms Applications

This section describes how you can use the Classic Agent to test Windows Forms applications.

For a complete list of the record and replay controls available for Win Forms testing with the Classic Agent, see *.Net Classes Used by the Classic Agent*.

No-Touch Windows Forms Application Support

This functionality is supported only if you are using the Classic Agent.

Windows Forms applications are desktop applications that are built using the Windows Forms classes of the .NET Framework. The .NET Framework allows system administrators to deploy applications and updates to applications through a remote Web server. This technology is called no-touch deployment. With no-touch deployment, applications can be downloaded, installed, and run directly on the machines of the user without any alteration of the registry or shared system components.

No-touch deployment support is part of the .NET extension of Silk Test Classic; this includes Window Forms applications.

Silk Test Classic provides built-in support for testing .NET no-touch Windows Forms applications. Silk Test Classic does not support side-by-side execution of no-touch applications.

Before you begin testing your no-touch application, see the *Prerequisites for Testing .NET No-Touch Applications*.

Prerequisites for Testing .NET No-Touch Applications

This functionality is supported only if you are using the Classic Agent.

Before you can use Silk Test Classic to test .NET no-touch applications, you must:

- Have System Administrator privileges on the machine, on which the Silk Test agent is running.
- Have either the 2.0 or 1.1.NET Framework installed; these Frameworks are available for any version of Windows.
- Have installed the .NET Framework Redistributable package, which contains everything needed to run a program under the .NET Framework.
- Ensure that the `Segue.SilkTest.Net.Shared.dll` has been installed into the Global Assembly Cache (GAC).
- Set the Machine Zone Security.

After you complete these steps, you can use the Basic Workflow to test any .NET no-touch application.

Recording New Classes for .NET Controls

This functionality is supported only if you are using the Classic Agent.

After you enable extensions for testing a .NET Windows Forms application with the Classic Agent, if you see CustomWin declarations in the Record Window Declarations, then click **Record** > **Class** in order to work with the CustomWin controls. The process of recording a class involves querying the objects in your application, retrieving information on properties and methods, and then translating the information into 4Test-style prototypes. Silk Test Classic does this automatically when you click **Record** > **Class**.

When you use the **Record Class** dialog box to record new classes for .NET controls, Silk Test Classic automatically inserts the __typeinfo keyword in front of any method that has parameters that are either:

- Of type POINT or RECT.
- Of a type that has been declared explicitly in 4Test and uses the alias mechanism.

We recommend that you create an include file, for example `userclass.inc`, for your new class definitions, instead of entering them in `dotnet.inc`. The `dotnet.inc` include file is shipped with Silk Test Classic and we reserve the right to modify this file in future releases. If you modify `dotnet.inc`, you may have to integrate your changes into future versions of that file.

To record new classes:

1. Start the .NET standalone Windows Forms application.
2. Open the include file that you created for your new class definitions.

   For information on how to load class definition files, see *Several Ways to Load Class Definition Files*.
3. Click **Record** > **Class** > **Scripted** to open the **Record Class** dialog box.
4. Position the mouse pointer over the control for which you want to record a class.
5. When the correct name displays in the **Window** text box, press **Ctrl+Alt**.

   Properties and methods for that class are displayed in the **Record Class** dialog box. Do not edit the tag name in the **Tag** text box. If you check **Show all methods** on the **Record Class** dialog box, you see a commented section called **Other methods**. Some methods are preceded by two slashes (//) and others are preceded by three slashes plus two asterisks (/// **). The methods that are preceded by "/// ** " cannot be called by Silk Test Classic; they are included only for reference purposes.

   The methods in the **Other methods** section that are preceded by two slashes can potentially be called by Silk Test Classic. These are methods that:

   - Have parameters or return values of a type not declared in 4Test. To call such a method, you must explicitly declare the types using alias. The data types in the argument list are just suggestions. You must consult the documentation for the control in order to determine the native data type name and definition. After you have declared the types, you should compile your frame file and click **Record** > **Class**. Now the recorder will recognize your new data types and will record as usable those methods that were previously commented out because the data types had not been defined.
   - Are overloaded methods, in other words methods for which the parameter list may vary. 4Test does not support overloading of methods, since each method must have a unique name. You may choose to un-comment one of the overloaded methods for use within your test scripts.
6. Click the **Derived From** list box to see the list of available 4Test classes. Then proceed as follows:

   - If there is a class type available that maps directly to your object, choose it. For example, if your object is a SuperListBox, you would likely choose `ListBox`. Your object will inherit all the standard 4Test methods and properties defined for a list box.

     Note: Similarly named objects might not behave as expected.
   - If there is not a class type that maps directly to your object, choose `Control`, which is a generic class.

   See `winclass` declaration and derived class for more details.

   The Agent provides special handling for certain classes of objects. If your object is one of these types, but does not work correctly while you are testing your application, you will also need to class map the object after completing this procedure. For additional information, see *Options for Non-Graphical Custom Controls*.
7. Click **Paste to Editor** to paste the new class into the include file.
8. Repeat this process for every type of control in your application that does not display in the list of classes provided.
9. When you are done recording classes, click **Close**.

Recording Actions on the DataGrid

This functionality is supported only if you are using the Classic Agent.

For standard `DataGrids`, Silk Test Classic records the following 4Test methods using the Classic Agent, depending on the location or component in the grid:

- `ClickCell`()
- `ClickCellButton`()
- `ClickCol`()

- `ClickRow()`
- `Collapse()`
- `Expand()`
- `SetCellValue()`
- `SetFocusCell()`

Notes

For DataGrids with natively supported, embedded controls, Silk Test Classic records the appropriate method call for the control with which you are interacting.

For DataGrids with embedded custom controls, Silk Test Classic records low-level events, such as TypeKeys and Click.

Silk Test Classic records a ClickCell before a SetCellValue.

If you use the Open Agent, the `DataGrid` class uses a different set of methods.

Example

If you record changing the value "Pine" to "Maple" in the Cell "Last_Name", Silk Test Classic generates:

```
SwfDialogBox("SamplesExplorer").SwfDialogBox("Sort").DataGrid("P
rototype Grid")
  .ClickCell ({{1,1,2}, "Last_Name"})
SwfDialogBox("SamplesExplorer").SwfDialogBox("Sort").DataGrid("P
rototype Grid")
  .SetCellValue({{1,1,2}, "Last_Name"}, "Maple")
```

Record Class, Window Declarations, and Window Identifiers

Record Window Declarations displays an instance of the `DataGrid` class when you cursor over the grid.

Record Class is not supported for components in the `DataGrid`, but it does record class on Controls, such as SwfTextField and CustomWin, in a DataGrid cell.

When you hover the cursor over a grid, Record Window Identifiers displays an instance of the `DataGrid` class. Record Window Identifiers does not display any components in the grid; it displays controls, such as SwfTextField and CustomWin, in a DataGrid cell.

Setting Your Machine Zone Security

This functionality is supported only if you are using the Classic Agent.

Before you can begin testing .NET no-touch applications, you must set up your machine security. You can disable security so that Silk Test Classic can test your no-touch application using the command prompt or the control panel.

Setting security changes various permissions for .NET no-touch applications. For example, UIPermission controls access to user interface, ReflectionPermission controls access to metadata through the System.Reflection APIs, and SecurityPermission controls a set of security permissions applied to code.

Setting Your Machine Zone Security using the Command Prompt

This functionality is supported only if you are using the Classic Agent.

1. Open a command prompt.
2. Type `caspol -machine -chggroup LocalIntranet_Zone FullTrust`. This opens the specified zone to the Full Trust level.

`LocalIntranet_Zone` is just one example; you might need to adjust to a different code group, depending on what your application is using. Check with your application's developer if you are not sure.

Now that you have set the Machine Zone Security, make sure you have installed the `Segue.SilkTest.Net.Shared.dll`.

Setting Your Machine Zone Security using the Control Panel

This functionality is supported only if you are using the Classic Agent.

1. Open the Control Panel and navigate to the **Administrative Tools** folder.
2. Double-click **Microsoft.NET Framework Configuration**.
3. In the **.NET Framework Configuration** tool, click the **Runtime Security Policy** node.
4. Click **Adjust Zone Security**.
5. Select **Make changes to this computer** and click **Next**.
6. On the **Adjust the Security Level for Each Zone** dialog box, choose **Full Trust level for this zone**.
7. Click **Next** to apply your changes.
8. Click **Finished**.
9. Repeat as necessary for the other Framework Configuration (if you have both installed).

Now that you have set the Machine Zone Security, make sure you have installed the `Segue.SilkTest.Net.Shared.dll`.

Ensuring that the Segue.SilkTest.Net.Shared.dll has been Installed

This functionality is supported only if you are using the Classic Agent.

Installing Silk Test Classic should install `Segue.SilkTest.Net.Shared.dll` into the GAC. To verify that the DLL is in the GAC:

1. Click **Control Panel** > **Administrative Tools**.
2. Click **Microsoft .NET Framework <version number> Configuration**.
3. Right-click **Assembly Cache** under **My Computer**.
4. Click **View** > **Assemblies** . The item Segue.SilkTest.Net.Shared should be in the list. You may have to scroll down to see it.
5. If the item Segue.SilkTest.Net.Shared is not in the list, add it by doing one of the following:
   - Right-click **Assembly Cache** under **My Computer** and click **Add**, then browse to the DLL in the Silk Test Classic installation directory.
   - Click `Segue.SilkTest.Net.Shared.dll` and drag it into the `<Windows directory>\Assembly directory`, which will cause the DLL to be added to the GAC.

Now that you have installed the `Segue.SilkTest.Net.Shared.dll`, you must also set the machine zone security.

Suppressing Controls (Classic Agent)

This functionality is supported only if you are using the Classic Agent.

You can suppress the controls for certain classes for .NET, Java SWT, and Windows API-based applications. For example, you might want to ignore container classes to streamline your test cases. Ignoring these unnecessary classes simplifies the object hierarchy and shortens the length of the lines of code in your test scripts and functions. Container classes or 'frames' are common in GUI development, but may not be necessary for testing.

The following classes are commonly suppressed during recording and playback:

Technology Domain	Class
.NET	Group
Java SWT	org.eclipse.swt.widgets.Composite
	org.eclipse.swt.widgets.Group
Windows API-based applications	Group

To suppress specific controls:

1. Click **Options** > **Class Map**. The **Class Map** dialog box opens.
2. In the **Custom class** field, type the name of the class that you want suppress.
   The class name depends on the technology and the extension that you are using. For Windows API-based applications, use the Windows API-based class names. For Java SWT applications, use the fully qualified Java class name. For example, to ignore the **SWT_Group** in a Windows API-based application, type `SWT_Group`, and to ignore to ignore the `Group` class in Java SWT applications, type `org.eclipse.swt.widgets.Group`.
3. In the **Standard class** list, select **Ignore**.
4. Click **Add**. The custom class and the standard class display at the top of the dialog box.

Infragistics Controls

This functionality is supported only if you are using the Classic Agent.

Silk Test Classic has several built-in 4Test classes that support recording and playback for key Infragistics Window Forms controls. These include:

- The `UltraWinGrid` controls through the 4Test `DataGrid` class.
- The `UltraWinToolbars` container and the elements within the `UltraWinToolbars` through the 4Test `Toolbar` class (Infragistics' ToolBase).

Native support means that features such as the following are available when testing these controls:

- Action-based recording.
- Record Window declarations.
- Record identifiers.
- Record class.

Before you can use Silk Test Classic to test Infragistics Windows Forms controls, you must:

- Install the .NET Framework.
- Install NetAdvantage Window Forms.

For specific versions of these applications and other installation requirements, refer to the *Release Notes*.

Testing the Infragistics UltraWinGrid and UltraWinToolbars

Silk Test Classic has two classes for Infragistics support:

- `DataGrid`, which supports the Infragistics UltraWinGrid.
- `UltraWinToolbars`, which supports the Infragistics UltraWinToolbars container and the UltraWinToolbars elements within the UltraWinToolbars.

There are many 4Test methods available with these new classes, some of which are available for recording.

Support for Infragistics .dll Files

This functionality is supported only if you are using the Classic Agent.

Silk Test Classic supports the following Infragistics .dll files (assemblies):

- Infragistics.Win.UltraWinGrid
- Infragistics.Win
- Infragistics.Shared
- Infragistics.Win.UltraWinToolbars

Silk Test Classic supports the following Infragistics NetAdvantage versions:

- CLR 1.1:
  - 4.3.20043.27
  - 5.1.20051.37
  - 5.2.20052.27
  - 5.3.20053.50
  - 6.1.20061.28
  - 6.2.20062.34
  - 6.3.20063.53
- CLR 2.0:
  - 7.3.20073.38

The CLR 2.0 library differs from the CLR 1.1 library. As a result, .dll files cannot be shared between CLR 2.0 and 1.1.

For information about new features, supported platforms and versions, known issues, and work-arounds, refer to the *Release Notes*.

If you are testing an application that uses a different version of these .dll files, you must modify either the application configuration or the machine configuration file with instructions that redirect Silk Test Classic to use the version that your application supports. Silk Test Classic can be configured post-installation to support NetAdvantage version variations in the last two version fields, for example, x.x.20061.28.

If you do not redirect Silk Test Classic to use the proper .dll files, you may experience problems with Silk Test Classic being unable to record Infragistics controls correctly, such as the following:

- Recording `Typekeys`() on a `DataGrid` instead of correctly recording `ClickCell`() and `SetCell`().
- Not recognizing the buttons inside the Toolbar when your mouse is pointing to the Toolbar.

Note:

- Since applications can specify which specific directory is used to load .dlls, the binding method described below will work only if it is not overwritten or ignored.
- There may be other backward compatibility issues with Infragistics .dlls. For more details about these issues, refer to the documentation of your .NET Framework SDK.

To redirect the .dlls, you must:

1. Choose whether to configure the machine or the application.
2. Choose whether to configure by editing a file or through the Control Panel. You can use either of these methods to configure the machine or the application.

Configuring .NET

This functionality is supported only if you are using the Classic Agent.

Now that you have decided whether to change the configuration for just your application under test or for your entire machine, you must decide how you want to make the change. You can use either of these methods, regardless of whether you are changing the configuration for the application or for the machine:

- Edit a configuration file directly.

- Use the Control Panel to edit the configuration file.

Editing a Configuration File Directly

This functionality is supported only if you are using the Classic Agent.

To edit the configuration file directly:

1. Open the correct configuration file:

To configure…	Open...	Located In...
just the application	<application name>.exe.config file	the same directory as the <application name>.exe
the whole machine	machine.config file	the Windows directory

2. If one of these configuration files do not exist, you must create it.

   Pay attention to the location of the configuration file, as described in the table above.
3. Copy the sample section from above and paste the information into the configuration file to redirect the .dlls.
4. Edit the sample so that your version number displays in place of 4.3.20043.54 where `newVersion="4.3.20043.54"`.
5. Save the configuration file.

When you restart your application and begin testing, the. dlls that Silk Test Classic requires will be redirected to use the .dlls that your application requires.

Using the Control Panel to Edit Your Configuration File

This functionality is supported only if you are using the Classic Agent.

To use the control panel to edit your configuration file:

1. Click **Start** > **Control Panel** > **Administrative Tools** > **Microsoft .NET Framework 1.1 Configuration**. The **.NET Configuration 1.1** dialog box displays.
2. In the **.NET Configuration 1.1** dialog box, you should see one or more of the Infragistics .dlls.
   - If you do, you can skip to step #7.
   - If you do not, you can either add all four Infragistics .dlls to your machine or just the application you are testing.
3. To add the .dlls to your computer, navigate to `My Computer/Configured Assemblies`.
4. To add the .dlls to your .NET application, navigate to `Applications/<your application name>/Configured Assemblies`.
5. Right-click `Configured Assemblies` and click **Add**.
6. On the **Configure an Assembly** dialog box, you select assemblies to add.

   If you need more information about adding assemblies, refer to the information provided by Microsoft on the *.NET Framework Configuration Tool*.
7. After you add the assemblies, you must change the binding policy for each of them by right clicking the assembly name and selecting **Properties**.
8. In the **Properties** dialog box, click the **Binding Policy** tab.
9. Click the **Requested Version** column and enter the requested version that Silk Test Classic uses, 4.3.20043.27.
10. Click the **New Version** column and enter the new version, which is the .dll your application requires.
11. Repeat Steps #7-10 for each of the other assemblies.
12. Click **OK** to save the information.

When you restart your application and start testing, the .dlls that Silk Test Classic requires are redirected to use the .dlls that your .NET application requires.

Choosing to Configure the Machine or the Application

This functionality is supported only if you are using the Classic Agent.

You may edit the configuration file for just your application or you may edit the configuration file for your entire machine. If you do not have a configuration file, you may create one; see the sample file if you are interested in creating one.

If you want to change the application configuration, you need to do so on the machine where the application .exe is, in the same directory as that .exe. This approach establishes a "default" configuration that is associated with the application.

If you want to change the machine configuration, you need to do so on the machine where the application will run (which is the same location as the Silk Test Agent). This approach establishes a configuration that will override any application configuration that may exist.

In either case, you can use the direct file editing or the Control Panel to edit the configuration file.

Recording Actions on the Infragistics Toolbars

This functionality is supported only if you are using the Classic Agent.

Silk Test Classic records actions on the Infragistics UltraWinToolbars with a prefix of "Toolbar". All other components, except for .NET SWF controls, are mapped as a CustomWin.

Class	Silk Test Classic Records	Description
`ToolbarButton`	`Click()`	You click a button on the toolbar.
`ToolbarComboBox`	`Select()`	You choose an item from the list box.
	`SetText()`	You type in the text area.
`ToolbarList`	`Select()`	You select an item.
	`Unselect()`	You unselect an item.
`ToolbarPopup`	`Select()`	You selected a new item.
`ToolbarPopupMenu`	`DropDown()`	You interact with the menu.
`ToolbarTextBox`	`SetText()`	You type in the text area.

Record Class, Window Declarations, and Window Identifiers

Silk Test Classic records the methods and properties of any UltraWinToolbars control or component when you Record/Class/Scripted on children of the `UltraWinToolbars` class. Likewise, Record Window Declarations displays the `UltraWinToolbars` class declaration and all the supported components and controls that are children of the UltraWinToolbars control.

When you hover the cursor over a component or a control in UltraWinToolbars, Record Window Identifiers displays that component or control as a child of an instance of the `UltraWinToolbars` class.

Testing Java AWT/Swing Applications with the Classic Agent

Silk Test Classic provides built-in for testing stand-alone Java applications developed using supported Java virtual machines and for testing Java applets using supported browsers. You must configure Silk Test Classic Java support before using it. When you configure a Java AWT/Swing application or applet, Silk Test Classic automatically provides support for testing standard Java AWT/Swing controls. You can also

test Java SWT controls embedded in Java AWT/Swing applications or applets as well as Java AWT/Swing controls embedded in Java SWT applications.

You can test Java AWT/Swing applications using the Classic Agent or the Open Agent. When you create a new Java AWT/Swing project, Silk Test Classic uses the Open Agent by default. However, you can use both the Open Agent and the Classic Agent within a single Java AWT/Swing environment. Certain functions and methods run on the Classic Agent only. As a result, if you are running an Open Agent project, the Classic Agent may also open because a function or method requires the Classic Agent.

When you are using the Classic Agent to test Java AWT/Swing applications, Silk Test Classic uses the Sun JDK by default.

For information about new features, supported platforms and versions, known issues, and work-arounds, refer to the *Release Notes*.

Testing Standard Java Objects and Custom Controls

Any single Java application or applet may contain a mix of standard and custom Java objects. With Java support, you can test both types of visible Java objects in applications and applets that you develop using the Java Development Kit (JDK)/Java Runtime Environment (JRE).

Standard Java objects are often defined in class libraries. The Java support of Silk Test Classic lets you record and play back tests against standard controls by providing 4Test definitions for many Java classes defined in the following class libraries:

- Abstract Windowing Toolkit (AWT)
- Java Foundation Class (JFC), which includes the Swing set of GUI components
- Standard Widget Toolkit (SWT)
- Symantec Visual Café Itools (only for the Classic Agent)

If you are using the Classic Agent, you can use the `setName`("<desiredwindow ID>") method to create a window ID that Silk Test Classic will detect. `setName`() is a method inherited from class `java.awt.Component`, so it should work for most, if not all, of the Java classes that Silk Test Classic can detect. If you are using the Open Agent, the equivalent of the `setName` method is the `Name` property of the `AWTComponent` class.

By contrast, custom controls often use native properties and native methods written in Java. Increasingly, custom controls also take the form of JavaBeans, which are reusable platform-independent software components written in Java. Developers frequently design custom controls to achieve functionality that is not available in standard class libraries. You can test custom Java objects, including JavaBeans, using the Silk Test Classic Java support.

The Silk Test Classic approach to testing custom Java objects is to give you direct access to their native methods and properties. A major advantage of this methodology is that it obviates the need to write your own native methods.

The procedure for testing custom Java objects is simple: Record a class for the custom control, then save the class definition in an include file. The class definition includes the native methods you can call and native properties you can verify from your 4Test script.

The predefined property sets supplied with Silk Test Classic have not been customized for Java; however, you can modify these property sets. For additional information about editing existing property sets or creating new property sets, see *Creating a Property Set*.

Recording and Playing Back JFC Menus

For Sun JDK v1.4 or later, Silk Test Classic can record and play back regular menus that conform to the Windows standard, as well as JFC heavyweight and lightweight pop-up menus.

Recording and Playing Back Java AWT Menus

Unlike JFC menus, AWT menus are not conform to the Java component-container paradigm. Therefore, their behavior is different than that of the JFC menus, and is independent of the JVM version. Silk Test Classic can record regular AWT menus for all versions of the JDK.

For context menus that are conform to the Windows standard, which means that they can be opened with a right-click, Silk Test Classic can play back, but not record, the context menus for all versions of the JDK.

For AWT popup menus that are not conform to the Windows standard, Silk Test Classic cannot record or play back for all versions of the JDK. The `JavaAwtPopupMenu` class is available for playback only. Silk Test Classic is not able to record it and you must hand script any interaction with such a menu.

Object Recognition for Java AWT/Swing Applications

Java AWT/Swing applications support hierarchical object recognition and dynamic object recognition. You can create tests for both dynamic and hierarchical object recognition in your test environment. Use the method best suited to meet your test requirements.

When you record a test case with the Open Agent, Silk Test Classic creates locator keywords in an INC file to create scripts that use dynamic object recognition and window declarations.

Using custom class attributes becomes even more powerful when it is used in combination with dynamic object recognition.

To test Java AWT/Swing applications using hierarchical object recognition, record a test for the application that you want to test. Then, replay the tests at your convenience.

Supported Controls for Java AWT/Swing Applications

For a complete list of the record and replay controls available for Java AWT/Swing testing with the Open Agent, refer to the *Java AWT and Swing Class Reference* in the *4Test Language* section of the Help.

For a complete list of the record and replay controls available for Java AWT/Swing testing with the Classic Agent, refer to the *Java AWT Classes for the Classic Agent* and the *Java JFC Classes* in the *4Test Language* section of the Help.

Java AWT Classes for the Classic Agent

This section lists classes for Java AWT handling with the Classic Agent.

Supported Java Virtual Machines

This functionality is supported only if you are using the Classic Agent.

Silk Test Classic supports the following Java virtual machines (JVMs) for testing standalone Java applications:

- Sun Microsystems' Java Development Kit (JDK) (including the Java AppletViewer)
- Sun Microsystems' Java Runtime Environment (JRE)
- IBM's Java Development Kit (JDK)
- Symantec Visual Café

Supported Browsers for Testing Java Applets

Silk Test Classic supports the following browsers for testing Java applets:

- For the Classic Agent: Internet Explorer 7 using the Java plug-in.
- For the Open Agent: All supported versions of Internet Explorer and Mozilla Firefox.

For information about new features, supported platforms and versions, known issues, and work-arounds, refer to the *Release Notes*.

Overview of JavaScript Support

Silk Test Classic provides support for executing JavaScript code within a Web application. You can test applications that include JavaScript by performing one of the following tasks:

- Configuring an xBrowser application that uses the Open Agent.
- Enabling extensions for a generic application that uses the Classic Agent.

The type of agent that you use determines the classes that are available for you to create tests with.

As a best practice, we recommend using xBrowser rather than the Web application because xBrowser uses the Open Agent and dynamic object recognition.

We recommend recording test cases using dynamic object recognition. Then, replay the tests at your convenience.

JavaScript Support for the Open Agent

With the Open Agent, you can use `ExecuteJavaScript` to test anything that uses JavaScript. You can:

- Call any function already contained in a document.
- Inject new functions into a document and call them.
- Trigger Document Object Model (DOM) events, such as calling `onmouseover` directly for an element.
- Modify the DOM tree.

JavaScript Support for the Classic Agent

If you use the Classic Agent, you can test JavaScript using the following methods:

- `ExecLine`
- `ExecMethod`
- `ExecScript`

Support for JavaBeans

This functionality is supported only if you are using the Classic Agent.

Many Java components are implemented as JavaBeans. Each JavaBean must provide an associated `BeanInfo` class that describes the capabilities of the component, including its methods and properties.

Our Java support provides a 4Test method called `GetBeanInfo` method that you can use in scripts to access information from the BeanInfo structure associated with JavaBeans in the applications or applets you are testing.

Using GetBeanInfo, you can access the following information about JavaBeans:

- Name of the JavaBean.
- Methods.
- Properties.
- Events supported by the JavaBean.
- Methods associated with event listeners supported by the JavaBean.
- Size of the icon associated with the JavaBean.

Classes in Object-Oriented Programming Languages

Classes are the core of object-oriented programming languages, such as Java. Applets or applications developed in Java are built around objects, which are reusable components of code that include methods and properties. Methods are tasks that can be performed on objects. Properties are characteristics of an object that you can access directly.

Each object is an instance of a class of objects. GUI objects in Java, for example, may belong to such classes as `Menu`, `Dialog`, and `Checkbox`. Each class defines the methods and properties for objects that are part of that class. For example, the `JavaAwtCheckBox` class defines the methods and properties for all Java Abstract Windowing Toolkit check boxes. The methods and properties defined for `JavaAwtCheckboxes` work only on these check boxes, not on other Java objects.

Configuring Silk Test Classic to Test Java

This section describes how to configure Silk Test Classic to test Java applications.

Prerequisites for Testing Java Applications

To test…	Install…
standalone Java applications	JDK/JRE
Java applets	JDK, supported browser, and plug-in (if necessary)
Java applets using the Java Applet Viewer	JDK and plug-in

Note:

- When you are using the Classic Agent, Java support is configured automatically when you use **Enable Extensions** in the **Basic Workflow** bar.
- When you are using the Open Agent, Java support is configured automatically when you use **Configure Applications** in the **Basic Workflow** bar.
- You can use the **Basic Workflow** bar to configure your application or applet or manually configure Java Support. If you choose to manually configure Java support, you may need to change the CLASSPATH environment variable. For JVM/JRE 1.2 or later, you must also copy the applicable Silk Test Classic .jar file to the `lib\ext` folder of your `JVM/JRE`.

For information about new features, supported platforms and versions, known issues, and work-arounds, refer to the *Release Notes*.

Enabling Java Support

There are several ways to enable Java support for testing standalone Java applications. Pick the scenario that fits your runtime environment and testing needs.

Scenario	How to enable Java support
You need to test your application on the 32-bit Windows host machine using a JVM that is invoked from a `java.exe`, `jre.exe`, `jrew.exe`, or `vcafe.exe` executable, including: • JDK/JRE • Symantec Visual Café (only if you are using the Classic Agent)	Enable the default Java application.
You need to test your application on a remote 32-bit Windows machine using a JVM that is invoked from	Install Silk Test Classic on your remote machine and enable the default Java application on your host machine.

Scenario	How to enable Java support
a`java.exe`, `jre.exe`, `jrew.exe`, or `vcafe.exe` executable.	
You need to test your application on the 32-bit Windows host machine using a JVM that is not invoked from a `java.exe`, `jre.exe`, `jrew.exe`, or `vcafe.exe` executable.	Enable a new Java application.
You need to test your application on a remote 32-bit Windows machine using a JVM that is not invoked from a `java.exe`, `jre.exe`, `jrew.exe`, or `vcafe.exe` executable.	Install Silk Test Classic on your target machine and enable a new Java application.

Configuring Silk Test Classic Java Support for the Sun JDK

When you are using the Classic Agent, Java support is configured automatically when you use **Enable Extensions** in the **Basic Workflow** bar. When you are using the Open Agent, Java support is configured automatically when you use **Configure Applications** in the **Basic Workflow** bar. We recommend that you use the basic workflow bar to configure your application or applet, but it is also possible to manually configure Java support.

If you incorrectly alter files that are part of the JVM extension, such as the `accessibility.properties` file, in the `Java\lib` folder, or any of the files in the `jre\lib\ext` directory, such as `SilkTest_Java3.jar`, unpredictable behavior may occur. There are two methods for configuring Silk Test Classic Java Support:

- Manually configuring Silk Test Classic Java support.
- Configuring Standalone Java Applications and Java Applets.

Manually Configuring Silk Test Classic Java Support

If you want to enable Java support manually, or if the **Basic Workflow** does not support your configuration, perform the following tasks:

If you are using the Classic Agent	Click **Options** > **Extensions** to open the **Extensions** dialog box and enable Java applet or application support by checking or un-checking the **Java** check box for your application. The **Java** check box can be checked or un-checked for a specific application or applet. If you check or un-check this check box for one extension, it is checked or un-checked for all.
If you are using the Open Agent	Click **Options** > **Application Configurations** to open the **Edit Application Configuration** and add a standard test configuration for your Java application.

Configuring Standalone Java Applications and Java Applets

In order for Silk Test Classic to recognize Java controls, you may need to change the *CLASSPATH* environment variable. For JVM/JRE 1.3 or later, you must also copy the applicable `SilkTest.jar` file to the `lib\ext` folder of your JVM/JRE. The `SilkTest.jar` file is located in the `<SilkTest Install Directory>\JavaEx` directory.

1. If you are using JVM/JRE 1.3 or later, use `SilkTest_Java3.jar`.

   For information about new features, supported platforms and versions, known issues, and work-arounds, refer to the *Release Notes*.
2. For Java 1.3 or later, you should not set a specific classpath variable – instead, use the default `CLASSPATH=.;`. Copy the `SilkTest_Java3.jar` file to the `lib\ext` folder of your JVM/JRE, and remove any previous Silk Test Classic JAR files.
3. In the Silk Test Classic folder, rename the file `access3.prop` to `accessibility.properties` and copy it to the `Java…\lib` folder.

4. Finally, `qapjconn.dll` and `qapjarex.dll` are new DLL files that must be installed in the `Windows\System32` directory.

   The Silk Test Classic installer places these files in the `Windows\System32` folder, and also places copies of these files in the `SilkTest\Extend` folder. If the default directory for your library files is in a location other than `Windows\System32`, you must also copy `qapjconn.dll` and `qapjarex.dll` to the alternate location.

Note:

- The Java recorder does not support applets for embedded Internet Explorer browsers(AOL).
- It is not possible, using normal configuration methods, to gain recognition of Java applications that use .ini files to set the environment. However, if your application sets the Java library path using the JVM launcher directive `Djava.library.path=< path >`, you can obtain full recognition by copying `qapjarex.dll` and `qapjconn.dll` from the `System32` directory into the location pointed to by the JVM launcher directive.

Configuring Silk Test Classic for JBuilder or JDeveloper

This functionality is supported only if you are using the Classic Agent.

To test applications running in JBuilder or Oracle JDeveloper, perform the following steps:

Note: These steps refer to JBuilder; the same procedure applies to JDeveloper.

1. Determine which version of Java you are using and copy the `SilkTest_Java3.jar` file from `<SilkTest installation directory>\extend to <JBuilder installation directory>\lib`.
2. Edit the `JBuilder.ini` file to include the path and name of the `SilkTest_Java3.jar` file in the CLASSPATH.
3. Inside JBuilder, bring up the application you want to test.
4. Inside JBuilder, click **Project** > **Properties** and add the `SilkTest_Java3.jar` file to the list of libraries and call the library Silk Test.
5. Add the Silk Test library to the particular application you are testing.
6. Within Silk Test Classic, make sure the Java extension in enabled. JBuilder uses the `javaw.exe` to run Java. `Javaw.exe` is part of the Silk Test Classic Java extension.

Java Security Privileges Required by Silk Test Classic

Before reviewing your security privileges, make sure that you have configured Silk Test Classic Java support.

Required security privileges

In order to load the Silk Test Classic Java support files, Silk Test Classic must have the appropriate Java security privileges. At a minimum, Silk Test Classic requires the following abilities:

- Create a new thread.
- Access members of classes loaded by your application.
- Create, read from, and write to file on a local drive.
- Access, connect, listen and send information through sockets.
- Access AWT event queue.
- Access system properties.

For standalone applications, the security policy is set in the `java.security` file which is located in `JRE/lib/security`. By default this file contains the following line:

```
Policy.provider = sun.security.provider.PolicyFile
```

which means that the standard policy file should be used. The standard policy file, `java.policy`, is located in the same folder, `JRE/lib/security`. It contains the following code that gives all necessary permission to any file located in `lib\ext` directory:

```
// Standard extensions get all permissions by default
grant codeBase "file:${java.home}/lib/ext/*"{permission
java.security.AllPermission;};
```

Silk Test Classic has the necessary privileges, if the `SilkTest_Java3.jar` file is in this directory and the JVM runs with the default set of security permissions.

If you have changed the Java security policy

The system administrator can change security policy by starting the JVM with the following option:

```
java _Djava.security.policy=Myown.policy MyApp
```

In this case the custom policy file `Myown.policy` should contain the following lines that grant all permission to classes from the `lib\ext` directory:

```
grant codeBase "file:${java.home}/lib/ext/*"{permission
java.security.AllPermission;};
```

The default `java.policy` may also be changed implicitly, for example, when the application uses an RMI server with the custom RMISecurityManager and the client security policy. In cases like these, the client security policy should grant all required permissions to Silk Test Classic by including the code listed above.

In some cases, setting these permissions may not provide Silk Test Classic with the necessary security privileges. The cause of the problem may be that permissions are frame specific. So if Silk Test Classic runs in the context of frames (thread) in which it does not have the necessary permissions, it may fail. In cases like this in which the client does not trust code running in the context of the AWT event thread, you need to set the parameter `ThreadSafe=False` in the `javaex.ini` in the `<Silk Test installation>/extend` directory. This prevents the Silk Test Classic Java code from running in the context of the AWT event thread, preserving permissions granted to Silk Test Classic, but could make the GUI less responsive.

Disabling Java Support

This functionality is supported only if you are using the Classic Agent.

To disable Java support:

1. Click **Options** > **Extensions** in the menu bar.
2. In the **Application** column, click the Java application that you want to disable and uncheck the **Java** check box, as follows:

If you installed ...	Uncheck Java check box for ...
Java Development Kit, Java Runtime Environment, or Symantec Visual Café	Java Application
Any other Java virtual machine	<name of executable file>.exe

3. Click **OK**.

Enabling Java Applications and Plugins

This section describes how you can enable Java applications and plugins.

Enabling Extensions for the Default Java Application

This functionality is supported only if you are using the Classic Agent.

To enable the default Java application:

1. In the Silk Test Classic menu bar, click **Options** > **Extensions**. The **Extensions** dialog box opens.
2. In the **Application** column, click **Java Application** and check the **Java** check box.
3. Click **OK**.

Enabling a New Extension for a Java Application

This functionality is supported only if you are using the Classic Agent.

You must enable a new Java application in both the **Extension Enabler** and the **Extensions** dialog box.

To enable a new extension for a Java application:

1. Click **Start** > **Programs** > **Silk** > **Silk Test** > **Tools** > **Extension Enabler**.
   If you are running your Java application on a remote 32-bit Windows machine, launch `extinst.exe`, which is in the Silk Test Classic installation directory on the remote machine.
2. On the **Extension Enabler** dialog box, click **New**.
3. Click to navigate to the location of the JVM executable you want to hook into, and then click **OK**.
4. Check the **Java** check box for the executable you just added, leave **Primary Extension** set to `(None)`, and then click **OK**.
5. Start Silk Test Classic.
6. In the Silk Test Classic menu bar, click **Options** > **Extensions**. The **Extensions** dialog box opens.
7. On the **Extensions** dialog box, click **New**.
8. Click to navigate to the location of the JVM executable you want to hook into, and then click **OK**.
9. Check the **Java** check box for the executable you just added, leave **Primary Extension** set to `(None)`, and then click **OK**.
10. Start your Java application.

Enabling Use of Sun Java Plug-In

This functionality is supported only if you are using the Classic Agent.

To use the JRE for running Java applets in any of our supported browsers, you must enable use of a plug-in and your applet must explicitly request to run in the Java plug-in.

To enable use of a plug-in:

1. Make sure you have installed the Java plug-in for each of the supported browsers you want to use for testing.
2. In the Silk Test Classic menu bar, click **Options** > **Extensions**. The **Extensions** dialog box opens.
3. In the **Application** column, click one of the browsers for which you installed the Java plug-in. If not enabled, then select **Enabled** from the list box in the **Primary Extension** column.
4. In the **Options** section of the **Extensions** dialog box, click **Extension**.
5. Check the **Enable use of Java plug-in** check box, and then click **OK**.
6. Repeat the steps 3–5 for the other supported browsers.
7. Click **OK** to close the **Extensions** dialog box.

You can designate the JRE that the plug-in uses by clicking **Start** > **Settings** > **Control Panel** > **Java Plug-in** and selecting the JRE on the **Advanced** tab of the **Java Plug-in Properties** dialog box. Make sure that the appropriate Silk Test .jar file is accessible to the plug-in. For additional information, see *Configuring Standalone Java Applications and Java Applets*.

Configuring Silk Test Classic to Support a Java Application Launched from a .lax File

This functionality is supported only if you are using the Classic Agent.

If you are running your Java application from a launcher application executable (*.lax), you must add the appropriate Silk Test Classic .jar file to the CLASSPATH inside the .lax file.

To find out which version of Java the .lax file is using, open a Command Prompt, type the path to Java that displays in the .lax file, and then type `java –version`. If it is using Java 1.3 and later, add the `SilkTest_Java3.jar` to the CLASSPATH inside the .lax file.

If you are running your application through a .lax file, make sure you add the .lax file as a new extension in Silk Test Classic.

Testing Java Applications and Applets

Silk Test Classic supports testing Java applications that use the Sun JDK. By default, Silk Test Classic uses the Sun JDK with the Classic Agent.

Preparing for Testing Stand-Alone Java Applications and Applets

To prepare for testing stand-alone Java applications using Silk Test Classic:

1. In the **Basic Workflow** bar:
   - If you are using the Classic Agent, enable extensions for Java support for application and applet testing.
   - If you are using the Open Agent, configure your Java application.
2. If you do not plan to test applets during the session, disable browser support.
3. Identify the custom controls in your Java application.
4. If you are testing Java applications with the Classic Agent, enable the recovery system.
5. Record classes for any custom controls you want to test in a new class include file or in your test frame file.

   If Silk Test Classic does not recognize some of your custom objects, see *Recording Classes for Ignored Java Objects*.
6. If you are testing standalone Java applications with the Classic Agent, record window declarations for your Java application, including declarations for any new classes you defined.

Testing Browser-Based Java Applications

This functionality is supported only if you are using the Classic Agent.

The following procedure applies to JDK 1.4 applications:

1. Make sure the Java extension and appropriate browsers are enabled.
2. Make sure that Silk Test Classic is configured properly.
3. Open the file `<SilkTest installation directory>\Silk\SilkTest\JavaEx\JFC \JPI_index.html` in the browser you want to use for testing.
4. Start Silk Test Classic and record as usual.

If you are testing JDK 1.4 browser-based applications and the application throws security exceptions, make sure that the `SilkTest_Java3.jar` file was copied to the correct location. Installing security certificates in the browser does not resolve this issue; the `SilkTest_Java3.jar` must be in the correct location.

Indexed Values in Test Scripts

4Test methods use a 1-based indexing scheme, where the first indexed value is stored in position 1. Native Java methods use a 0-based indexing scheme, where the first indexed value is stored in position 0. This incompatibility can create challenges in coding test scripts that access indexed values using both native methods and 4Test methods.

Multitags and Java Applications

This functionality is supported only if you are using the Classic Agent.

Silk Test Classic permits multitags when recording window declarations for Java applications. You are no longer restricted to just the caption. The exception is that the default for top-level windows is only the caption. The reason is that the window ID usually defaults to the class name with an index, and for top-level windows, the index ("#1") leads to misidentification. If the window ID will be unique for a top-level window, then you can highlight that window in the **Record Window Declarations** dialog box and check **Window ID** in the **Tag Information** box.

When to Use 4Test Versus Native Java Controls

Silk Test Classic provides a predefined set of Java classes, including Abstract Windowing Toolkit (AWT) controls, Java Foundation Class (JFC) controls, and Symantec Visual Café controls. To test these controls, you can use their inherited 4Test methods. Inherited methods are the 4Test methods associated with the class from which the control is derived.

For custom Java controls, we provide access to native Java methods, as defined in JDK 1.1.2 or later. You can also access native methods for predefined Java classes.

When both 4Test methods and native methods are available for all controls you want to test, we recommend using 4Test methods in your test scripts. 4Test provides a richer, more efficient set of methods that more closely mirror user interaction with the GUI elements of an application. For the `JavaAwtPushButton`, for example, use the 4Test methods associated with the `PushButton` class.

When you must use native methods for controls that are not supported in 4Test, refer to the Java API documentation to gain a full understanding of how the native method works. For example, 4Test methods and native Java methods use incompatible array indexing schemes so you must use caution when accessing indexed values.

Note: We recommend not to mix 4Test and native methods because of incompatibilities between Java and 4Test.

Predefined Class Definition File for Java

The file `javaex.inc` includes 4Test class definitions for the following controls:

- Abstract Windowing Toolkit (AWT) controls
- Java Foundation Class (JFC) library controls
- Symantec Visual Café Itools controls
- Java-equivalent window controls

We provide these class definitions to help you quickly get started with testing your Java applications. You can record additional classes if you determine that additional controls are necessary to test your application.

The file `javaex.inc` is installed in the `Extend` subdirectory under the directory where you installed Silk Test Classic.

Web Applications and the Recovery System

This functionality is supported only if you are using the Classic Agent.

When the recovery system needs to restore the base state of a Web application that uses the Classic Agent, it does the following:

1. Invokes the default browser if it is not running.
2. Restores the browser if it is minimized.
3. Closes any open additional browser instances or message boxes.

4. Makes sure the browser is active and is not loading a page.
5. Sets up the browser as required by Silk Test Classic.

   The recovery system performs the next four steps only if the `wMainWindow` constant is set and points to the home page in your application.
6. If `bDefaultFont` is defined and set to TRUE for the home page, sets the fonts.
7. If `BrowserSize` is defined and set for the home page, sets the size of the browser window.
8. If `sLocation` is defined and set for the home page, loads the page specified by `sLocation`.
9. If `wMainWindow` defines a `BaseState` method, executes it.
10. For additional information, see *DefaultBaseState and the wMainWindow Object*.

To use the recovery system, you must have specified your default browser in the **Runtime Options** dialog box. If the default browser is not set, the recovery system is disabled. There is one exception to this rule: You can pass a browser specifier as the first argument to a test case. This sets the default browser at runtime. For more information, see *BROWSERTYPE Data Type*.

The constant `wMainWindow` must be defined and set to the identifier of the home page in the Web application for the recovery system to restore the browser to your application's main page. This window must be of class `BrowserChild`. When you record a test frame, the constant is automatically defined and set appropriately. If you want, you can also define a `BaseState` method for the window to execute additional code for the base state, for example if the home page has a form, you might want to reset the form in the `BaseState` method, so that it will be empty at your base state.

On Internet Explorer 7.x and 8.x, when recording a new frame file using **Set Recovery System**, by default Silk Test Classic does not explicitly state that the parent of the window is a browser. To resolve this issue, add the "parent Browser" line to the frame file.

Java Extension Options

This functionality is supported only if you are using the Classic Agent.

There are several options that you can set for the Java extension by manually editing the `javex.ini` file. These options apply to `SilkTest_Java3.jar` only.

The settings go in the `[Options]` section of `javaex.ini` unless otherwise noted. These are optional and you do not have to include the setting in your `javaex.ini` if you want the default behavior.

`AwtCompTreeLockTimeout`

Default is -1. This option applies only to AWT-based applications that contain AWT popup menus. It should not be used with other applications because it will slow performance. If your application contains AWT popup menus, then you should set this value to a positive number representing milliseconds (for example, 1000, representing 1 second) in order to prevent Silk Test Classic from freezing when interacting with AWT popup menus.

It is strongly recommended that you use `PopupSelect()` rather than `JavaAwtPopupMenu::Pick()`.

`EnumAwtPeers`

Default is TRUE. This option applies only to AWT-based applications. It will not affect applications based on JFC (Swing). Setting the option to FALSE may significantly improve playback speed. However, it may change the window identifier hierarchy, which may cause existing automation to fail. The default value is TRUE, for backward compatibility with previous versions of Silk Test Classic.

The option controls whether Silk Test Classic searches for peers of AWT controls. Peers are windows that represent counterparts to Java controls, but are not Java classes themselves. Examples of peer classes that you may see in your window declarations are `SunAwtFrame` or `SunAwtCanvas`. If the option is set to FALSE, Silk Test Classic does not recognize peer classes, and so does not include them in the window identifier hierarchy. You should not set the option to FALSE if any of your window declarations mention

such classes; otherwise you will have to change your existing test scripts to accommodate the modified window identifier hierarchy.

If the option is set to FALSE, Silk Test Classic will, as always, recognize most custom Java components without having to add their class names to the `[ClassList]` section of `javaex.ini`. There are two exceptions: custom components that have children, and custom components that are derived from ignored container classes such as `Panel`. In order to enable Silk Test Classic to recognize a component in one of those two categories, you must add the class name to the `[ClassList]` section of `javaex.ini`. See *Recording classes for ignored Java objects* for more information. In order to identify the classes to add to the `[ClassList]` section, you may need to enable **Show all classes** on the **Record Class** dialog box and examine the resulting window declarations. You must remember to disable **Show all classes** before you modify the `[ClassList]` section. Never play back scripts with **Show all classes** enabled, or performance will slow down greatly.

`IncludeInstanceNumber`

Default is FALSE. The Java extension uses the component name as the window ID form of the tag. For most JFC (Swing) components, the component name is the component class name. For AWT components, however, the component name has the form: base name + instance number. The base name resembles the component class name. For example, AWT Button has a base name of "button", while AWT Dialog has a base name of "dialog". The instance number reflects the order in which this instance of the component was created by the JVM. The order of creation, and therefore the instance number, may vary depending on the sequence of actions performed against the application. This may cause the window ID of a specific window to vary between runs of a testcase, or even within a testcase.

For example, the first time that you invoke a specific popup window, its window ID will be "dialog0" (base class = "dialog"; instance number = 0). But if you discard this popup window and then invoke it again, the window ID will be "dialog1". This will invalidate the window declaration if the window ID is the only tag form being used for that window. If `IncludeInstanceNumber` is set to FALSE (the default value), then the instance number will be omitted from the window ID, and the window ID will consist only of the base class. If there are multiple windows with the same base class, then the window ID's will have the form "baseClass[n]", following SilkTest's standard convention for distinguishing between multiple controls with the same tag value.

Setting the option to TRUE will reintroduce the Java instance number into the window ID tag form, which usually will make the window ID less robust. However, the option is included for purposes of backward compatibility.

`ParentPopupToInvoker`

Default is TRUE. This option specifies how the parent of a popup menu should be determined. When the option is set to TRUE (default), the parent of the popup menu will be the window that invoked it (or the first ancestor of the invoking window that is not ignored by the Java extension). When the option is set to FALSE, the parent of the popup menu will be the container of the popup menu (or the first ancestor of the container that is not ignored by the Java extension).

Recognition of popup menus should be more robust using the default value, TRUE, because that value will eliminate changes to the window hierarchy that occur when the container of a popup menu changes because the size or placement of the menu changes. Prior to the introduction of this option, the container was always used as the parent of the popup menu, corresponding to an option value of FALSE.

For example, for JFC (Swing) menus, the container by default is a panel. If the panel is ignored by the Java extension, as usually will be the case, and if Silk Test Classic is using the container as the parent (option value is FALSE), then Silk Test Classic will recognize the parent as the `JavaMainWin` that contains the menu. However, if the menu extends beyond the boundary of the `JavaMainWin`, then the container, and therefore the parent (if option value is FALSE) will be seen as a popup window (usually a `JavaDialog`). So if the option value is FALSE, then the parent of the menu may change depending on the size of the menu, causing the window declaration to be invalid. If Silk Test Classic is using the invoking window as the parent (meaning that this option is TRUE), however, then the parent will not change because the same

window (usually the JavaMainWin) invokes the menu whether or not the menu lies within the boundaries of the JavaMainWin.

The default value, TRUE, also allows Silk Test Classic to distinguish between popup menus that are invoked in different contexts, for example by clicking different buttons in the same toolbar. The parent of the popup menu is the button that was clicked to bring up the menu. In contrast, setting the value to FALSE may cause popup menus invoked by clicking different buttons in a toolbar to be seen as children of the same JavaMainWin, and therefore to be seen as part of the same popup menu.

TableGetValueAtOnly

Default is FALSE. This option, which is part of the [JavaJFCTable] section of the javaex.ini file, determines how Silk Test Classic obtains the cell text for JavaJFCTable controls. The default value, FALSE, uses the cell renderer object to find the cell text for non-string cell contents, such as for a cell that contains a custom component that displays text but does not implement toString(). If Silk Test Classic does not return the correct table cell text with the value set to FALSE, then change the value to TRUE. Setting the value to TRUE causes Silk Test Classic to use the cell data object rather than the renderer object.

TreeNodeValueHasPrecedence

Default is FALSE. This option, which is a part of the [JavaJFCTreeView] section of the javaex.ini file, determines how Silk Test Classic obtains the node text for JavaJFCTreeView controls. With this option set to FALSE, Silk Test Classic attempts to find meaningful text for the tree node by relying more heavily on the Java API. If Silk Test Classic does not return the correct tree node text with the value set to FALSE, then change the value to TRUE. Setting the value to TRUE gives precedence to the string representation of the value of the node, even if that value does not yield apparently meaningful text.

UseExpandButton

Default is FALSE. This option lets you specify how Silk Test Classic should expand/collapse nodes in a JavaJFCTreeView. The default value, FALSE, specifies that Silk Test Classic should double-click nodes to expand or collapse them. This value works for default implementations of a JavaJFCTreeView. However, if your implementation of a JavaJFCTreeView does not use a double-click to expand or collapse nodes, then set the value to TRUE, which directs Silk Test Classic to click on the **Expand** button (usually a small square with a +/- sign).

Setting Java Extension Options Using the javaex.ini File

This functionality is supported only if you are using the Classic Agent.

You can set several options for the Java extension by manually editing the javaex.ini file. These options apply to SilkTest_Java3.jar only

To set Java extension options using the javaex.ini file:

1. Close Silk Test Classic and the AUT, if they are open.
2. Open javaex.ini, located in the extend subdirectory of the directory where you installed Silk Test Classic.

   If you are using a Silk Test Classic Project, the applicable javaex.ini file is in the project, not in the Silk Test Classic install directory.
3. Go to the [Options] section, unless otherwise noted, and change the value of the option. You may need to add a line containing the section name, if it does not already exist.
4. Save and close the javaex.ini file.
5. Restart Silk Test Classic.

Troubleshooting Java Applications

This section provides solutions for common reasons that might lead to a failure of the test of your standalone Java application or applet. If these do not solve the specific problem that you are having, you can enable your extension manually.

The test of your standalone Java application or applet may fail if the application or applet was not ready to test, the Java plug-in was not enabled properly, if there is a Java recognition issue, or if the Java applet does not contain any Java controls within the **JavaMainWin**.

Why Is My Java Application Not Ready To Test?

This functionality is supported only if you are using the Classic Agent.

If your Java application is not ready to test, enable the extension for the application and restart the application.

1. On the **Basic Workflow** bar, click **Enable Extensions**. The **Enable Extensions** dialog box opens.
2. On the **Enable Extensions** dialog box, select the Java application for which you want to enable extensions.
3. Click **OK**. The **Enable Extensions** dialog box closes.
4. Close and restart the Java application.
5. When the application has finished loading, click `Test`.

Why Can I Not Test a Java Application Which Is Started Through a Command Prompt?

This functionality is supported only if you are using the Classic Agent.

If you are starting your standalone Java application through a **Command Prompt** window, close and re-open the **Command Prompt** window when you restart your application.

If you have forgotten to close and re-open the **Command Prompt** window, use the **Basic Workflow** bar to enable the extension again, making sure that you close and re-open both your Java application and the **Command Prompt** window before you click **Test** on the **Test Extension Settings** dialog box.

1. On the **Basic Workflow** bar, click **Enable Extensions**. The **Enable Extensions** dialog box opens.
2. On the **Enable Extensions** dialog box, select the Java application for which you want to enable extensions.
3. On the **Extension Settings** dialog box, click **OK**.
4. Close your Java application and the **Command Prompt** window.
5. Open a **Command Prompt** and restart your application.
6. When the application has finished loading, click **Test**.

What Can I Do If My Java Application Not Contain Any Controls Below JavaMainWin?

This functionality is supported only if you are using the Classic Agent.

If your Java application (or applet) does not contain any Java children within `JavaMainWin`, your tests against the application will fail. However, you might configure the Java extension to prevent this kind of failure. Record against Java controls to make sure that the extension is enabled. For example, record a push button as a `JavaAWTPushButton` or a `JavaJFCPushButton`.

How Can I Enable a Java Plug-In?

This functionality is supported only if you are using the Classic Agent.

If the browser that you are using has a plug-in enabled, or if the applet uses a plug-in, you must check the **Java Plug-in** check box on the **Extension Settings** dialog box. In all other cases, uncheck the **Java Plug-in** check box.

In Internet Explorer, click **Tools** > **Internet Options** and then click the **Advanced** tab, to determine if Internet Explorer has a plug-in enabled. Scan the **Settings** list to see if a third party plug-in, such as Java (Sun), has been enabled.

What Can I Do If the Java Plug-In Check Box Is Not Checked?

This functionality is supported only if you are using the Classic Agent.

If a plug-in is enabled for the browser, and the applet is using a plug-in, but you did not check the **Java Plug-In** check box, check the **Java Plug-In** check box and enable the extension again.

1. On the **Basic Workflow** bar, click **Enable Extensions**. The **Enable Extensions** dialog box opens.
2. On the **Extension Settings** dialog box, make sure `DOM` is the **Primary Extension**.
3. Check the **Java Plug-in** check box.
4. Click **OK**.
5. Close and restart your Java application.
6. When the application has finished loading, click **Test**.

What Can I Do When I Am Testing an Applet That Does Not Use a Plug-In, But the Browser Has a Plug-In Loaded?

This functionality is supported only if you are using the Classic Agent.

When you are testing an applet that does not use a plug-in, but the browser has a plug-in loaded, disable the plug-in and enable the extension again.

1. In the browser that you are using, disable all plug-ins.
2. In the **Basic Workflow** bar, click **Enable Extensions** and enable the extension for the applet again.
3. In the **Extension Settings** dialog box, uncheck the **Java Plug-in** check box.

What Can I Do If the Silk Test Java File Is Not Included in a Plug-In?

If the `SilkTest_Java3.jar` file is not included in the `lib/ext` directory of the plug-in that you are using:

1. Locate the `lib/ext` directory of the plug-in that you are using and check if the `SilkTest_Java3.jar` file is included in this folder.
2. If the `SilkTest_Java3.jar` file is not included in the folder, copy the file from the `javaex` folder of the Silk Test installation directory into the`lib\ext` directory of the plug-in.

What Can I Do If Java Controls In an Applet Are Not Recognized?

Silk Test Classic cannot recognize any Java children within an applet if your applet contains only custom classes, which are Java classes that are not recognized by default, for example a frame containing only an image. For information about mapping custom classes to standard classes, see *Mapping Custom Classes to Standard Classes*. Additionally, you have to set the Java security privileges that are required by Silk Test Classic.

Supported Java Classes

We provide 4Test definitions in our class definition file for the following Java classes:

- Abstract Windowing Toolkit (AWT) classes

- Java Foundation Class (JFC) library classes
- Symantec Visual Café Itools classes (only for the Classic Agent)
- Java-equivalent window classes

Each of these predefined classes inherits 4Test properties and methods, which are referenced in the class descriptions in this Help. Not all inherited methods have been implemented for Java controls.

You can also access the native methods of the supported classes by removing the 4Test definition and re-recording the class.

The only assumption that the Java extension makes about the implementation of the Java classes in an AUT is that the classes do not violate the standard Swing or AWT models. The Java extension should be able to recognize and manipulate a Java class in an application, as long as the class extends one of the components that the Java extension supports, and any customization does not violate the API of that component. For example, changing a method from public to private violates the API of the component.

Predefined Java-Equivalent Window Classes

The following 4Test classes are provided for testing Java-equivalent window controls:

Classic Agent	Open Agent
`JavaApplet`	`AppletContainer`
`JavaDialogBox`	`AWTDialog`
	`JDialog`
`JavaMainWin`	`AWTFrame`
	`JFrame`

Predefined AWT Classes

The following 4Test classes are provided for testing Abstract Windowing Toolkit (AWT) controls:

Classic Agent	Open Agent
`JavaAwtCheckBox`	`AWTCheckBox`
`JavaAwtListBox`	`AWTList`
`JavaAwtPopupList`	`AWTChoice`
`JavaAwtPopupMenu`	No corresponding class.
`JavaAwtPushButton`	`AWTPushButton`
`JavaAwtRadioButton`	`AWTRadioButton`
`JavaAwtRadioList`	No corresponding class.
`JavaAwtScrollBar`	`AWTScrollBar`
`JavaAwtStaticText`	`AWTLabel`
`JavaAwtTextField`	`AWTTextField`
	`AWTTextArea`

Predefined JFC Classes

The following 4Test classes are provided for testing Java Foundation Class (JFC) controls:

Classic Agent	Open Agent
`JavaJFCCheckBox`	`JCheckBox`

Classic Agent	Open Agent
JavaJFCCheckBoxMenuItem	JCheckBoxMenuItem
JavaJFCChildWin	No corresponding class.
JavaJFCComboBox	JComboBox
JavaJFCImage	No corresponding class.
JavaJFCListBox	JList
JavaJFCMenu	JMenu
JavaJFCMenuItem	JMenuItem
JavaJFCPageList	JTabbedPane
JavaJFCPopupList	JList
JavaJFCPopupMenu	JPopupMenu
JavaJFCProgressBar	JProgressBar
JavaJFCPushButton	JButton
JavaJFCRadioButton	JRadioButton
JavaJFCRadioButtonMenuItem	JRadioButtonMenuItem
JavaJFCRadioList	No corresponding class.
JavaJFCScale	JSlider
JavaJFCScrollBar	JScrollBar JHorizontalScrollBar JVerticalScrollBar
JavaJFCSeparator	JComponent
JavaJFCStaticText	JLabel
JavaJFCTable	JTable
JavaJFCTextField	JTextField JTextArea
JavaJFCToggleButton	JToggleButton
JavaJFCToolBar	JToolBar
JavaJFCTreeView	JTree
JavaJFCUpDown	JSpinner

Predefined Symantec Itools Classes

This functionality is supported only if you are using the Classic Agent.

The following 4Test classes are provided for testing Symantec Visual Café Itools controls:

- JavaItoolsComboBox
- JavaItoolsListBox
- JavaItoolsPageList
- JavaItoolsPushButton
- JavaItoolsScale
- JavaItoolsTable

- `JavaItoolsTreeView`
- `JavaItoolsUpDown`

Recording Java Classes

This functionality is supported only if you are using the Classic Agent.

This section describes how you can record Java classes.

When to Record Classes

This functionality is supported only if you are using the Classic Agent.

Consider these criteria when deciding whether to record classes for Java objects. When you record classes, Silk Test Classic derives tags from the Java class name. You may also find it helpful to consult the decision tree for dealing with custom Java classes.

Am I testing only predefined Java classes?

If you are testing only predefined Java classes, then you do not need to record additional classes. Check the list of predefined Java classes to be sure. If you want to access native methods for predefined Java classes, then you must remove the existing definition and re-record the class.

Am I testing visible custom controls?

If you are testing custom Java controls that are not predefined, then you must record classes for these controls. In this case, the custom controls are visible, but display as `CustomWin` objects. After you record the class, you can retrieve information about any number of instances (objects) of that class.

Do I want to test custom controls that are currently ignored?

To maintain efficiency during the recording process, Silk Test Classic ignores custom Java controls that are not considered relevant for testing, such as containers or panels. Ignored objects are not recognized at all by Silk Test Classic, not even as `CustomWin` objects.

Even so, you can expose and record classes for ignored Java objects in standalone Java applications or in Java applets that you consider important for testing purposes.

Have I modified an existing class definition?

If you add, delete, or modify any native methods or parameters for a custom Java class, you need to either re-record the class or modify your class include file to reflect the changes.

Decision tree for dealing with custom Java classes

1. Can you see the object without **Show All Classes** checked?
    a. If yes, then do you get any methods and properties using **Record Class**?
        a. If yes, then use **Record Class** to generate a winclass for the custom class. We recommend that you check **Show all methods**.
        b. If no, then do you get any methods and properties using the `CaptureObjectClass`() or `CaptureAllClasses`() function?
            a. If yes, then use `CaptureObjectClass`() to generate a winclass for the custom class, or `CaptureAllClasses`() to generate winclasses for the custom class and all child custom classes.
            b. If no, then go to Step 1.b.ii. You already know the class, but you will need to determine the public methods.
    b. If no, then can you see it with **Show All Classes** checked?

a. If yes, then you need to adjust the `[ClassList]` section in `extend\JavaEx.ini`.

   a. Expose classes that are ignored by default, which means with **Show All Classes** unchecked, by setting them to `true`.
   b. You may also need to hide some classes that are exposed by default by setting them to `false`. This may be true for tables where you see the individual cells but not the entire table, or for classes that are obscured by container or panel classes.
   c. Uncheck **Show All Classes** after you modify the `ClassList`, before recording any classes or window declarations.
   d. Go back to Step 1.

b. If no, then that indicates that the component is not derived from AWT Component. You will need to find out from the customer what the custom class is and which public methods are available for that class. The easiest way to determine the public methods is to find it out from the customer, but you can also try using 'javap', which is part of the JDK, to extract the public methods from the class.

   a. Can you access the object (ObjectA) of this class (Class A) through a method of a different class (Class B) that Silk Test Classic can recognize? The useful Class B method would only be recorded if you check **Show all methods** when recording the class for the Class B object (ObjectB).

      a. If yes, then does the useful Class B method take only 4Test-compatible values as parameters?

         a. If yes, then do the methods that you want to call for Class A return 4Test-compatible values and take 4Test-compatible values as parameters?

            a. If yes, then you should be able to call *ObjectB*.`invokeMethods`() to call the methods for *ObjectA*: *ObjectB*.`invokeMethods` ({"ClassBMethod", "ClassAMethod"}, {lArgumentsForClassBMethod, lArgumentsForClassAMethod}).
            b. If no, then use *ObjectB*.`InvokeJava`(). Within the class that you create for `InvokeJava`(), call the Class B method that returns ObjectA, then call the Class A method.

         b. If no, then use *ObjectB*.`InvokeJava`(). Within the class that you create for `InvokeJava`(), call the Class B method that returns ObjectA, then call the Class A method.

      b. If no, then use `InvokeJava`(). You will have to find a Silk Test Classic-recognizable object that can indirectly be used to access ObjectA through intermediate objects.

2. Does **Record Class** or the capture class functions give you useful 4Test-accessible methods for the class in question (Class A)?

   a. If yes, then call the recorded methods directly in your scripts.
   b. If no, then are the methods that you need to use commented out?

      a. If yes, then are the methods commented out only because they return values that are not 4Test-compatible, probably because they return custom classes?

         a. If yes, then are there methods on the returned classes that return 4Test-compatible values and take only 4Test-compatible values as parameters?

            a. If yes, then you should be able to call `invokeMethods`() on the object of interest (*ObjectA*): *ObjectA*.`invokeMethods`({"ClassAMethodOfInterest", "4TestCompatibleMethodForClassReturnedByClassAMethod"}, {lArgumentsForClassAMethod, lArgumentsForMethodForClassReturnedByClassAMethod})
            b. If no, then use `InvokeJava`() on the object of interest. Call the non-4Test-compatible methods within the class that you create for `InvokeJava`(). Make sure that you eventually return a 4Test-compatible value from the `InvokeJava`() class.

b. If no, then use `InvokeJava()` on the object of interest. Call the non-4Test-compatible methods within the class that you create for `InvokeJava()`. Make sure that you eventually return a 4Test-compatible value from the `InvokeJava()` class.

c. If no, then use `InvokeJava()` on the object of interest.

How Methods and Properties are Enumerated

This functionality is supported only if you are using the Classic Agent.

When you record classes, Silk Test Classic enumerates properties and methods as follows:

- By default, Silk Test Classic filters out methods and properties inherited at or above a certain level in a class hierarchy. The threshold at which filtering occurs varies according to the hierarchy. The filtering process makes it easier for you to find the methods and properties you use most frequently. You can turn off this filter to access any of these inherited methods and properties.
- After filtering, Silk Test Classic enumerates only those native methods that accept or return supported Java classes. You can however use unsupported native methods with subclasses that are supported.

Using Native Methods that are Not Enumerated

This functionality is supported only if you are using the Classic Agent.

When you record classes, Silk Test Classic does not enumerate native methods that pass unsupported Java classes as arguments; however, you can use these methods with supported subclasses. Add the prototype by hand in your class definition include file and then use the method in your test script as defined.

Example

If you record a class on a JFC ComboBox, the following native method prototypes are not enumerated because the Java support in Silk Test Classic does not support the `Object` class:

- Object getItemAt(int)
- Object getSelectedItem()
- void setSelectedItem(Object)

If you know that all the items on the JFC ComboBox are instances of a supported class, such as the default Java string class, you can add these prototypes in your class definition include file. Here's how the declarations should look for the Java string class:

- String getItemAt(int)
- String getSelectedItem()
- void setSelectedItem(String)

Thresholds for Filtering Methods and Properties

This functionality is supported only if you are using the Classic Agent.

In this hierarchy…	Classes are filtered at…
Abstract Windowing Toolkit (AWT)	java.awt.Component and above
Java Foundation Classes (JFC)	com.sun.java.swing.Jcomponent and above

Naming Conflicts

When you record a class, Silk Test Classic checks for and resolves naming conflicts that arise between 4Test methods and the supported native methods. When naming conflicts arise, make sure you call the

appropriate method in your test scripts. The following table shows how Silk Test Classic resolves the naming conflicts:

Type of Naming Conflict	How Silk Test Classic Resolves the Conflict
overloaded native methods	Appends "_n" to overloaded method names to ensure that each is unique, where n is an integer that starts at 2.
native method has the same name as a 4Test method	Prefixes the letter "x" to the name of the native method. For ActiveX/Visual Basic methods only, Silk Test Classic prefixes an underscore character (_) to native methods that begin with "set" and "get" in order to distinguish them from the Get and Set methods that Silk Test Classic constructs from properties.
native method uses a reserved 4Test keyword in an inappropriate context	Prefixes the letter "x" to the name of the native method.

When Silk Test Classic prefixes the letter "x" to the name of a native method or property, it also adds the alias keyword with the original name of the native method.

Example: Resolve Conflicts for Overloaded Methods

Overloaded Method	How Silk Test Classic Resolves the Conflict
obj void wait(int i1)	obj void wait(int i1)
obj void wait(int i1, int i2)	obj void wait_2(int i1, int i2)
obj void wait()	obj void wait_3()

Example: Resolve Naming Conflicts Between 4Test and Native Java Methods

4Test Method	Native Java method
getName()	getName() -> xgetName()
isEnabled()	isEnabled() -> xisEnabled()

Example: Resolve Naming Conflicts Between 4Test Reserved Words and Native Java Methods

4Test Reserved Word	Native Java method
LIST data type	list() -> xlist()
select statement	select() -> xselect()

Java Custom Windows

This functionality is supported only if you are using the Classic Agent.

You do not need to write your own extensions in Java for testing custom Java objects, which are also called `CustomWins`. Instead, the Silk Test Classic Java support lets you access native methods that you can call in your test scripts to manipulate custom Java controls.

You access native methods for a custom Java object by recording a class for that object. To get started, take a look at our guidelines for when and how to record classes.

Loading Class Definition Files or Test Frame Files

This functionality is supported only if you are using the Classic Agent.

You can use these procedures to load 4Test include files or test frames.

We recommend that you record new class definitions in a new include file or in your test frame file. Do not store these definitions in `javaex.inc`, or another predefined class definition file such as `dotnet.inc`, because we may upgrade *.inc files in a later Silk Test release.

To load class definitions or test frames in selected test scripts

Insert a use statement in each test script that needs to manipulate the controls that you have declared or the objects of the classes that you have defined. Use the following format:

```
use "<directory\file-name>.inc"
```

For example, if your class include file is `custobj.inc` and it resides in the directory `c:\mydir`, insert the following statement:

```
use "c:\mydir\custobj.inc"
```

Recording Classes for Custom Java Controls

This functionality is supported only if you are using the Classic Agent.

To enable Silk Test Classic to recognize custom controls, you must record classes for these objects.

The process of recording a custom Java class involves querying the objects, retrieving information on methods and properties for these objects, and then translating this information into 4Test-style prototypes that you can use to write test scripts.

How to Record Classes

Using the Java support in Silk Test Classic, you can use the following approaches to record classes for custom controls in Java applications and applets:

- Record classes for custom Java objects using the recorder.
- Recording classes for custom Java objects from a script.
- Record classes for a window and all of its children using one function call.

Where to store your new class definitions

We recommend that you store your class definitions in a new include file, for example `custobj.inc` or in your test frame file. Do not store these definitions in `javaex.inc`, which is the predefined Java class definition file, because we will upgrade `javaex.inc` in future versions of our Java support. You will need to load new class include files in Silk Test Classic before testing your application or applet.

If you add, delete, or modify any native methods or parameters for a custom Java class, you need to either re-record the class or modify your class include file to reflect the changes.

Recording Classes for Custom Java Controls Using the Recorder

This functionality is supported only if you are using the Classic Agent.

Before beginning this procedure, make sure you have taken the necessary prerequisite steps to set up your environment as described in *Configuring Silk Test Classic to Test Java*.

To record new classes for custom controls using the recorder:

1. Start your Java application or applet.
2. Create a new include (.inc) file, open an existing include file, or open the test frame file for storing your new class definitions.

3. Click **Record** > **Class** > **Scripted** to open the **Record Class** dialog box.
4. To include native methods with return or parameter types that do not match valid 4Test methods, check the **Show all methods** check box in the lower left corner of the dialog box.

   You cannot call these methods directly from your 4Test scripts but you can use the `InvokeMethods` or the `InvokeJava` method to call them. When you capture the class, Silk Test Classic displays these methods prefaced by comments in the **Record Class** dialog box. When you click **Paste to Editor**, Silk Test Classic adds the `InvokeMethods` method and these methods, prefaced by comments, to your test script.
5. Position the mouse pointer over the control for which you want to record a class.
6. When the correct name displays in the Window text box, press **Ctrl+Alt**.

   Methods and properties for that class are displayed in the **Record Class** dialog box. If you have checked **Show All Methods**, Silk Test Classic displays these non-4Test methods as comments, that is, prefaced by forward slashes (//).
7. Click the **Derived From** list box to see the list of available 4Test classes. If there is a class type in the list that maps directly to your object, choose it. If not, choose `AnyWin`, which is a generic class.

   For example, if your object is `JavaAwtTextField`, choose `TextField`. If your object is `Spinner`, choose `Anywin`. For additional information, see *winclass Declaration* and derived class.
8. Click **Paste to Editor** to paste the new class into the include file.
9. Uncheck the **Show all methods** check box in the **Record Class** dialog box, if you chose it for this record action.
10. Repeat this procedure for each custom control that does not display in the predefined list of Java classes provided. When you are finished recording classes, click **Close**.
11. Load the include file that contains the new class definitions.

If you find that Silk Test Classic does not recognize some of your custom Java controls, you may need to take additional steps to record classes for these "ignored" objects.

To include native methods with return or parameter types that do not match valid 4Test methods, check the `Show All Methods` check box on the **Record Class** dialog box. Silk Test Classic displays these methods as comments in Methods list.

When you finish recording the class, uncheck the **Show all methods** check box to turn off the recording of all methods. Turning off **Show all methods** when you don't need it helps to keep performance optimal.

Although you cannot call these methods directly from your 4Test script, you can use `InvokeMethods` or `InvokeJavaCode` to call them from your script.

If you add or delete native methods, or modify the parameters of native methods for a custom Java class, you need to either re-record the class or edit your class include file to reflect the changes.

If your test script fails with the error `Function x is not defined for window y`, you might need to modify your window tag from `CustomWin` to the name of your new window class.

Recording Classes for Custom Java Controls from a Script

This functionality is supported only if you are using the Classic Agent.

Silk Test Classic provides two functions that allow you to capture the class information of custom Java controls from a script:

- `CaptureAllClasses`
- `CaptureObjectClass`

The following procedure explains how to use these functions to record new classes for custom Java controls. Before beginning this procedure, make sure you have taken the necessary prerequisite steps to set up your environment as described in *Configuring Silk Test Classic Java Support* and *Overview of Testing Java Applications and Applets*.

To record new classes for custom controls from a script:

1. Start your Java application or applet.
2. Create a new include (.inc) file, open an existing include file, or open an existing test frame file for storing your new class definitions.
3. Open a new script (.t) file.
4. Load the include file `captureclass.inc`, located in the directory where you installed Silk Test Classic.

   Note: We recommend loading this include file only in the scripts that call `CaptureObjectClass` or `CaptureAllClasses`.
5. Open the windows that contain the custom controls and their parent windows.
6. Record declarations for the windows containing your custom controls and paste the declarations into your script file.
7. In your script file, write a `main` routine that calls one or both of the capture functions, according to the following guidelines:

If you want to capture …	Call …
The class for one custom control	`CaptureObjectClass`.
The class for a custom control and all of its children	`CaptureAllClasses`.

8. Save and run your script file. The results file opens on your desktop and contains the new class definitions.
9. Copy the class definitions from your results file and paste them into the include file you have designated in step 2.
10. Load the include file that contains the new class definitions.

- If you find that Silk Test Classic does not recognize some of your custom Java controls, you may need to take additional steps to record classes for these "ignored" objects.
- If you add or delete native methods, or modify the parameters of native methods for a custom Java class, you need to either re-record the class or edit your class include file to reflect the changes.
- If your test script fails with the error `Function x is not defined for window y`, you might need to modify your window tag from `CustomWin` to the name of your new window class. For the correct sequence of steps to perform before you begin writing test scripts, see *Testing Java Applications and Applets*.

Example Script that Calls CaptureObjectClass

This functionality is supported only if you are using the Classic Agent.

This call to `CaptureObjectClass` records a class named `SwingSplitPane` for `JavaxSwingJSplitPane`, a custom control in the `SplitPane` window.

Note: You must pass the full path of the window whose class you want to capture. In this example, the full path is `SplitPane.JavaxSwingJSplitPane`.

```
// capture_obj.t
use "captureclass.inc"

window JavaMainWin TestApplication
  tag "TestApplication"
  JavaJFCMenu File
  JavaJFCMenu Control
  JavaJFCMenu Menu
window JavaDialogBox SplitPane
  tag "SplitPane"
  parent TestApplication
  JavaJFCCheckBox Horizontal
```

```
  JavaJFCCheckBox Enabled
  JavaJCFCheckBox Exit
  CustomWin JavaxSwingJSplitPane

main()
  print("Calling
CaptureObjectClass(""SwingSplitPane"",SplitPane.JavaxSwingJSplitPane)")
  CaptureObjectClass("SwingSplitPane",SplitPane.JavaxSwingJSplitPane)
```

Results of Call to CaptureObjectClass

This functionality is supported only if you are using the Classic Agent.

Following is the results file produced by running a sample script that calls the `CaptureObjectClass` function to record a class named `SwingSplitPane` for `JavaxSwingJSplitPane`, a custom control in the `SplitPane` window. The new class declaration has been expanded to show the class information that is recorded.

```
// capture_obj.res
Calling CaptureObjectClass("SwingSplitPane", SplitPane.JavaxSwingJSplitPane)
  winclass SwingSplitPane : Control
  tag "[javax.swing.JSplitPane]"

// Properties
property int iDividerLocation alias "$DividerLocation"
property int iDividerSize alias "$DividerSize"
property int iLastDividerLocation alias "$LastDividerLocation"
property int iOrientation alias "$Orientation"

// Accessible Native Methods
obj boolean isContinuousLayout()
obj boolean isOneTouchExpandable()
obj boolean isValidateRoot()
obj int getDividerLocation()
obj int getDividerSize()
obj int getLastDividerLocation()
obj int getMaximumDividerLocation()
obj int GetMinimumDividerLocation()
obj int GetOrientation()
obj String getUIClassID()
obj void remove(int il)
obj void removeAll()
obj void resetToPreferredSizes()
obj void setContinuousLayout(boolean bl)
obj void setDividerLocation(float fl)
obj void setDividerLocation_2(int il)
obj void setDividerSize(int il)
obj void setLastDividerLocation(int il)
obj void setOneTouchExpandable(boolean bl)
obj void setOrientation(int il)
obj void updateUI()
```

Example Script that Calls CaptureAllClasses

This functionality is supported only if you are using the Classic Agent.

```
// capture_all.t
use "captureclass.inc"

window JavaMainWin TestApplication
  tag "TestApplication"
  JavaJFCMenu File
  JavaJFCMenu Control
  JavaJFCMenu Menu
window JavaDialogBox SplitPane
```

```
  tag "SplitPane"
  parent TestApplication
  JavaJFCCheckBox Horizontal
  JavaJFCCheckBox Enabled
  JavaJCFCheckBox Exit
  CustomWin JavaxSwingJSplitPane

main()
  print("Calling CaptureAllClasses(TestApplication)")
  CaptureAllClasses(TestApplication)
  print("****************")
  print("Calling CaptureAllClasses(SplitPane, FALSE)")
  CaptureAllClasses(SplitPane, FALSE)
  print("****************")
  print("Calling CaptureAllClasses(SplitPane, TRUE)")
  CaptureAllClasses(SplitPane, TRUE)
```

The first two calls to `CaptureAllClasses` record classes for custom controls in the named window and its children. Classes are not recorded for controls whose classes are already defined, for example, controls that have predefined 4Test classes.

The third function call records classes for all controls in `SplitPane` and its children, including controls whose classes are already defined.

Results of Call to CaptureAllClasses

This functionality is supported only if you are using the Classic Agent.

Following is the results file produced by running a sample script that calls the `CaptureAllClasses` function to record classes for visible custom controls.

Note: The third call to `CaptureAllClasses` passes TRUE as the second argument, directing the function to capture classes for all child objects - even those whose classes are predefined, such as `JavaJFCPushButton`, `JavaJFCTextField`, and `JavaJFCImage`.

```
Calling CaptureAllClasses(TestApplication)
  winclass SunAwtDialog : Control
  winclass JavaxSwingSplitPane : Control
  winclass JavaxSwingPlafMetalMetalSplitPaneDivider : Control

****************
Calling CaptureAllClasses(SplitPane, FALSE)
  winclass JavaxSwingSplitPane : Control
  winclass JavaxSwingPlafMetalMetalSplitPaneDivider : Control

****************
Calling CaptureAllClasses(SplitPane, TRUE)
  winclass JavaJFCCheckBox : Control
  winclass JavaJFCPushButton : Control
  winclass JavaxSwingSplitPane : Control
  tag "[javax.swing.JSplitPane]"

// Properties
property int iDividerLocation alias "$DividerLocation"
property int iDividerSize alias "$DividerSize"
property int iLastDividerLocation alias "$LastDividerLocation"
property int iOrientation alias "$Orientation"

// Accessible Native Methods
obj boolean isContinuousLayout()
obj boolean isOneTouchExpandable()
obj boolean isValidateRoot()
obj int getDividerLocation()
obj int getDividerSize()
```

```
obj int getLastDividerLocation()
obj int getMaximumDividerLocation()
obj int GetMinimumDividerLocation()
obj int GetOrientation()
obj String getUIClassID()
obj void remove(int il)
obj void removeAll()
obj void resetToPreferredSizes()
obj void setContinuousLayout(boolean bl)
obj void setDividerLocation(float fl)
obj void setDividerLocation_2(int il)
obj void setDividerSize(int il)
obj void setLastDividerLocation(int il)
obj void setOneTouchExpandable(boolean bl)
obj void setOrientation(int il)
obj void updateUI()
```

Recording Classes for Ignored Java Objects

This functionality is supported only if you are using the Classic Agent.

Silk Test Classic ignores certain objects during recording that normally should remain transparent to users, such as panels and containers. Typically these classes don't have a graphical component, or are used solely to aid the placement of objects. However, there may be cases in which these ignored classes have been extended and contain objects that you want to test. In some situations, custom objects, such as user-defined objects or third-party JavaBeans, might be inadvertently ignored.

Using our Java support in Silk Test Classic, you can expose these ignored objects, then record classes for them in Java applications and in Java applets.

To record classes for ignored Java objects:

1. Start your Java application and make sure Java support is enabled.
2. Create a new include (.inc) file, open an existing include file, or open the test frame file for storing your new class definitions.

   We recommend that you store your new class definitions in a new include file, for example `custobj.inc`, or in your test frame file. Do not store these definitions in `javaex.inc`, the predefined Java class definition file, because we will upgrade `javaex.inc` in future versions of our Java support.
3. Click **Record** > **Class** and then check the **Show all classes** check box in the lower left corner of the dialog box.
4. To include native methods with return or parameter types that do not match valid 4Test methods, check the **Show all methods** check box in the lower left corner of the dialog box.

   You cannot call these methods directly from your 4Test scripts but you can use the `InvokeMethods` or the `InvokeJava` method to call them. When you capture the class, Silk Test Classic displays these methods prefaced by comments in the **Record Class** dialog box. When you click **Paste to Editor**, Silk Test Classic adds the `InvokeMethods` method and these methods, prefaced by comments, to your test script.
5. Position the mouse pointer over the control for which you want to record a class, and when the correct name displays in the Window field, press **Ctrl+Alt**.

   Methods and properties for that class are displayed in the **Record Class** dialog box. If you have checked **Show All Methods**, Silk Test Classic displays these non-4Test methods as comments, that is, prefaced by forward slashes (//).
6. Click the **Derived From** list box to see the list of available 4Test classes.
   - If there is a class type in the list that maps directly to your object, choose it.
   - If not, choose `AnyWin`, which is a generic class. See winclass declaration and derived class for more details.

Note: Note the Tag for the class you just recorded; you will need this name later.

7. Click **Paste to Editor** to paste the new class into the include file.
8. Uncheck the **Show all classes** check box in the **Record Class** dialog box, and the **Show all methods** check box, if you chose it for this record action, and then click **Close**.

   It is very important to check **Show all classes** only while you are trying to record the class for an ignored Java object.
9. Open `javaex.ini`, located in the extend subdirectory of the directory where you installed Silk Test Classic.
10. In `javaex.ini`, create a section called `[ClassList]` and add a line that reads `<class tag name>=true`.

    For example, if the tag of the class you just recorded is `[com.mycompany.Spinner]`, add this line:

    ```
    com.mycompany.Spinner=true
    ```

    Note: The name can contain wildcards, which can be useful for exposing all classes in a package, for example:

    ```
    com.mycompany.module_classes_to_expose.*\=true
    ```
11. Save and close `javaex.ini`.

    When you uninstall Silk Test Classic, the file `javaex.ini` is backed up as `javaex.bak` in the `<Silk Test install directory>\extend` folder. Any changes you made to `javaex.ini` can be reinstated by copying them from `javaex.bak` and pasting them into the new `javaex.ini` that is created when you reinstall Silk Test Classic.
12. Restart the Agent and your application.
13. Load the include file that contains the new class definitions.

Recording Java Window Declarations

This functionality is supported only if you are using the Classic Agent.

To record Java window declarations:

1. Create a new test frame file.

   Silk Test Classic loads a new test frame file by automatically specifying its full path in the **Use Files** text box of the **Runtime Options** dialog box. Instead of creating a new test frame, you can open a test frame file that you already created for this Java test script or suite. If you use an existing test frame file that has not been loaded, load the test frame file now.
2. If you recorded classes for any Java controls, load the class definition file now.
3. With your test frame file as the active window, choose **Record** > **Window Declarations** and record declarations.

You can now use multitags when recording window declarations for Java applications. However, additional considerations must be made for top-level windows.

Turning On the Class Declaration Filter

This functionality is supported only if you are using the Classic Agent.

When you record classes, Silk Test Classic by default filters out properties and methods inherited at or above a certain level in a class hierarchy. The threshold at which filtering occurs varies according to the hierarchy.

You can turn off this filter if you want these properties and methods to be enumerated. If you then want to turn the filter back on, follow the procedure described below.

Note: We provide this filter as a convenience, but it has not been thoroughly tested.

To turn on the class declaration filter:

1. Open the `javaex.ini` file.

   The path is `<SilkTest install directory>\extend\javaex.ini`.
2. Either remove the declaration `FilterClassDecl=false` from the `[Recording]` section, or change the declaration to `FilterClassDecl=true`.
3. Restart the Agent, and the application or applet under test.

Turning Off the Class Declaration Filter

This functionality is supported only if you are using the Classic Agent.

When you record classes, Silk Test Classic by default filters out properties and methods inherited at or above a certain level in a class hierarchy. The threshold at which filtering occurs varies according to the hierarchy. You can turn off this filter if you want these properties and methods to be enumerated.

We provide this filter as a convenience, but it has not been thoroughly tested.

To turn off the class declaration filter:

1. Open the `javaex.ini` file.

   The path is `<SilkTest install directory>\extend\javaex.ini`.
2. In the `[Recording]` section, add the command `FilterClassDecl=false`.
3. Save and close `javaex.ini`.
4. Restart the Agent, and the application or applet under test.

Extending Java Support

This functionality is supported only if you are using the Classic Agent.

This section describes how you can extend Java support.

Keeping the DOS Window Open when Returning to DefaultBaseState

This functionality is supported only if you are using the Classic Agent.

Java applications running in Windows are launched from DOS.

You do not need to perform this task if appstate is set to `none`.

To keep the DOS window open when returning to base state:

1. Launch your Java application.
   - If you are running JDK, your DOS window is minimized.
   - If you are running JRE, your DOS window is not minimized.
2. Click **Record** > **Window Identifiers** in the Silk Test Classic menu bar.
3. Click the DOS window. If the DOS window is minimized, restore it first. Place your cursor over the title bar of the DOS window and press **Ctrl+Alt** to record the identifier.
4. Open your test frame file and expand the main window declaration for the Java application which you are testing.
5. Un-comment the line that begins `const lwLeaveOpen` and select the text ? near the end of that line.
6. In the **Record Windows Identifiers** dialog box, click **Paste to Editor** to insert the DOS window identifier as the value for `lwLeaveOpen`, inside the {} brackets.
7. Close the **Record Windows Identifiers** dialog box and save the test frame file.

Redirect Output from Java Console to File

This functionality is supported only if you are using the Classic Agent.

When you enable Java support, you can redirect Java console output to a local file, where you can more easily scroll and copy the text.

To redirect Java console output to a local file:

1. Choose **Options** > **Extensions** from the Silk Test Classic menu bar. The **Extensions** dialog box opens.
2. Select and highlight an enabled Java application or browser.
3. Click **Java**. The **Extension Options** dialog box opens.
4. Check the **Redirect Java Console Output** check box.
5. In the **Java Console File Name** text box, enter the path to the file where you want to redirect Java console output.
6. Click **OK**.

Note: Silk Test Classic hangs if you are using Java Plug-In for JVM 1.4.2 and call `Browser.Close()` with the Java Console open.

Using Java Database Connectivity (JDBC)

This functionality is supported only if you are using the Classic Agent.

To verify information from an SQL-compliant database, you might want to hook into JDBC to access the data. You can access JDBC directly by using the `InvokeJava` method to call a Java class that makes a series of JDBC calls and returns the data of interest.

invokeMethods Method

Class

`JavaMainWin`.

Availability

This functionality is supported only if you are using the Classic Agent.

Action

Retrieves information about a visible Java GUI object by calling a sequence of nested methods inside Java with arguments specified in 4Test.

Syntax

```
anyValue = object.invokeMethods(lsMethods, llanyArgs)
```

Variable	Description
anyValue	Value describing the object. `ANYTYPE`.
lsMethods	A list of method names in the order that they are to be called. `LIST OF STRING`.
llanyArgs	A list of argument lists, to be matched with the method names in *lsMethods*. `LIST OF LIST OF ANYTYPE`.

Notes

This method is available for Windows environments. The first method in the *lsMethods* method list is invoked on *object* with the argument list specified by the first element of *llanyArgs*. This method call and all subsequent calls except the last one must have a return object. Each return object becomes the target for calling the next method in *lsMethods* with the next argument list in *llanyArgs* until the last method call. If the

method names, argument lists, and return classes do not match, a runtime error is generated and reported to 4Test.

Requirements

The `invokeMethods` method is declared for all built-in 4Test Java classes. However, you must add the method to the class declaration of any custom class for which `invokeMethods` is invoked.

Note: `Record` class will include the `invokeMethods` declaration if you check the **Show all methods** check box.

Example 1

The following simple example uses `invokeMethods` to call `getScrollPosition`, which returns a `Point` object. Then `invokeMethods` calls the `toString` method, which returns a string representing the point object returned from the first call to `getScrollPosition`.

```
sValue =
MyDialog.ScrollableObject.invokeMethods({"getScrollPosition",
"toString"}, {{}, {}})
```

Example 2

To draw a line in a multiline text field, you need to access a graphics object inside the text field by calling the following methods in Java:

```
main()
{
  TextField multiLine = ...; // get reference to multiline text
field
  Graphics graphObj = multiLine.getGraphics();
  graphObj.drawLine(10, 10, 20, 20);
}
```

However, you cannot call the above sequence of methods from 4Test because `Graphics` is not 4Test-compatible. Instead, you can insert the `invokeMethods` prototype in the `TextField` class declaration, then add `invokeMethods` by hand to your test script to draw a line in the `Graphics` object nested inside the multiline text field, as shown in this 4Test function:

```
DrawLineInTextField()
MyDialog.Invoke() // Invoke Java dialog that contains the text
field
MyDialog.TheTextField.invokeMethods({"getGraphics",
"drawLine"}, {{}, {10, 10, 20, 20}})
```

In this code, the following methods are called in Java:

- `getGraphics` is invoked on the multiline text field `TheTextField` with an empty argument list, returning a `Graphics` object.
- `drawLine` is invoked on the `Graphics` object, to draw a line starting from (x,y) coordinates (10,10) and continuing to (x,y) coordinates (20,20).

Invoking Java Applications and Applets

This section describes how you can invoke Java applications and applets.

Invoking Java Applets

To invoke the Java applet from within a supported browser, perform the following tasks:

- If you are using the Classic Agent, configure Silk Test Classic for Java support and enable the Java extension.
- If you are using the Open Agent, configure the application.

Invoking JDK Applications

Once you configure Silk Test Classic for testing standalone Java applications, you can invoke JDK applications as you normally would from the command line.

To invoke JDK applications using -classpath

Enter the following command:

```
java -classpath <Java support path>;<other directories, if any> <name of application>
```

Example

In this example, let us assume the following:

- You have installed Silk Test Classic in the default directory `c:\Program Files\Silk\SilkTest`.
- You are using JDK 1.3 as your Java Virtual Machine (JVM).
- Your CLASSPATH contains only the Java support path.

Given these assumptions, you would launch the application MyJDKapp by entering:

```
java -classpath c:\Program Files\Silk\SilkTest\JavaEx\SilkTest_Java3.jar MyJDKapp
```

To invoke JDK applications without using command line arguments

Enter the following command:

```
java <name of application>
```

Example

To invoke the application MyJDKapp, enter:

```
java MyJDKapp
```

Invoking JRE Applications

Once you set CLASSPATH for testing standalone Java applications, you are ready to invoke your application using the Java Runtime Environment (JRE).

Note: The JRE ignores the CLASSPATH environment variable. As a result, you must invoke JRE applications with command line arguments to pick up the value of CLASSPATH.

The following table describes the commands you can use:

Command	Description
-cp	Searches first through directories and files specified, then through standard JRE directories.
-classpath	Ignores the value of your CLASSPATH environment variable. You must specify a complete search path on the command line. Does not search the standard JRE directories.

To invoke JRE applications using -cp

Enter the following command:

```
jre -cp %CLASSPATH%;<other directories, if any> <name of application>
```

> **Example**
>
> Assuming your CLASSPATH is set to the complete search path including the Java support path, you would launch the application MyJREapp by entering:
>
> ```
> jre -cp %CLASSPATH% MyJREapp
> ```

Invoking JRE Applications Using -classpath

To invoke JRE applications using –classpath, enter the following command:

```
jre -classpath <Java support path>;<other directories, if any>
```

> **Example**
>
> Assuming you installed Silk Test Classic in the default directory `c:\Program Files\Silk\SilkTest`, you are using JRE 1.1.5 as your Java Virtual Machine (JVM), and your CLASSPATH contains only the Java support path, you would launch the application MyJREapp by entering:
>
> ```
> Java -classpath c:\Progra~1\Silk\SilkTest\JavaEx
> \SilkTest_Java3.jar MyJREapp
> ```

Invoking Symantec Visual Café Applications from Command Line

This functionality is supported only if you are using the Classic Agent.

When you invoke Visual Café applications from the command line, the *CLASSPATH* environment variable is ignored. Therefore, to tell Visual Café about the Java support path, you must use the -classpath argument on the command line to specify the locations of the following class libraries:

Library	Path
Java support	Java support path
AWT and java.* classes	`<Visual Café install directory>\java\lib\classes.zip`
Symantec Itools classes This library is required only if your application uses Itools	`<Visual Café install directory>\bin\components\SymBeans.jar`

To invoke Visual Café applications from the command line:

1. Move to the directory where your Visual Café executable resides or put this directory on your path. The directory is `<Visual Café install directory>\java\bin\`.
2. Enter the following command:

   ```
   java.exe -classpath <Java support path>;
     <Visual Café install directory>\java\lib\classes.zip;
     <Visual Café install directory>\bin\components\SymBeans.jar <application>
   ```

Sample Visual Café Command Line

This functionality is supported only if you are using the Classic Agent.

Assuming the following conditions:

- You install Silk Test Classic in `c:\Silk`
- You install Visual Café 2.0+ in `c:\Symantec`
- Your application uses Itools controls
- Your application is `MyApp.class`

Then, your Visual Café command line should look like the following:

Invoking Symantec Visual Café Applications from IDE

This functionality is supported only if you are using the Classic Agent.

Before you invoke Visual Café applications from the Integrated Development Environment (IDE), you must first add the Java support path to the CLASSPATH by editing the `sc.ini` file.

To invoke Visual Café applications from the IDE:

1. Make sure Silk Test Classic is configured to test Java.
2. Open the `sc.ini` file in your favorite text editor.

   The file is located in `<Visual Café install directory>\bin\sc.ini`.
3. On the line that begins with `CLASSPATH=`, add the Java support path.

   For example if you have installed Silk Test Classic in `c:\Silk` and you are using Visual Café 2.0 as your Java Virtual Machine (JVM), add the following Java support path to the end of the CLASSPATH:

   ```
   ;c:\Silk\SilkTest_Java3.jar
   ```

   For additional information, refer to the VisualCafe documentation.
4. Save and close `sc.ini`.
5. Restart Visual Café.
6. Click **Project** > **Execute** to invoke your application within the IDE.

invokeMethods Example: Draw a Line in a Text Field

To draw a line in a multiline text field, you need to access a graphics object inside the text field by calling the following methods in Java:

```
main()
{
  TextField multiLine = ...; // get reference to multiline text field
  Graphics graphObj = multiLine.getGraphics();
  graphObj.drawLine(10, 10, 20, 20);
}
```

However, you cannot call the above sequence of methods from 4Test because Graphics is not 4Test-compatible. Instead, you can insert the invokeMethods prototype in the `TextField` class declaration, then add invokeMethods by hand to your test script to draw a line in the Graphics object nested inside the multiline text field, as shown in this 4Test function:

```
DrawLineInTextField()
MyDialog.Invoke() // Invoke Java dialog that contains the text field
MyDialog.TheTextField.invokeMethods ({"getGraphics", "drawLine"}, {{}, {10,
10, 20, 20}})
```

In this code, the following methods are called in Java:

- getGraphics is invoked on the multiline text field TheTextField with an empty argument list, returning a Graphics object.
- drawLine is invoked on the Graphics object, to draw a line starting from (x,y) coordinates (10,10) and continuing to (x,y) coordinates (20,20).

Accessing Java Objects and Methods

This section describes how you can access Java objects and methods.

Accessing Native Methods for Predefined Java Classes

This functionality is supported only if you are using the Classic Agent.

In some situations, you might want to access the native methods for predefined Java classes, for example if a particular function was not supported in 4Test, but could be performed using a native method. You can access the native methods for controls that are part of predefined Java classes by re-recording the class for the control.

To access native methods for predefined Java classes:

1. Start your Java application or applet, and Silk Test Classic.
2. Open the predefined class definition file, `javaex.inc`, and comment out the predefined definitions for classes whose native methods you want to access.
3. Create a new include (.inc) file or open an existing include file for storing your new class definitions.
4. Click **Record** > **Class** > **Scripted** to open the **Record Class** dialog box.
5. Position the mouse pointer over the predefined control for which you want to record a new class.
6. When the correct name displays in the Window field, press **Ctrl+Alt**. Methods and properties for that class are displayed in the **Record Class** dialog box.
7. Click the **Derived From** list box to see the list of available 4Test classes. If there is a class type in the list that maps directly to your object, choose it. If not, choose **AnyWin**, which is a generic class. See winclass declaration and derived class for more details.
8. Click **Paste to Editor** to paste the new class into the include file.
9. Repeat this procedure for each predefined class whose native methods you want to access. When you are finished recording classes, click **Close**.
10. Load the class include file that stores your new class definitions.

Accessing Nested Java Objects

Sometimes you cannot retrieve 4Test-compatible information about a Java control with a single call to a 4Test method; instead, you need to call several nested methods, each returning an intermediate object to be passed to the next method. If any of these methods returns intermediate results that are not 4Test-compatible, you will not be able to perform these nested calls from 4Test.

You can use the following methods to access nested Java objects:

Method	Agent	What it does
`InvokeJava`	Classic Agent	This method allows you to invoke a Java class from 4Test for manipulating a nested Java object.
`invokeMethods`	Classic Agent Open Agent	Allows you to perform nested calls inside Java, even if intermediate results are not 4Test-compatible.You can call `invokeMethods` for any Java object as long as you add the `invokeMethods` prototype inside the object's class declaration.

Accessing Non-Visible Java Objects

This functionality is supported only if you are using the Classic Agent.

Currently, Silk Test Classic cannot enumerate or manipulate Java objects that are not derived from the Abstract Windowing Toolkit (AWT) Component object.

With the `InvokeJava` method, you can access these non-visible objects by performing the following steps:

1. A development group adds a Java class to an application that provides methods for accessing object references to non-visible objects of interest.
2. Create the Java class to be invoked by the `InvokeJava` method.
3. In the Java class that is invoked by `InvokeJava`, call the access methods that provide a reference to the non-visible objects of interest in your application.
4. Manipulate the non-visible objects as desired.

Calling Nested Methods

Sometimes you cannot retrieve 4Test-compatible information about a Java control with a single call to a 4Test method; instead, you need to call several nested methods, each returning an intermediate object to be passed to the next method. If any of these methods returns intermediate results that are not 4Test-compatible, you will not be able to perform these nested calls from 4Test.

You can use the following methods to call nested methods:

Method	Agent	What it does
`InvokeJava`	Classic Agent	This method allows you to invoke a Java class from 4Test for manipulating a nested Java object.
`invokeMethods`	Classic Agent Open Agent	Allows you to perform nested calls inside Java, even if intermediate results are not 4Test-compatible.You can call `invokeMethods` for any Java object as long as you add the `invokeMethods` prototype inside the object's class declaration.

Example: How to add an invokeMethods prototype to your script

This example shows how to add an `invokeMethods` prototype inside the declaration for a `JavaAwtListBox` in `javaex.inc`.

```
winclass JavaAwtListBox : ListBox
  tag "[JavaAwtListBox]"
```

```
  setting MultiTags = {TAG_CAPTION}

  obj AnyType invokeMethods(list of Strings stra, List of List
of Anytype anyaa)
```

Identifying Custom Controls

This functionality is supported only if you are using the Classic Agent.

To identify custom Java controls:

1. Click **Record** > **Window Declarations**.
2. Pass your cursor over the controls you want to test. You will be able to identify the custom controls by their class, which appears as CustomWin in the **Record Windows Declarations** dialog box. You can also press **Ctrl+Alt** to pause tracking and view the controls you want to test.

Ignoring a Java Object

This functionality is supported only if you are using the Classic Agent.

If you are using the Java extension and you want to ignore a Java class, object, or container, you must edit your `javaex.ini` file:

1. Open `javaex.ini`, located in the `<SilkTest install directory>\extend` folder.
2. Add `myclass=false` as a new line to the file, where myclass is the full class name.

   For example, if the tag of the class you just recorded is `[com.mycompany.Spinner]`, add the following line:

   ```
   com.mycompany.Spinner=false
   ```

   Note: The name can contain wildcards, which can be useful for ignoring all classes in a package, for example `com.mycompany.module_classes_to_ignore.*\=false`. You must use the value `false` and not the value `ignore`.

You might find that ignoring a top level object causes all objects underneath it to be ignored as well. If this is an issue, you can mark the top level object as `false` and add any objects that you do not want to be ignored to the ClassList and set to the value to true. For example:

```
com.ignore.this.object = false
 com.dontignore.this.object = true
```

The second object displays in the GUI as a child of the first.

Testing Java Scroll Panes

A scroll pane in Java is a container that holds a single child component. If the scroll pane is smaller than the child component, you can scroll vertically and horizontally to see all parts of that component.

The state of the scroll bars in a scroll pane is managed by internal objects that implement the `Adjustable` interface. To manipulate the scroll bars, you must first get an `Adjustable` object, and then use `Adjustable` and scroll bar methods to move them.

To test scroll bars in a scroll pane, use `invokeMethods`, a method that allows you to perform nested calls inside Java to access `Adjustable` objects.

Frequently Asked Questions About Testing Java Applications

This section provides answers to frequently asked questions about classpath and testing Java applications and applets.

Why Do I See so Many Java CustomWin Objects?

Objects that do not belong to any of our predefined Java classes are custom controls, which are identified as `CustomWin` objects by Silk Test Classic. Most Java applications and applets use many custom controls to fine tune functionality and the user interface.

To manipulate a custom Java object for testing, you do not need to write your own extensions. Instead, you can use the object's own native methods and properties. Our Java support lets you access native methods and properties, by recording classes for custom controls.

Why Do I Need to Disable the Classpath if I have Java Installed but Am not Testing It?

If you are not testing Java but do have Java installed, we recommend that you disable the classpath before using Silk Test Classic. If you do not disable the classpath, Silk Test Classic checks for a Java classpath every time you run a test plan. To disable the classpath during the Silk Test Classic installation, select **None** on the **Java** dialog box. To verify that you have disabled the classpath, verify that the path to the Java extension is disabled in the *Java* variable, which is stored in the system variables.

For example, to verify that the path to the Java extension is disabled on Microsoft Windows 7, perform the following steps:

1. Click **Start** > **Control Panel**.
2. In the **Control Panel**, click **System and Security**.
3. In the **System and Security** pane, click **System**.
4. In the **System** pane, click **Advanced System Settings**.
5. In the **System Properties** dialog box, click **Environment Variables**.
6. In the **System variables** area of the **Environment Variables** dialog box, select the *Java* variable.
7. Disable the path to the Java extension, by placing an underscore at the beginning of the path.

When Should I Record Classes?

If you are using the Classic Agent, and Silk Test Classic has recognized an object in the tested Java application or applet as a `CustomWin`, record the class for this object if you need to manipulate this object.

How Do I Decide Whether to Use 4Test Methods or Native Methods?

For information on when to use 4Test methods or native methods, see *When to Use 4Test Versus Native Java Controls*.

How Can I Save the Changes I Make to javaex.ini?

This functionality is supported only if you are using the Classic Agent.

When you uninstall Silk Test Classic, the file `javaex.ini` is backed up as `javaex.bak` in the extend subdirectory of your install location. Any changes you made to `javaex.ini` can be reinstated by copying them from `javaex.bak` and pasting them into the new `javaex.ini` that is created when you reinstall Silk Test Classic.

How Can I Record AWT Menus?

You cannot use the `JavaAwtPopupMenu` class to record AWT menus. It is available for playback only. You must manually script any interaction with AWT menus.

Can I Use the Java Plug-In to Test Applets Outside My Browsers Native JVM?

For testing purposes, you can use the Java plug-in to run applets outside the native Java virtual machine of your browser.

Can I Test JavaScript Objects?

With the Classic Agent, you can use `InvokeJava` to access methods for testing JavaScript objects, if these objects reside on a Web page that contains an applet.

With the Open Agent, you can use `ExecuteJavaScript` to test anything that uses JavaScript.

Can I Invoke Java Code from 4Test Scripts?

- If you are using the Classic Agent, you can invoke Java code from 4Test scripts using the method `InvokeJava`.
- You can invoke Java code from 4Test scripts using the method `invokeMethods`, for both the Classic Agent and the Open Agent.

Testing Java SWT and Eclipse Applications with the Classic Agent

This section describes how to test Java SWT and Eclipse applications with the Classic Agent.

Suppressing Controls (Classic Agent)

This functionality is supported only if you are using the Classic Agent.

You can suppress the controls for certain classes for .NET, Java SWT, and Windows API-based applications. For example, you might want to ignore container classes to streamline your test cases. Ignoring these unnecessary classes simplifies the object hierarchy and shortens the length of the lines of code in your test scripts and functions. Container classes or ‘frames’ are common in GUI development, but may not be necessary for testing.

The following classes are commonly suppressed during recording and playback:

Technology Domain	Class
.NET	Group
Java SWT	org.eclipse.swt.widgets.Composite
	org.eclipse.swt.widgets.Group
Windows API-based applications	Group

To suppress specific controls:

1. Click **Options** > **Class Map**. The **Class Map** dialog box opens.
2. In the **Custom class** field, type the name of the class that you want suppress.
   The class name depends on the technology and the extension that you are using. For Windows API-based applications, use the Windows API-based class names. For Java SWT applications, use the fully

qualified Java class name. For example, to ignore the **SWT_Group** in a Windows API-based application, type `SWT_Group`, and to ignore to ignore the `Group` class in Java SWT applications, type `org.eclipse.swt.widgets.Group`.

3. In the **Standard class** list, select **Ignore**.
4. Click **Add**. The custom class and the standard class display at the top of the dialog box.

Java SWT Classes for the Classic Agent

Testing Web Applications with the Classic Agent

To create a web application that uses the Classic Agent, create a Generic Classic Agent project. Projects that use the Classic Agent support hierarchical object recognition only.

As a best practice, we recommend using the Rich Internet Application Web project rather than the Generic Classic Agent project type because the Web application uses the Open Agent and supports dynamic object recognition. You can create tests for both dynamic and hierarchical object recognition in your test environment. You can use both recognition methods within a single test case if necessary. Use the method best suited to meet your test requirements.

For information about new features, supported platforms and versions, known issues, and work-arounds, refer to the *Release Notes*.

Supported Controls for Web Applications

For a complete list of the controls available for record and replay of Web applications, see the `browser.inc` and `explorer.inc` files. By default, these files are located in `C:\Program Files\Silk\SilkTest\extend\`. The `browser.inc` file contains the objects that are shared by all Web browsers, for example the **Back** button on the toolbar. Objects that are unique to each browser are included in a separate file. Internet Explorer objects are contained in `explorer.inc`.

Sample Web Applications

To access the Silk Test Classic sample Web applications, go to:

- *http://demo.borland.com/gmopost*
- *http://demo.borland.com/InsuranceWebExtJS/*

API Click Versus Agent Click

This functionality is supported only if you are using the Classic Agent.

By default, Silk Test Classic issues API-based clicks rather than Agent-based clicks to improve the reliability of recorded and scripted clicks in HTML applications. An API click is generated internally by the browser, instead of the Silk Test Classic Agent. API clicks are more reliable than Agent clicks, which can click the wrong location of an object.

By default, Silk Test Classic issues API clicks instead of Agent-based clicks for the following Html classes and method combinations:

Class	Method
`HtmlCheckBox`	• `Click` • `Check` • `UnCheck`

Class	Method
	• `Toggle`
`HtmlColumn`	`Click`
`HtmlHeading`	`Click`
`HtmlImage`	`Click`
`HtmlLink`	`Click`
`HtmlMarquee`	`Click`
`HtmlPushButton`	`Click`
`HtmlRadioButton`	`Click`
`HtmlRadioList`	`Select`
`HtmlText`	`Click`
`HtmlTextField`	`Click`

API Clicks and OnClick JavaScript Events

Generally, API-based clicks behave just like the Agent-based clicks. If an HTML object has an OnClick JavaScript event, an API click should cause the event to fire as normal. However, on `HtmlText` objects an API click may not trigger an OnClick event in the same way an Agent click does.

This could happen because `HtmlText` might not map to a single element within the HTML. The API click could apply to a different element than the one containing the OnClick event. If API clicks on `HtmlText` do not start the expected events, you should use Agent clicks instead.

Testing Dynamic HTML (DHTML) Popup Menus

Silk Test Classic supports testing Dynamic HTML (DHTML) popup menus in tests that use hierarchical and dynamic object recognition; specifically for JavaScript popup menus.

- For tests that use hierarchical object recognition, to produce an accurate recording of interactions with a DHTML popup menu, you can record window declarations and record your actions.
- For tests that use dynamic object recognition, you can manually create tests since recording is not supported for dynamic object recognition.

Recording Dynamic HTML (DHTML) Popup Menus

You must enable extensions for the application that contains the JavaScript and use the Classic Agent to record dynamic popup menus.

If you want your action-based recordings to contain references to window identifiers instead of dynamic instantiations, first record the window declarations for the pages with DHTML popup menus. There are various techniques used to build DHTML popup menus and their menu hierarchies. The techniques you use affect what Silk Test Classic sees when recording window declarations. You may find that once a page is completely loaded in the browser, all of the menus and submenus are recognized immediately by Silk Test Classic. Other times, in order for the menus and submenus to be completely seen in the **Record Window Declarations** dialog box, you may need to expose some or all of the menus and submenus by moving the mouse over the menu items.

Typically when you record actions, the recorder ignores mouse movement events, which are set in the **Ignore Mouse Move Events** text box of the **Recorder Options** dialog box. However, the recorder generates `MoveMouse()` method calls as you expose popup menus. Those calls are necessary to ensure that when you play back the script, Silk Test Classic exposes the menus as it navigates through them. The

`MoveMouse()` calls contain coordinates because the hot spot of the item used to expose the menu may not be the entire rectangle for that item. Therefore, Silk Test Classic cannot assume that moving the mouse to the default spot, which is the upper-left corner of the rectangle for the item, will actually expose the menu.

Web Application Setup Steps

Before testing a Web application, take the following steps to set up Silk Test Classic for this type of testing:

- If you are using the Classic Agent, enable support for browsers and disable all non-Web extensions.
- If you are using the Open Agent, configure the Web application.
- Specify your default browser.
- Make sure your browser is configured properly.
- Set the proper agent options, if necessary.

Recording the Test Frame for a Web Application

When you record a test frame for a Web application, the results differ from those for a non-Web application.

1. Start your browser and go to the initial page of your Web application.
2. Click **File** > **New** from the menu bar.
3. Click **Test Frame** and then click **OK**. The **New Test Frame** dialog box displays.
4. If you are using the Open Agent, perform the following steps:
   a) Click **Web Site Test Configuration**. The **New Web Site Configuration** dialog box opens.
   b) From the **Browser Type** list box, select the browser type that you are using.
   c) In the **Browser Instance** section, check the appropriate check box to determine whether you want to test an application in an existing instance of the browser, or you want to start a new browser.
   d) Click **Finish**.
5. If you are using the Classic Agent, perform the following steps:
   a) Select your Web application.

      The **New Test Frame** dialog box displays the following fields:

File name	Name of the frame file you are creating. You can change the name and path to anything you want, but make sure you retain the `.inc` extension.
Application	The title of the currently loaded page.
URL	The URL of the currently loaded page.
4Test identifier	The identifier that you will use in all your test cases to qualify your application's home page. We recommend to keep the identifier short and meaningful.

   b) Edit the file name and 4Test identifier as appropriate.
   c) Click **OK**.

Recording Window Declarations for a Web Application

This functionality is supported only if you are using the Classic Agent.

Window declarations can be saved in a single file or in multiple files. If you save them in multiple files, make them available to scripts using the 4Test `use` statement.

1. Click the test frame to make it active.
2. Click **Record** > **Window Declarations**. Silk Test Classic displays the **Record Window Declarations** dialog box.

3. Load a page in your application in the browser.
4. Place your mouse pointer over the page.

   The value in the **Class** field should be `BrowserChild`, since that is the class for a page in a Web application.
5. Move your mouse pointer around the page. The values in the dialog box update to reflect the object the mouse is over.
6. Notice all the objects that Silk Test Classic recognizes. Press `Ctrl+Alt`. The contents freeze in the **Record Window Declarations** dialog box.
7. Click **Paste to Editor**. Silk Test Classic pastes the new declarations to your frame file.
8. To declare another page of the application, go to that page. Then, in the **Record Window Declarations** dialog box, click **Resume Tracking**.
9. Click **Close**.

You can use multitags when recording window declarations for Java applications. However, additional considerations must be made for top-level windows.

Modifying the identifiers in the declaration	After you have declared each of your application's pages, you will probably want to modify the identifiers to be more meaningful.
Test frame for the GMO application	We provide a complete test frame for the sample GMO application, the `gmow.inc` file. You can download a web-based version of the GMO application at *http://demo.borland.com/gmopost/*.

Streamlining HTML Frame Declarations

This functionality is supported only if you are using the Classic Agent.

As you navigate within a web site that uses frames, the GUI objects in individual frames may change independently of other frames in the same window. When you capture declarations for the new GUI objects inside a frame, Silk Test Classic re-declares the frame and the frame's own parent window. If all pages in the frame have the same caption, you will want to do the following:

1. Record a new test frame for the Web page. Silk Test Classic captures all the active HTML frames as displayed on the browser.
2. To declare other HTML frames, make the page display in the browser. This is usually done by clicking a link in an "index" type HTML frame. For example, a static HTML frame region may contain a menu bar or image map to navigate the Web site.
3. Open the newly recorded declaration, and locate the declaration for the new HTML frame. Copy this `BrowserChild` object and paste it into bottom of the declaration. This new `BrowserChild` is a sibling (at the same level) to the initial `BrowserChild` declarations recorded. Re-name this `BrowserChild` as desired for easier recognition.
4. Remove the recorded window declaration (remember you just copied the declaration for the new Html frame into the "`main/root`" `BrowserChild` declaration. This declaration and all its children should not be deleted.)

Test Frames

This section describes how Silk Test Classic uses test frames as global information repositories about the application under test.

Tags and Identifiers

This functionality is supported only if you are using the Classic Agent.

Each object in a declarations file, such as a test frame file, has a class, a tag, and an identifier. The home page has the class `BrowserChild`. Its default identifier is the name in the 4Test identifier field you

specified when you created the test frame. The tag is generated by Silk Test Classic. It is the way that Silk Test Classic identifies the page at runtime.

Overview of Test Frames

A test frame is an include file (`.inc`) that serves as a central global repository of information about the application under test. It contains all the data structures that support your test cases and test scripts. Though you do not have to create a test frame, by declaring all the objects in your application, you will find it much easier to understand, modify, and interpret the results of your tests.

When you create a test frame, Silk Test Classic automatically adds the frame file to the **Use files** field of the **Runtime Options** dialog box. This allows Silk Test Classic to use the information in the declarations and recognize the objects in your application when you record and run test cases.

When you enable extensions, Silk Test Classic adds an include file based on the technology or browser type that you enable to the **Use files location** in the **Runtime Options** dialog box. For extensions that use the Open Agent, Silk Test Classic names the include file `<technology_type>.inc`. For instance, if you enable extensions for an Apache Flex application, a file named `flex.inc` is added. If you enable extensions for an Internet Explorer browser, Silk Test Classic adds the `explorer.inc` file to the **Runtime Options** dialog box.

A constant called `wStartup` is created when you record the test frame. By assigning the identifier of the login window to `wStartup` and by recording a new `invoke` method, your tests can start the application, enter any required information into the login window, then dismiss the login window.

See *Marking 4Test Code as GUI-Specific* to learn about the ways you modify the test frame when porting your test cases to other GUIs.

The Test Frame File for a Web Application

This functionality is supported only if you are using the Classic Agent.

The test frame file includes the following:

- A constant named `wMainWindow`.
- A window of class `BrowserChild`.

wMainWindow

This constant points to the home page of your application, that is, the page that was loaded when you created the test frame. The recovery system uses `wMainWindow` to restore the browser to that page when a test fails. Just as a non-web application typically has a state where you want the tests to start (the base state), Web applications also have a base state. Typically, it is the first page in the application.

BrowserChild

The window has the same identifier as the value of `wMainWindow`. This window loads in order to restore the base state. The window declaration contains:

- The constant `sLocation`, which is the URL for the page. The recovery system uses this constant to load the page when necessary.
- Two commented constants, `sUserName` and `sPassword` which specify the user name and password to access the application. See *Specifying username and password*.
- Two commented constants, `BrowserSize` and `bDefaultFont`, which specify the size of the browser window and the default font to use for displaying text. See *Specifying browser size and fonts*.
- All the objects in the page, such as `HtmlHeadings`, `HtmlText`, `HtmlLinks`, `HtmlText`, `HtmlPushButtons`, and so on.

Specifying Browser Size and Fonts

This functionality is supported only if you are using the Classic Agent.

By default, two other built-in constants are also enclosed in comment tags in a generated test frame: `BrowserSize` and `bDefaultFont`. The recovery system uses these two constants to set the browser's state before and after Silk Test Classic runs each test. They are useful in establishing a consistent environment for your testing.

BrowserSize

Specifies the width and height of the browser window, in pixels. The data type is `POINT`, which is a record of two integers. The first integer is the width in pixels. The second integer is the height in pixels. The default value ({600, 400}) is an appropriate size for a screen with VGA resolution (640 by 480).

If you are using a higher resolution, you would want a larger size for the browser window (the larger the better, in order to minimize scrolling in the test cases). For example, if you are using a resolution of 1280 by 1024, a good `BrowserSize` might be {650, 900}, which would take up about half the width and most of the height of the desktop.

bDefaultFont

If this constant is set to TRUE, the recovery system will restores the fonts to the shipped defaults for the browser, as described in `SetDefaultFont`.

Using these constants

To have the recovery system set the size and fonts, un-comment the constants in the test frame and specify appropriate values for `BrowserSize`.

Having the recovery system set the browser size and fonts ensures maximum portability of your window declarations in different testing sessions and between browsers. We strongly recommend that you un-comment these constants and use the recovery system for your Web testing.

Specifying Username and Password

This functionality is supported only if you are using the Classic Agent.

The two built-in constants, *sUserName* and *sPassword*, are enclosed in comment tags in the generated test frame. You can un-comment those constants and supply values to specify your user name and password, if your application requires you to enter such information. Once you have done this, whenever you are loading a page, and you are prompted for user name and password, Silk Test Classic will automatically supply the values and click **OK** in the message box. The test case can run unattended.

Modifying the Identifiers

Identifiers are arbitrary strings. You use identifiers to identify objects in your scripts. Tags, on the other hand, are not arbitrary and should not be changed except in well-specified ways.

To make your tests easier to understand and maintain, you can change your objects' identifiers to correspond to their meaning in your application. Then when Silk Test Classic records tests, it will use the identifiers that you specify.

User Options

This section describes how you can set user options for DOM extensions and table recognition.

Setting DOM Extension Options

This functionality is supported only if you are using the Classic Agent.

There are three different ways to set options for the DOM extension.

- In the **DOM Extensions** dialog box.
- By using `BrowserPage.SetUserOption()` in a script.
- By editing the values in the `[Options]` section of the `domex.ini` file.

Depending on where you set the option, the option can be set globally, or turned on and off at various points of your testing.

Setting an option in `domex.ini` or in the **DOM Extensions** dialog box sets it globally. However, if you want to set the option only at certain points in your script, use `BrowserPage.SetUserOption()` as described in `SetUserOption()`.

Regardless of how an option is set, you can read its value by using `GetUserOption()`.

Options you can set in `domex.ini`

To set these options, you enter them on a line in the `[Options]` section of `domex.ini`, for example:

```
ScrollListItemIntoView=FALSE
```

You can set the following options in `domex.ini`, but you cannot set them with `SetUserOption`():

DOMWaitForBrowser	Default is 10000 milliseconds (or 10 seconds). The value you set is expressed in milliseconds. This option specifies how long Silk Test Classic will wait for the browser to complete an action. If the browser fails to respond within the given time, Silk Test Classic will try to force a ready-state. If this fails too, an error will occur.
IgnoreDivTags	Default is FALSE. If HTML controls nested between the `<DIV>` and `<\DIV>` tags are not recognized, set this option to TRUE to ignore the `<DIV>` and `<\DIV>` tags.
ReturnListContentsPropertyAsString	Default is FALSE meaning that normally the `$Contents` property for `HtmlList` objects returns a LIST OF STRING. Set this option to TRUE if you want `$Contents` to return a STRING for `HtmlList` objects.
SetActiveXBrowserStateActive	Default is FALSE. Set this option to TRUE, if Silk Test Classic does not recognize the properties and methods of an ActiveX control in the browser. Setting the option to TRUE forces the DOM extension to behave as if the browser is ready when recognizing the ActiveX control, even though the browser `Document complete` message was not received.
ScrollListItemIntoView	Default is TRUE. Set this option to FALSE to avoid scrolling a list box item or `PopupList` item into view before selecting it.
ShowHTCViewlink	Default is FALSE. Set this option to TRUE to allow the DOM extension to look for HTC ViewLinks. Setting this option to TRUE slows performance for recording and playback.
UseDocumentEvents	Default is FALSE. For more information, see *HtmlPopupList Causes the Browser to Crash when Using IE DOM Extension*.
XMLNodeNamingVersion	Default is 0. For more information, see *XMLNode Class*.

Options that you can set in domex.ini and through SetUserOption

To set the following options, you can edit `domex.ini` or you can use `SetUserOption`():

IgnoreSpanTags	Default is FALSE. If Silk Test Classic recognizes multiple text objects as one `HtmlText` object and the object is a SPAN object that is parented to a SPAN object, set this option to TRUE to ignore the <SPAN> and < \SPAN> tags.
RowTextIncludesEmptyCells	Controls the recognition of blank cells in bordered and borderless tables. This option is set to FALSE by default. If you want to return blank cells in tables as empty strings, set this option to TRUE.
ShowBodyText	Default is FALSE meaning that the DOM extension does not record `BodyText` objects, which are text that is not contained within an HTML tag. In previous releases body text displayed as `HtmlText`. Set this option to TRUE if you do want the DOM extension to record `BodyText` objects. We suggest keeping this option set to FALSE for improved performance, particularly when recording window declarations on large pages. You can also set this option on the DOM Extensions dialog box.
ShowBorderlessTableFlags	Indicates input elements that you do not want Silk Test Classic to consider as input elements; for details, see *Overview of Input Elements and Borderless Tables*.
ShowBorderlessTables	Default is .5 meaning that the DOM extension does record `BorderlessTable` objects. However, .76 is the threshold where Silk Test Classic starts to recognize more objects within tables, such as images, hidden text, check boxes, text fields, and buttons. Set this option to .76 or greater if you want the DOM extension to record `BorderlessTable` objects. You can also set this option on the DOM Extensions dialog box.
ShowHtmlForm	Default is FALSE meaning that the DOM extension does not record Form objects. Set this option to TRUE if you do want the DOM extension to record Form objects. You can also set this option on the DOM Extensions dialog box.
ShowInvisible	Default is TRUE, meaning that invisible objects are recorded. This option lets you control whether or not invisible objects are recorded by the DOM extension. If your browser-based application consists of pages that contain many invisible objects that you do not need to test, then you can improve performance by setting the option to FALSE in order to ignore all invisible objects. You can also set this option on the **DOM Extensions** dialog box.
ShowHtmlHidden	Default is TRUE meaning that the DOM extension records Hidden objects. Set this option to FALSE if you do not want the DOM extension to record Hidden objects. You can also set this option on the **DOM Extensions** dialog box.
ShowHtmlMeta	Default is TRUE meaning that the DOM extension records Meta objects. Set this option to FALSE if you do not want the DOM extension to record Meta objects. You can also set this option on the **DOM Extensions** dialog box.
ShowHtmlTable	default is TRUE meaning that the DOM extension records `HtmlTable` objects. Set this option to FALSE if you do not want the DOM extension to record `HtmlTable` objects. You can also set this option on the **DOM Extensions** dialog box.
ShowHtmlText	Default is TRUE meaning that the DOM extension records Text objects. Set this option to FALSE if you do not want the DOM extension to record Text objects. You can also set this option on the DOM Extensions dialog.

If you are testing a transaction type page with lots of text consider not recording Text objects. This prevents Silk Test Classic from recording the many text objects, which helps your declarations to be clean. If, on the other hand, you're looking for formatting and styles of text objects, you'll want to select this option.

ShowListItem

Default is TRUE. Set this option to FALSE if you do not want to show the text contained within `HtmlList` controls in your browser. If mouse events are associated with your list items, set this option to TRUE so Silk Test Classic can interact with the list items. When set in the `domex.ini` file or **DOM Extensions** dialog box, this setting is global. However, if you want to set this option for only certain points in your script, use `BrowserPage.SetUserOption()` as described in `SetUserOption()`.

ShowOverflow

Default is TRUE meaning that Silk Test Classic recognizes elements with overflow styles. These elements are very similar to IFrame elements in that they have their own scrollbar and can contain their own elements. Set this option to FALSE if you want Silk Test Classic to ignore elements with overflow styles; this means that Silk Test Classic may not interact with the elements contained by this overflow element.

ShowVirtualColumns

Default is FALSE. Affects how the DOM extension records asymmetric tables. These are tables that use either column span or row span attributes, or tables whose rows don't have the same number of columns. An example of an asymmetric table is a typical calendar page that has the month of January written across the top row and the seven days of the week in seven columns across the second row. We recommend that you check this box if you are working with tables that have asymmetrical rows. Check this check box if you want Silk Test Classic to create virtual columns for any row in a table. In the example below, it causes the top row to contain one real column for `January`, followed by six virtual columns which are blank. These virtual columns appear where there are none in order to complete the table and they are named virtual1, virtual2, and so on. These virtual columns cause the table to be symmetrical. Uncheck this check box to avoid creating virtual columns. This causes Silk Test Classic to record the top row as the name for first column. This occurs because there is no second column in the top row; Mon is promoted to the name of the second column, and so on. You can also set this option on the **DOM Extensions** dialog box.

```
January
Sun Mon Tues Weds Thurs Fri Sat
```

SearchWholeDOMTree

Default is FALSE. This check box determines how windows declarations are found. If this value is set to TRUE, when recording window declarations, the search algorithm of current objects searches the whole DOM tree.

ShowXML

Default is TRUE meaning that the DOM extension records XML objects. Set this option to FALSE if you do not want the DOM extension to record XML objects. You can also set this option on the **DOM Extensions** dialog box.

UseBrowserClosestText

Default is FALSE. Determines how the DOM extension finds the closest static text for HtmlTable, HtmlLink-text, HtmlColumn, HtmlLink-image, HtmlImage, HtmlHeading, HtmlText, HtmlRadioList, and HtmlPushbutton. Check this check box if you want Silk Test Classic to use the DOM extension to find the closest static text for the objects listed above. This does not apply to invisible objects such as XML, Meta, and Hidden; those

	objects do not rely on any text on a page and so it would be meaningless to try to associate them with any objects. Uncheck this check box if you want to use the Agent to determine closest static text for the objects listed above. You can also set this option on the **DOM Extensions** dialog box.
UseOverflowScrolling	Default is FALSE. Set this option to TRUE if off-screen HTML elements with overflow do not scroll into view properly.
UseScrollIntoView	Default is FALSE. This option is useful if you are having problems scrolling objects into view. This sometimes happens on HTML pages that contain scrollable iframes or scrollable HTCs. In general, the action/testcase recorder does not record the content inside an HTC. For other nested scrollable objects such as IFrames and overflow elements, the flashing rectangle displays in the wrong place, but the action/testcase recorder does generate correct script actions. If you are having problems playing back actions against nested scrollable objects, then set UseScrollIntoView to TRUE. Setting this option to TRUE helps avoid getting nested scrollable objects into view. **Note:** API-based clicking is not affected by the value of this option.

User Options for Table Recognition

This functionality is supported only if you are using the Classic Agent.

The following user options control table recognition:

RowTextIncludesEmptyCells	Controls the recognition of blank cells in bordered and borderless tables. This option is set to FALSE by default. If you want to return blank cells in tables as empty strings, set : `BrowserPage.SetUserOption ("RowTextIncludesEmptyCells", TRUE)`.
ShowBorderlessTables	Changes the level of recognition of tables in your web pages. See *Setting options for ShowBorderlessTables*.
ShowBorderlessTableFlags	Indicates input elements that you do not want to consider as input elements. See *Overview of input elements and borderless tables*.

Testing Methodology for Web Applications

You test Web applications with using the same methodology as when you test standalone and client/server applications. Testing of both Web-based applications and non-Web-based applications includes the following test phases:

- Creating and working with test plans.
- Designing and recording test cases.
- Running tests and interpreting results.
- Debugging test cases.
- Generalizing test cases.
- Handling exceptions.
- Making test cases data-driven.
- Customizing Silk Test Classic.

Testing Web Applications on Different Browsers

One of the challenges of testing Web applications is that your users will probably be using different browser types and your application must support the browsers that your users use. When you develop tests for Web applications running on different browser types, you must decide:

- How your test cases will handle differences between browsers.
- How to specify which browser to use for the test case or test script.

Handling differences between browsers

In most cases, your include files (declarations) and scripts apply to any browser. You can run test cases against different browsers by simply changing the default browser and running the test case, even if the pages look a bit different, such as pushbuttons being in different places. Because Silk Test Classic is object-based, it doesn't care about layout. It just cares what objects are on the page.

There may be times when declarations and scripts have one or more lines that apply only to particular browsers. In these situations you can use `browser specifiers` to make lines specific to one or more browsers. Browser specifiers are of the built-in data type `BROWSERTYPE`.

VO Automation

This section describes how you can convert your automation to use the DOM extension, because Silk Test Classic no longer supports the VO extension.

Information for Current Customers that Are Using VO

This functionality is supported only if you are using the Classic Agent.

Silk Test Classic no longer supports the VO extension and you must convert your automation to use the DOM extension. If you simply enable the DOM extension, there is no guarantee that previously written scripts will run without modification.

Using the DOM extension will require you to make changes to your scripts and include files. The number of automation changes will depend on how you have structured your tests and is hard to predict, since everyone has their own style of structuring their automation.

There are differences in the way DOM and VO recognize objects and so you will have to make script changes regardless of which way you choose to use the DOM. For example, if you use the DOM extension, the caption tag as well as the window ID tag might be different for some objects than if you used VO. This means you would have to at least update your window declarations and possibly update your scripts.

Changing Existing VO Automation to the DOM Extension

This functionality is supported only if you are using the Classic Agent.

Perform the following steps to port your current VO automation to the DOM extension:

1. Make sure the DOM extension is on.
2. Record a new window declaration for the particular objects with which you are having difficulties.
3. Look for any differences in the identifier names and object hierarchy. Make changes in the include file and scripts where appropriate.

If you decide to use the DOM exclusively, use one of these two approaches after you turn on the DOM extension:

Approach #1 Run your scripts and look for a `"Window '[class] Tagname' was not found"` error message. Look for any differences in the identifier names and object hierarchy. Repeat this until all errors are resolved.

Approach #2 Re-record all of your declarations. Look for any differences in the identifier names and object hierarchy. Make changes in the include file and scripts where appropriate. This approach could potentially be more work than is necessary. Exactly how much work depends on how you have structured your automation.

Comparison of DOM and VO

In many respects, the VO and DOM extensions provide similar functionality, but the following differences are worth noting:

	DOM Extension	VO Extension
Properties	The value of exposed properties cannot be set	The value of exposed properties can be set.
Tags	In the DOM, the tags may vary compared to VO. This could apply for check boxes, radio lists, and text boxes. For additional information, refer to the *Release Notes*.	Always uses the closest static text.
Tables	Recognizes all tables, bordered and borderless, as they are defined. By default, support for some borderless tables is turned on for some borderless tables. You may edit the level of support for tables.	Recognizes all bordered tables as they display. Cannot distinguish between borderless tables used for aligning objects and borderless and bordered tables used for presenting content.
Classes	Includes the following additional classes: • HtmlForm • HtmlHidden • HtmlMarquee • XMLNode	
Embedded links	Records embedded links twice. The first time, it records the text of the link (the HTML <.a ref>. tag); the second time, it records the text of the jump.	Records just the link as an HtmlLink object.

Testing Objects in a Web Page

This section describes how you can test the objects in a Web page.

Document Object Model Extension

Silk Test Classic uses the Document Object Model (DOM) extension of Internet Explorer which uses information in the HTML source to recognize and manipulate objects on a Web page.

Advantages of DOM

The Document Object Model (DOM) extension has several advantages:

- By default, when you are using the DOM extension the recorder displays a rectangle which highlights the controls as you are recording them.
- The DOM extension is highly accurate, because it gets information directly from the browser. For example, the DOM extension recognizes text size and the actual name of objects.

- The DOM extension is independent of the browser size and text size settings.
- The DOM extension will find non-GUI, which means non-visible, objects. For example, if you are using the Classic Agent, the DOM extension will find objects of the types `HtmlMeta`, `HtmlHidden`, `XML Node`, and `HtmlForm`.
- The DOM extension offers support for borderless tables.
- The DOM extension is consistent with the standard being developed by the W3C.

Useful Information About DOM

Internet Explorer

- When you use the DOM extension with Internet Explorer, in order to interact with a browser dialog box, the dialog box must be the active (foreground) window. If another application is active, then Silk Test Classic is not able to interact with the browser dialog box, and the test case times out with an `Application not ready` exception.
- You may receive a `Window not found error` when you are running scripts using the DOM extension. This error occurs when the test case calls `Exists`() on the browser page before it is finished loading. This problem is due to the fact that the DOM extension does not check for DOM Ready in the `Exists`() method. The workaround is to call `Browser`.`WaitForReady`() in your script, prior to the `Exists`() method.
- If you are using the Classic Agent, see the `GetProperty` method and `GetTextProp` method for information about how Silk Test Classic recognizes tags.
- If you are using the Classic Agent, you may see differences in image tags based on the same URL if you used two different URLs to get there. For example, Silk Test Classic cannot differentiate between two images if Internet Explorer displays two different URLs that both point to the same image.
- The DOM extension does not record inside a secure frame. This means that if an Html page contains frames with security, for example on a banking page, the DOM extension on Internet Explorer will not be able to record the window declaration for the page because the secure site prevents DOM from getting any information.

Mozilla Firefox

There are several things to remember when you work with Mozilla Firefox and XML User-interface Language (XUL). XUL is a cross-platform language for describing user interfaces of applications. The support of XUL elements in Silk Test Classic is limited. All menu, toolbar, scrollbar, status bar and most dialog boxes are XUL elements. Almost all elements in the browser are XUL elements except the area that actually renders HTML pages.

- If you are using the Classic Agent, you can record window declarations on the menu and toolbar by pointing the cursor to the caption of the browser.
- You can record actions and test cases against the menu and toolbar through mouse actions.
- If you are using the Classic Agent, you can record window declarations on a single frame XUL dialog box, such as the authentication dialog box. However, you cannot record window declarations on a multi-framed XUL dialog box, for example, the preference dialog box.
- Silk Test Classic does not support:
  - Keyboard recording on the menu and toolbar. There is no keyboard recording on the URL.
  - Record actions and record test case on XUL dialog boxes.
  - Record identifier and location on XUL elements.

For information about new features, supported platforms and versions, known issues, and work-arounds, refer to the *Release Notes*.

Recording and playback

- When you record using the Internet Explorer DOM extension, a rectangle will flash to indicate the current target object of the recorder.

- Silk Test Classic can recognize XML Node objects in your Web page if your Web page contains XML content.
- The DOM extension supports HTML Components (HTCs), including those implemented using the `viewLink` property.
- It is a limitation of DOM that it cannot see the location of any text that is a direct descendant of the `<body>` tag. `GetRect()` does not work for body text. For example, when you record window declarations with the Classic Agent over body text, you do not get any objects. This was implemented for HTML pages where no <p> tags or other text formatting tags preface the displayed text.
- DOM cannot find an insertion on a multi-line text field.
- If you are using the Classic Agent, images created with the `<input type="image">` tag are seen as `HtmlPushButtons`.
- If you are using the Classic Agent and you open a font statement on a Web page with several `HtmlText` fields and `HtmlCheckbox` controls, but do not close it off, the DOM extension will not recognize anything beyond the first object. Closing off the font statement with a `</font>` tag enables Silk Test Classic to work correctly.
- The DOM extension is not designed to handle multiple links with the same file name. If you do have multiple links, be sure to use the full URL to identify links.
- If you are using the Classic Agent to test Html pages that do not have explicit titles and which load ActiveX applications, you may have to modify test frames that you have previously recorded using the VO Extension before you can use them with the DOM extension. This is because the DOM extension tags the `BrowserChild` slightly differently. Alternatively you could record new declarations for the page.
- If you are using the Classic Agent, the `GetPosition()` function of the `TextField` class always returns the last position when called on an `HtmlTextField`. There is no method in the DOM which allows Silk Test Classic to get the cursor position in an `HtmlTextField`.
- If you are using the Classic Agent to record a window declaration over a table that has indented links, the indentation is recorded as an additional `HtmlText` object.
- If you are using the Classic Agent and you are recording with the DOM extension, `TypeKeys ("<Tab>")` are not captured. Since the script refers to the object to type in directly, it is not necessary to record this manual Tab. You can manually enter a `TypeKeys ("<Tab>")` into your script if you want to; it just is not recorded.
- For additional information about Silk Test Classic's rules for object recognition, refer to *Object Recognition with the Classic Agent*. To open the document, click **Start** > **Programs** > **Silk** > **Silk Test** > **Documentation** > **Silk Test Classic** > **Tutorials**.

The 4Test language and the DOM extension

- If you are using the Classic Agent, use the `ForceReady` method when Silk Test Classic never receives a `Document complete` message from the browser. Unless Silk Test Classic receives the `Document complete` message, Silk Test Classic acts as if the browser is not ready and will raise an `Application not ready` error.
- For a list of the supported classes for the DOM extension on each agent, see *Differences in the Classes Supported by the Open Agent and the Classic Agent*.
- If you are using the Classic Agent, use the `FlushCache` method of the `BrowserChild` class to re-examine the currently loaded page and to get any new items as they are generated. This method is very useful when you are recording dynamic objects that may not initially display.

Dynamic Tables

This functionality is supported only if you are using the Classic Agent.

If columns have objects in them, such as links or controls, Silk Test Classic declares each of the links and controls when you record window declarations for the column.

If your application dynamically builds tables, such that you do not know at runtime how many rows there will be and consequently do not know how many objects there will be, you should not declare individual

objects in tables. You should remove their declarations from those that Silk Test Classic creates when you declare a window.

You can use the `GetRowChildren` method to get a list at runtime of all children (controls and objects) in a specified row of a table or column.

How Silk Test Classic Declares HTML Frames

This functionality is supported only if you are using the Classic Agent.

HTML frames are multiple web pages displayed concurrently in a browser. Each HTML frame is an independent scrollable region. The HTML frames form a hierarchy defined by the main web page. The main web page lays out the regions and the web pages associated with these regions. In more complex HTML designs, frames may be further nested.

Silk Test Classic recognizes each frame as a `BrowserChild` and nests the `BrowserChild` objects as defined by the HTML frame hierarchy. The main web page is the root object and is recognized as `BrowserChild` object which may contain children of type `BrowserChild`.

Silk Test Classic declares each frame on a web page as a child of the main window. Silk Test Classic derives the identifier and tag of `BrowserChild` object from the title element in the html source. If the title element does not exist, then Silk Test Classic will use the main page's title plus an index to identify each `BrowserChild`.

By default, Silk Test Classic derives the identifier and tag of an HTML frame from its caption. In turn, the caption of an HTML frame is derived from the first text contained in the frame, such as an `HtmlText` or `HtmlHeading` element. If there is no text in an HTML frame, Silk Test Classic derives the identifier from the frame's Window ID and the tag from the frame's index.

Testing Columns and Tables

- If you are using the Classic Agent, tables in Web applications are recognized as `HtmlTable` controls. An `HtmlTable` consists of two or more `HtmlColumn` controls.
- If you are using the Open Agent, tables in Web applications are recognized as `DomTable` controls. Rows in a table are recognized as `DomTableRow` controls.

Definition of a Table

Classic Agent

If you are using the Classic Agent, the definition of a table in HTML is the following:

- An `HtmlTable` with 2 or more rows, which are specified with the `<tr>` tag in the page source.
- Where at least 1 row has 2 or more columns, which are specified with the `<td>` tag in the page source.

A single `<td>` with a `colspan > 1` does not qualify as 2 or more columns.

If a table with insufficient dimensions is nested inside other tables, then the parent tables of this table are not recognized as `HtmlTable` controls, even if these parent tables have sufficient dimensions.

If a table does not meet this definition, Silk Test Classic does not recognize it as a table. For example, if a table is empty, which means that it has no rows or columns, and you attempt to select a row by using `table.SelectRow (1, TRUE, FALSE)`, you will get an error message saying `E_WINDOW_NOT_FOUND`, when you might expect to see a message such as `E_ROW_INDEX_INVALID` instead.

Open Agent

If you are using the Open Agent, the definition of a table is a `DomTable`, which is a DOM element that is specified using the `<table>` tag.

Testing Controls

Web applications can contain the same controls as standard applications, including the following:

Control	Classic Agent Class	Open Agent Class
check box	`HtmlCheckBox`	`DomCheckBox`
combo box	`HtmlComboBox`	No corresponding class.
list boxes	`HtmlListBox`	`DomListBox`
popup lists	`HtmlPopupList`	`DomListBox`
pushbuttons	`HtmlPushButton`	`DomButton`
radio lists	`HtmlCheckBox`	`DomCheckBox`

All these classes are derived from their respective standard class. For example, `HtmlCheckBox` is derived from `CheckBox`. So all the testing you can do with these controls in standard applications you can also do in Web applications.

Classic Agent Example

The following code gets the list of items in the credit card list in the **Billing Information** page of the sample GMO application:

```
LIST OF STRING lsCards
lsCards = BillingPage.CreditCardList.GetContents ()
ListPrint (lsCards)

Result:
American Express
MasterCard
Visa
```

Open Agent Example

The following code gets the list of items in the credit card list in the **Billing Information** page of the sample GMO application:

```
LIST OF STRING lsCards
lsCards = WebBrowser.BrowserWindow.CardType.Items
ListPrint(lsCards)

Result:
American Express
MasterCard
Visa
```

Testing Images

Classic Agent

If you are using the Classic Agent, images in your Web application are objects of type `HtmlImage`. You can verify the appearance of the image by using the **Bitmap** tab in the **Verify Window** dialog box.

If an `HtmlImage` is an image map, which means that the image contains clickable regions, you can use the following methods to test the clickable regions:

- `GetRegionList`

- `MoveToRegion`
- `ClickRegion`

Open Agent

If you are using the Open Agent, you can test images by using the `IMG` locator. For example, the following code sample finds an image and then prints some of the properties of the image:

```
Window img = FindBrowserApplication("/
BrowserApplication").FindBrowserWindow("//BrowserWindow").Find("//
IMG[@title='Image1.png']")
String src = img.GetProperty("src")
String altText = img.GetProperty("alt")
print(src)
print(altText)
```

Testing Links

- If you are using the Classic Agent, links in your application are objects of type `HtmlLink`.
- If you are using the Open Agent, links in your application are objects of type `DomLink`.

Silk Test Classic provides several methods that let you get their text properties as well as the location to which they jump.

Classic Agent Example

The following code returns the definition for the `HtmlLink` on a sample home page:

```
STRING sJump
sJump = Acme.LetUsKnowLink.GetLocation ()
Print (sJump)

Result:
mailto:support@acme.com
```

Open Agent Example

The following code returns the definition for the `DomLink` on the sample home page:

```
STRING sJump
sJump =
WebBrowser.BrowserWindow.LetUsKnowLink.GetProperty("href")
Print(sJump)

Result:
mailto:support@acme.com
```

Testing Text in Web Applications

Classic Agent

Straight text in a Web application can be in the following classes:

- `HtmlHeading`
- `HtmlText`

Silk Test Classic provides methods for getting the text and all its properties, such as color, font, size, and style.

There are also classes for text in Java applets and applications.

Classic Agent Example

For example, the following code gets the copyright text on a sample Web page:

```
STRING sText
sText = Acme.Copyright.GetText ()
Print (sText)

Result:
Copyright © 2006 Acme Software, Inc. All rights reserved.
```

Open Agent

When you are using the Open Agent, use the `GetText`() method to get text out of every `DomElement` control.

Open Agent Example

For example, the following code gets the text of a `DomLink` control:

```
Window link = FindBrowserApplication("/BrowserApplication")

                    .FindBrowserWindow("//BrowserWindow")
                     .FindDomLink("A[@id='story2128000']")
String linkText = link.GetText()
print(linkText)
```

Tips on how Silk Test Classic Recognizes Objects in Browsers

This functionality is supported only if you are using the Classic Agent.

The following notes describe how Silk Test Classic recognizes objects in browsers with the Document Object Model (DOM) extension:

- DOM uses the `name` attribute for input elements as the object's window ID. This makes object recognition for input objects independent of the way those objects appear in a browser.
- If you are using the Classic Agent, an `HtmlHeading` must be tagged with `<H1>` through `<H6>` to be found as text. If the text is tagged with `<TH>(table header)`, the text is identified as a header if it is in the first row, or as `HtmlText` otherwise. If the text is simply bold it is considered simply a row and `GetTextProp("$FontStyle")` will record `FS_BOLD`. If you have bold text, DOM does not interpret that text as headings.
- If you are using the Classic Agent, the `GetText` method returns the first line as defined by any existing line break characters. For example `<BR>`. Because the DOM extension does not offer a visual interpretation of browser content, `GetText` always returns the same value regardless of browser size, font size, or browser.
- When you use the DOM extension, Silk Test Classic attempts to group HTML text objects into one 4Test text object. However, Silk Test Classic will separate objects if it encounters `<br>` tags. This means if you use `<br>` tags within your HTML pages, Silk Test Classic may record more text objects than you expect. This is because with the DOM extension, Silk Test Classic considers text separated by `<br>` tags as separate objects. For example:

  ```
  <p>
  Welcome
  <br>
  and Opening Remarks
  </p>
  ```

  You might expect this to be recorded as one object, but Silk Test Classic records this as two.
- If you are using the Classic Agent, the DOM extension records both an `HtmlImage` and an `HtmlLink` for an image.

- If you spawn an additional browser window, Silk Test Classic sees the second browser as another `BrowserPage`, which means that you have to set the window active before interacting with it. This will ensure that you are working with the correct browser window.
- With the DOM extension, Silk Test Classic captures only the first text style within a paragraph and assumes that the captured style applies to the whole paragraph.

For additional information about the Silk Test Classic rules for object recognition, refer to *Object Recognition with the Classic Agent* and *Silk Test Classic Quick Start Tutorial for Dynamic Object Recognition*. To access these tutorials, click **Start** > **Programs** > **Silk** > **Silk Test** > **Documentation** > **Silk Test Classic** > **Tutorials**.

Testing Borderless Tables

This functionality is supported only if you are using the Classic Agent.

Borderless tables are used to present content, but more commonly they are used to align text, figures, and other objects on a Web page. Often, borderless tables are nested within other borderless tables, many levels deep. Depending on the Web page, you may want to test only those tables that visually appear to be tables on the Web page, that is, those with actual borders. At other times, you might want to test a borderless table that presents content.

To meet your varying needs, the IE DOM extension has an option that allows you to set the level of recognition of tables. This is optional. You do not need to specify a value in order to use the DOM extension.

Overview of Input Elements and Borderless Tables

This functionality is supported only if you are using the Classic Agent.

You can use the `ShowBorderlessTableFlags` option in `domex.ini` to indicate input elements which you do not want Silk Test Classic to consider as input elements.

This feature is provided as a convenience to you, but it has not yet been thoroughly tested.

To describe the HTML input element, you must use the tag you'd use in HTML. For example, setting `ShowBorderlessTableFlags=img` indicates that a borderless table having at least 1 image input element, described by the `<img>` HTML tag, is considered to have NO input elements at all, even if other input elements such as push buttons are contained in the table. This flag is implemented using OR functionality. For example:

```
ShowBorderlessTableFlags=img|input
```

means that any HTML table with EITHER `<img>` tag(s) OR `<input …>` tag(s) is considered to have NO input elements. To be even more specific about the type of input element you want to describe, you can use the values that are permitted for the "type" attribute of the input tag in HTML. For example,

```
ShowBorderlessTableFlags=submit
```

means that the presence of the HTML construct `<input type=submit>`, which creates a submit button, causes Silk Test Classic to consider a table having this tag as having NO input elements.

Note:

- Input elements are: `HtmlTextField`, `HtmlImage`, `HtmlPushButton`, `HtmlPopupList`, `HtmlRadioButton`,`HtmlCheckBox`, `HtmlListBox`, and `HtmlHidden`.
- These settings do not apply to bordered tables. All bordered tables are recognized as tables.
- Except for the value of 1, all tables must meet the basic definition of a table.
- If performance is a consideration for you, consider setting the value for borderless tables to zero or 1. That causes Silk Test Classic to find either no tables or all tables and your scripts will run faster.

- If the value for `ShowBorderlessTables` is set to less than .75, the `ShowBorderlessTableFlags` option is set, and the value matches an element in any cell of a table, then Silk Test Classic will show that table.
- If the value for `ShowBorderlessTables` is set to less than .75, the `ShowBorderlessTableFlags` option is not set, and a cell contains an input element (`IMG`, `SELECT`, `INPUT`, `BUTTON` AND `TEXTAREA`), then Silk Test Classic will not show the table. It will ignore it.

Guidelines to Recognizing Borderless Tables

This functionality is supported only if you are using the Classic Agent.

Silk Test Classic, by default, uses `ShowBorderlessTables=.5`. At this setting, a borderless table which contains one or more input elements will not be considered to be an HtmlTable by Silk Test Classic. Additionally, at the .5 setting, if a borderless table containing an input element is part of a set of nested borderless tables, none of the tables that contain that table , regardless of their content, will be considered to be an `HtmlTable`.

For this discussion, input elements are defined as:

- HtmlTextField
- HtmlImage
- HtmlPushButton
- HtmlPopupList
- HtmlRadioButton
- HtmlCheckBox
- HtmlListBox
- HtmlHidden

To change the level of recognition of tables in your web pages, you must set the value of the `ShowBorderlessTables` option.

Below are two charts that describe the guidelines for recognizing borderless tables on a Web page in Silk Test Classic.

Use the chart below if the borderless table in question does not contain input elements.

Does the table have nested tables?	Do any of the nested tables have input elements?	Level of nested tables	To recognize this table as an HtmlTable, set ShowBorderlessTables to:
No	N/A	N/A	0 < x < .30
Yes	No	1	.30 < x <.60
Yes	No	2	.60 < x < .75
Yes	No	> 2	1
Yes	Yes	Irrelevant	1

Use the chart below if the borderless table in question does contain input elements.

Does the table have nested tables?	Do any of the nested tables have input elements?	Level of nested tables	To recognize this table as an HtmlTable, set ShowBorderlessTables to:
No	N/A	N/A	.75<=x<.9
Yes	No	1 or 2	.75<=x<.9
Yes	No	3	.91<=x<.99

Does the table have nested tables?	Do any of the nested tables have input elements?	Level of nested tables	To recognize this table as an HtmlTable, set ShowBorderlessTables to:
Yes	No	> 3	1
Yes	Yes	Irrelevant	1

Notes

All values between the suggested ranges in the preceding tables are the same. For example, it does not make any difference if you use 0 or 0.28 as borderless table value. Silk Test Classic will ignore all borderless HtmlTables if the level is set to less than 0.0001, if the level is set to greater than or equal to .75, Silk Test Classic will show all borderless tables.

Levels of Recognition for Borderless Tables

This functionality is supported only if you are using the Classic Agent.

The following chart describes in general how the ShowBorderlessTable values affect the tables that Silk Test Classic recognizes.

To recognize	Then set the value to
no borderless tables as 4Test HtmlTables	ShowBorderlessTables=0
all borderless tables, regardless of content or dimensions, as 4Test HtmlTables.	ShowBorderlessTables=1

If performance is a consideration for you, consider setting the value for borderless tables to 0 or 1. Silk Test Classic will find no or all tables and your scripts will run faster.

Setting Options for ShowBorderlessTables

This functionality is supported only if you are using the Classic Agent.

To change the level of recognition of tables in your Web pages, you must set the value of the `ShowBorderlessTables` option. There are two ways to do this.

- On the DOM Extensions dialog box, check the **Table** check box and set a value for borderless table recognition. .76 is the threshold where Silk Test Classic starts to recognize more objects within tables, such as images, hidden text, check boxes, textfields, and buttons.
- Tweak your script by setting the option within the script itself. This does not apply the value to the whole script but applies the value after that point in your script. You can use this method to adjust the level of recognition within your scripts, as you require. You do this by entering the following into your script: `BrowserPage.SetUserOption ("ShowBorderlessTables",value)`. For example, `BrowserPage.SetUserOption ("ShowBorderlessTables",.5)`.

Setting the ShowListItem Option

To change the level of recognition of text contained within `HtmlList` controls in your browser, you must set the value of the ShowListItem option. For instance, if mouse events are associated with your list items, check this check box or set this value to TRUE, so Silk Test Classic can interact with the list items. You can do this in the following ways:

- On the **DOM Extensions** dialog box, check the **List Item** check box. Access this dialog box by clicking **Options** > **Extensions**, enabling the browser extension in the **Primary Extension** column, and then clicking **Extension** in the **Options** area. Note that the browser extension must be enabled before you can click **Extension**. The information that you enter on this dialog box is saved in the `domex.ini` file. This setting is global. However, if you want to set this option for only certain points in your script, use `BrowserPage.SetUserOption()`.

- In the domex.ini file, set the ShowListItem value to TRUE. This setting is global. However, if you want to set this option for only certain points in your script, use BrowserPage.SetUserOption().
- Modify your script by setting the option within the script itself. This does not apply the value to the whole script but applies the value after that point in your script. You can use this method to adjust the level of recognition within your scripts, as you require. You do this by entering the following into your script:

```
BrowserPage.SetUserOption ("ShowListItem",  true)
```

Tag Declaration for SSTab Control

This functionality is supported only if you are using the Classic Agent.

If your application contains an SSTab control, which is associated with the class OLESSTab, you must use either the index or window ID for the tag in the window declaration. You cannot use the caption for the tag, because the caption changes based on which tab is selected.

Testing XML

This functionality is supported only if you are using the Classic Agent.

You can use Silk Test Classic to verify that different renderings of your Web page display the same XML data. For example, if you change the presentation layer of your website, you can use Silk Test Classic to "see" through the new presentation and test the XML data.

4Test Class

To support testing XML data, the 4Test language was modified to include the XmlNode. Users can access the XML Elements through properties and methods that have been defined within this new class.

Identifiers and Tags with XML Objects

This functionality is supported only if you are using the Classic Agent.

When creating identifiers and tags for XML objects, Silk Test Classic first looks to the objects value. If no value is declared Silk Test Classic takes the objects name/content. This is most important to remember in terms of Attributes. For example, the following is the source for some information about a book in a book catalog:

```
<Book BookType="Fiction" BookISBN ="x0682">
```

From this example we see that this book has two attributes: it is of BookType Fiction and its Book ISBN is x0682. The declaration for this appears as follows:

```
[-] XmlNode Book1
 [+] multitag "Book [1]"
 [-] XmlNode FICTION
  [+] multitag "FICTION"
  [-]XmlNode X0682
   [+] multitag "x0682"
```

Note: Silk Test Classic grabs the values Fiction and x0682 as the identifiers and tags.

Window Declarations for XML

This functionality is supported only if you are using the Classic Agent.

When Silk Test Classic captures the window declarations for an XML page, it first captures the presentation layer, the HTML objects that represent the XML data. Then below the declared HTML objects, the XML objects are declared.

Setting Options for XML Recognition

This functionality is supported only if you are using the Classic Agent.

To recognize XML elements:

1. Click **Options** > **Extensions** to display the **Extension** dialog.
2. Click in the **Primary Extension** column for the browser that you want to use, then select the browser name from the list box to enable the DOM extension.
3. In the **Options** area, click **Extension**.
4. On the **DOM Extension** dialog, check the **XML** check box and click **OK**.
5. Close the **Option Extensions** dialog, click **OK**.
6. Close the **Extensions** dialog, click **OK**.

To turn off XML recognition, uncheck the check box you checked in step 4 above.

General Web Application Classes

This section lists the classes that you can use to test Web applications with the Classic Agent.

This functionality is supported only if you are using the Classic Agent.

Testing Windows API-Based Applications

This section describes how Silk Test Classic provides built-in support for testing Microsoft Windows API-based applications.

Overview of Windows API-Based Application Support

Silk Test Classic provides built-in support for testing Microsoft Windows API-based applications. Several objects exist in Microsoft applications that Silk Test Classic can better recognize if you enable Accessibility. For example, without enabling Accessibility Silk Test Classic records only basic information about the menu bar in Microsoft Word and the tabs that display in Internet Explorer 7.0. However, with Accessibility enabled, Silk Test Classic fully recognizes those objects. You can also improve Silk Test Classic object recognition by defining a new window, if necessary.

You can test Windows API-based applications using the Classic or Open Agent.

For information about new features, supported platforms and versions, known issues, and work-arounds, refer to the *Release Notes*.

Object Recognition

Windows API-based applications support hierarchical object recognition and dynamic object recognition. You can create tests for both dynamic and hierarchical object recognition in your test environment. Use the method best suited to meet your test requirements.

When you record a test case with the Open Agent, Silk Test Classic creates locator keywords in an INC file to create scripts that use dynamic object recognition and window declarations.

Existing test cases that use dynamic object recognition without locator keywords in an INC file will continue to be supported. You can replay these tests, but you cannot record new tests with dynamic object recognition without locator keywords in an INC file.

To test Windows API-based applications using hierarchical object recognition, record a test for the application that you want to test. Then, replay the tests at your convenience.

Supported Controls

For a complete list of the record and replay controls available for Windows-based testing for each Agent type, view the `WIN32.inc` and `winclass.inc` file. To access the `WIN32.inc` file, which is used with the Open Agent, navigate to the `<SilkTest directory>\extend\WIN32` directory. By default, this file is

located in `C:\Program Files\Silk\SilkTest\extend\WIN32\WIN32.inc`. To access the `winclass.inc` file, which is used with the Classic Agent, navigate to the `<SilkTest directory>\` directory. By default, this file is located in `C:\Program Files\Silk\SilkTest\winclass.inc`.

Locator Attributes for Windows API-Based Applications

Silk Test Classic supports the following locator attributes for the controls of Windows API-based client/server applications:

- `caption`.
- `windowid`.
- `priorlabel`. For controls that do not have a caption, `priorlabel` is used as the caption automatically. For controls with a caption, it may be easier to use the caption.

Note: Attribute names are case sensitive. Attribute values are by default case insensitive, but you can change the default setting like any other option. The locator attributes support the wildcards ? and *.

Suppressing Controls (Classic Agent)

This functionality is supported only if you are using the Classic Agent.

You can suppress the controls for certain classes for .NET, Java SWT, and Windows API-based applications. For example, you might want to ignore container classes to streamline your test cases. Ignoring these unnecessary classes simplifies the object hierarchy and shortens the length of the lines of code in your test scripts and functions. Container classes or 'frames' are common in GUI development, but may not be necessary for testing.

The following classes are commonly suppressed during recording and playback:

Technology Domain	Class
.NET	Group
Java SWT	org.eclipse.swt.widgets.Composite
	org.eclipse.swt.widgets.Group
Windows API-based applications	Group

To suppress specific controls:

1. Click **Options** > **Class Map**. The **Class Map** dialog box opens.
2. In the **Custom class** field, type the name of the class that you want suppress.

   The class name depends on the technology and the extension that you are using. For Windows API-based applications, use the Windows API-based class names. For Java SWT applications, use the fully qualified Java class name. For example, to ignore the **SWT_Group** in a Windows API-based application, type `SWT_Group`, and to ignore to ignore the `Group` class in Java SWT applications, type `org.eclipse.swt.widgets.Group`.
3. In the **Standard class** list, select **Ignore**.
4. Click **Add**. The custom class and the standard class display at the top of the dialog box.

Suppressing Controls (Open Agent)

This functionality is supported only if you are using the Open Agent.

You can suppress the controls for certain classes for win32 applications. For example, you might want to ignore container classes to streamline your test cases. Ignoring these unnecessary classes simplifies the object hierarchy and shortens the length of the lines of code in your test scripts and functions.

To suppress specific controls:

1. Click **Options** > **Recorder**. The **Recording Options** dialog box opens.
2. Click the **Transparent Classes** tab.
3. Type the name of the class that you want to ignore during recording and playback into the text box.

   If the text box already contains classes, add the new classes to the end of the list. Separate the classes with a comma. For example, to ignore both the `AOL Toolbar` and the `_AOL_Toolbar` class, type `AOL Toolbar, _AOL_Toolbar` into the text box.

   The OPT_TRANSPARENT_CLASSES option is set to true for these classes.
4. Click **OK**. The OPT_TRANSPARENT_CLASSES option is set to true for these classes, which means the classes are added to the list of the classes that are ignored during recording and playback.

Configuring Standard Applications

A standard application is an application that does not use a Web browser, such as a Windows application or Java SWT application.

Configure the application that you want to test to set up the environment that Silk Test Classic will create each time you record or replay a test case.

1. Start the application that you want to test.
2. Click **Configure Application** on the basic workflow bar.

   If you do not see **Configure Application** on the workflow bar, ensure that the default agent is set to the Open Agent.

   The **Select Application** dialog box opens.
3. Select the **Windows** tab.
4. Select the application that you want to test from the list.

   Note: If the application that you want to test does not appear in the list, uncheck the **Hide processes without caption** check box. This option, checked by default, is used to filter only those applications that have captions.
5. *Optional:* Check the **Create Base State** check box to create a base state for the application under test.

   By default, the **Create Base State** check box is checked for projects where a base state for the application under test is not defined, and unchecked for projects where a base state is defined. An application's base state is the known, stable state that you expect the application to be in before each test begins execution, and the state the application can be returned to after each test has ended execution. When you configure an application and create a base state, Silk Test Classic adds an include file based on the technology or browser type that you enable to the **Use files location** in the **Runtime Options** dialog box.
6. Click **OK**.

   - If you have checked the **Create Base State** check box, the **Choose name and folder of the new frame file** page opens. Silk Test Classic configures the recovery system and names the corresponding file `frame.inc` by default.
   - If you have not checked the **Create Base State** check box, the dialog box closes and you can skip the remaining steps.
7. Navigate to the location in which you want to save the frame file.
8. In the **File name** text box, type the name for the frame file that contains the default base state and recovery system. Then, click **Save**. Silk Test Classic creates a base state for the application and opens the include file.

9. Record the test case whenever you are ready.

Note: For SAP applications, you must set **Ctrl+Alt** as the shortcut key combination to use. To change the default setting, click **Options** > **Recorder** and then check the **OPT_ALTERNATE_RECORD_BREAK** check box.

Determining the priorLabel in the Win32 Technology Domain

To determine the priorLabel in the Win32 technology domain, all labels and groups in the same window as the target control are considered. The decision is then made based upon the following criteria:

- Only labels either above or to the left of the control, and groups surrounding the control, are considered as candidates for a priorLabel.
- In the simplest case, the label closest to the control is used as the priorLabel.
- If two labels have the same distance to the control, the priorLabel is determined based upon the following criteria:
  - If one label is to the left and the other above the control, the left one is preferred.
  - If both levels are to the left of the control, the upper one is preferred.
  - If both levels are above the control, the left one is preferred.
- If the closest control is a group control, first all labels within the group are considered according to the rules specified above. If no labels within the group are eligible, then the caption of the group is used as the priorLabel.

Testing Applications with the SilkBean

This functionality is supported only if you are using the Classic Agent.

Using the SilkBean, you can test standalone Java applications on non-Windows platforms, such as UNIX and Linux. You can perform cross-platform testing of 100% pure Java controls in standalone Java applications in a number of test environments. SilkBean provides flexibility that enables you to:

- Test a single standalone Java application on a non-Windows target machine.
- Set up multiple testing sessions on the same Windows host machine to test multiple standalone Java applications on the same non-Windows target machine.
- Set up multiple testing sessions on different Windows hosts to test multiple standalone Java applications on the same target machine.

When using SilkBean, you create all functions using Silk Test Classic on your Windows host machine, and then play back the scripts on a non-Windows target machine running SilkBean.

SilkBean runs on the following certified platforms:

- Solaris 2.5 or later.
- Redhat Linux 6.0 and 6.2.
- Hewlett-Packard UNIX (HP-UX) 10.2 and 11.0.
- Advanced Interactive Executive (AIX) 4.3.2 and 4.3.3.

For information about new features, supported platforms and versions, known issues, and work-arounds, refer to the *Release Notes*.

Preparing Test Scripts to Run with SilkBean

This functionality is supported only if you are using the Classic Agent.

When you are preparing test scripts to run with the SilkBean, keep the following tips in mind:

- Do not use ~ActiveApp in window declarations that will be used with SilkBean. Either use a generic parent tag, such as `[JavaMainWin]#1/` in the tag statement, or use a tag function if the parents can occur at different levels in the window hierarchy. For details see the multitag statement topic.
- **Note:** In certain cases where dialog boxes are parented to other dialog boxes, the window hierarchy may differ between SilkBean and Windows. As for ~ActiveApp, use a tag function to compensate for the different levels of the parent windows.
- Do not call `Desktop.GetActive()` in your scripts, since it is invalid for SilkBean.
- The index tag of a MoveableWin, for example a main window or dialog box, may differ between SilkBean and the standard Java extension. The difference is not due to a difference in operating systems, like UNIX and Windows, but rather to a difference between the Java extension and SilkBean. For example, the Java extension may see the top visible window as "#1", but SilkBean may see the bottom-most window, which was the first one created, as "#1".
- If you must use an index tag, then you can use a conditional expression in the tag to accommodate both the Java extension and SilkBean. The condition should be based on the value of the OPT_USE_SILKBEAN Agent option, which is TRUE if SilkBean is currently being used. For example, if the index for the Java extension is #1 and the index for SilkBean is #3, use:
  ```
  tag "#{Agent.GetOption (OPT_USE_SILKBEAN) ? "3" : "1"}"
  ```
- Insert a right-mouse Click() before JavaAwtPopupMenu or JavaJFCPopupMenu in order to bring up the menu.

Additional considerations when testing AWT

- If your application contains Abstract Windowing Toolkit (AWT) menus or AWT menu items, modify your declarations for these controls in a new test frame file as follows:

If the AWT declaration is …	Change the declaration to …
Menu	JavaAwtMenu
MenuItem	JavaAwtMenuItem

- If you are testing AWT controls, do not use low-level methods to simulate mouse and keyboard events. Due to the limitations of platform-specific implementations of AWT controls, low-level events are not supported. Instead, use high-level methods when possible.

Java Foundation Class (JFC) controls support low-level events.

Configuring SilkBean Support on the Target (UNIX) Machine

This functionality is supported only if you are using the Classic Agent.

This section contains instructions for configuring the target UNIX machine when the test application is running in Java 2 environments, which means JDK/JRE versions greater than or equal to 1.2.

1. Make sure that JDK 1.2 or later is installed on the UNIX machine and that the path to its "bin" directory is included in the PATH variable.
2. Copy the following files from your Silk Test Classic installation on the Windows machine onto the UNIX machine:
   - Copy `SilkTest_Java3.jar` to the JVM's `lib/ext` directory.
   - Copy `access3bean.prop` to the JVM's lib directory and rename it to `accessibility.properties`.
3. Start the SilkBean using the following command:
   ```
   java segue.server.SilkBean debug <port number> &
   ```

- Include the optional `debug` parameter if you want to run the SilkBean server in debug mode.
- The port number defaults to 2966, if it is not specified.
- The ampersand (&) at the end of the line should only be used on an UNIX target machine. It specifies that the SilkBean should run in the background.

4. Start the test application manually or from the script.

Note: There are two SilkBean-specific options for the Java command line that is used to start the AUT. The options are specified using the '-D' switch.

ST_CONN_TIMEOUT	The maximum time (in seconds) allowed for connection between the SilkBean and the application. If unspecified, the default value of 30 seconds is used.
qap.port	The port through which to connect SilkBean. The default is 2966. This number must match the port number specified in the SilkBean command line. `java segue.server.SilkBean <port number> &`.

Example

For example, to start the Java application `myapp.jar` with port number 2970 and a connection timeout of 60 seconds, use:

```
java -Dqap.port=2970 -DST_CONN_TIMEOUT=60 myapp.jar
```

Configuring SilkBean Support on the Host Machine when Testing Multiple Applications

This functionality is supported only if you are using the Classic Agent.

If you are testing multiple applications on the same machine, or different platforms, you must make the following changes on the host machine to enable the Agent to interact with the SilkBean running on the UNIX machine:

Enable SilkBean by adding the following code to the script file:

```
Agent.SetOption(OPT_USE_SILKBEAN, FALSE)
Agent.SetOption(OPT_SET_TARGET_MACHINE, "targetmachine:port#"))
Agent.SetOption(OPT_USE_SILKBEAN, TRUE)
```

Correcting Problems when Using the SilkBean

This functionality is supported only if you are using the Classic Agent.

General help

For general help when testing Java applications with the SilkBean, you can start up the SilkBean in debug mode on the target machine. Then, look for AppRegistered debug messages to display after invoking your Java application.

Specific workarounds

There are workarounds for the following specific problems:

- I cannot test the AWT FileOpen dialog box.
- I cannot pick AWT menus.
- `SetActive`, `Minimize`, and `Restore` methods do not work with SilkBean.
- I cannot select menu items in JFC cascaded menus.

- SilkBean cannot find a main window or a dialog box declared with an index tag.
- I cannot redirect console output to a text file.

Using Advanced Techniques with the Classic Agent

This section describes advanced techniques for testing you applications with Silk Test Classic and the Classic Agent.

Starting from the Command Line

This section describes how you can start Silk Test Classic from the command line.

Starting Silk Test Classic from the Command Line

You can start the Silk Test Classic executable program from the command line by:

- Clicking **Run** in the **Start** menu.
- Using the command-line prompt in a DOS window or batch file.

The syntax is:

```
Partner [-complog filename] [-m mach] [-opt optionset.opt] [-p mess] [-proj
filename [-base filename]] [[-q] [-query query name] [-quiet] [-r filename] [-
resexport] [-resextract] [-r] scr.t/suite.s/plan.pln/link.lnk [args]]
```

The `filename` specified for various options expects the file to be located in the working directory (the default location is the Silk Test Classic install directory, `c:\Program Files\Silk\SilkTest\`). If you want to use a file that is located in another directory, you must specify the full path in addition to the filename.

Options

The following table lists all the options to the `partner` command.

args

Optional arguments to a script file. You can access the arguments using the `GetArgs` function and use them with your scripts.

If you pass arguments in the command line, the arguments provided in the command line are used and any arguments specified in the currently loaded options set are not used. To use the arguments in the currently loaded options set, do not specify arguments in the command line.

For more information, see *Passing arguments to a script*.

-complog

Tells Silk Test Classic to log compilation errors to a file you specify. Enter the argument as `-complog filename`. For example: `partner [-complog c:\testing \logtest1.txt]`.

If you include this argument, each time you do a compilation Silk Test Classic checks to see if the file you named exists. If it does not already exist, Silk Test Classic creates and opens it. If the file already exists, Silk Test Classic opens it and adds the information. The number of errors is written in the format `n error(s)`, for example `0 errors`, `1 error`, or `50 errors`. Compilation errors are written to the error log file as they are displayed in the "Errors" window. The error log file is automatically saved and closed when Silk Test Classic finishes writing errors to it.

-m Specifies the target machine. The default is the current machine. Call the 4Test built-in function `Connect` to connect to a different machine at runtime.

In order to use the `-m` switch, you need to have the **Network setting** of the **Runtime Options** dialog box set to `TCP/IP` or `NetBIOS`. If this is set to `'(disabled)'`, the target machine is ignored. To set the **Network setting**, either set it interactively in the **Runtime Options** dialog box before running from the command line, or save the setting in an option set and add the `'-opt <option set>'` argument to the command line.

-opt Specifies an options set. Must be followed by the path of the `.opt` file you want to use.

-p Provided for use with a Windows shell program that is running Silk Test Classic as a batch task. The option enables another Windows program to receive a message containing the number of errors that resulted from the scripts run. Silk Test Classic broadcasts this message using the Windows PostMessage function, with the following arguments:

- `hWnd = HWND_BROADCAST`
- `uiMsg = RegisterWindowMessage (mess)`
- `wParam = 0`
- `lParam = number of errors`

To take advantage of the `-p` option, the shell program that runs Silk Test Classic should first register mess, and should look for mess while Silk Test Classic is running.

-proj Optional argument specifying the project file or archived project to load when starting Silk Test Classic or Silk Test Classic Runtime. For example, `partner –proj d:\temp \testapp.vtp –r agent.pln`.

`-base` is an optional argument to `–proj`. You use the base argument to specify the location where you want to unpack the package contents. For example, `partner –proj d:\temp\testapp.stp –base c:\rel30\testapp` unpacks the contents of the package to the `c:\rel30\testapp directory`.

-q Quits Silk Test Classic after the script, suite, or test plan completes.

-query Specifies a query. Must be followed by the name of a saved query. Tells Silk Test Classic to perform an **Include** > **Open All**, then **Testplan** > **Mark By Named Query**, then **Run** > **Marked Tests**.

-quiet Starts Silk Test Classic in "quiet mode", which prevents any pop-up dialog boxes from displaying when Silk Test Classic starts up.

The quiet option is particularly useful if you are doing unattended testing where a user is not available to respond to any pop-up dialog boxes that may display.

-r Must be the last option specified, followed only by the name of a Silk Test Classic file to open. This includes files such as script (and, optionally, arguments that the script takes), a suite, test plan, or link file. If you specify a link file, tells Silk Test Classic to resolve the link and attempt to open the link target. Otherwise, tells Silk Test Classic to run the specified script, suite, or test plan, optionally passing args as arguments to a script file. For example, `partner –proj d:\temp\testapp.stp –base c:\rel30\testapp –r Agent.pln` unpacks the archive from the temp subdirectory into the `c:\rel30\testapp` subdirectory and then loads and executes the `Agent.pln` file.

-resexport Tells Silk Test Classic to export a one line summary of the most recent results sets to `.rex` files automatically. Specifying `-resexport` has the same effect as if each script run invokes the `ResExportOnClose` function during its execution.

-resextract Tells Silk Test Classic to extract all information from the most recent results sets to a `.txt` file. Both the Silk Test Classic Extract menu command and the `-resextract` option create UTF-8 files.

script.t/ suite.s/ plan.pln/ link.lnk The name of the Silk Test Classic script, suite, test plan, or link file to load, run, or open.

Examples

To load Silk Test Classic, type: `partner`

To run the test.s suite, type: `partner -r test.s on system "sys1"`

To run the test.t script, type: `partner -m sys1 -r test.t`

To run the test.t script with arguments, type: `partner -r test.t arg1 arg2`

To run the tests marked by the query named query3 in tests.pln, type: `partner -query query3 -r tests.pln`

To run tests.pln, and export the most recent results set from tests.res to tests.rex, type: `partner -q -resexport -r tests.pln`

To edit the test.inc include file, type: `partner test.inc`

Starting the Classic Agent from the Command Line

This functionality is available only for projects or scripts that use the Classic Agent.

You can start the Classic Agent executable program from the command line by:

- Clicking **Start**>**Run**.
- Using the command-line prompt in a DOS window.

The syntax is:

```
agent [-p port]
```

Options

The following option is available for the `agent` command:

-p <port> Temporarily set network protocol to TCP/IP and port to specified <port>. For the Classic Agent, the default port is the value in the `partner.ini` file. If there is no value set in the `partner.ini` file, then the port is set to 2965. To permanently set a protocol and port, right-click the Agent window and select **Network**.

Port numbers may not be negative numbers or _D.

Examples

To load the Classic Agent, enter:

```
agent
```

To load the Classic Agent and specify network protocol to TCP/IP and port to 2965 (the default port for the Classic Agent), enter:

```
agent -p
```

To load the Classic Agent and specify network protocol to TCP/IP and port to 1234, enter:

```
agent –p 1234
```

Recording a Test Frame

This section describes how you can record a test frame.

Overview of Object Files

Object files are the compiled versions of include (`.inc`) or script (`.t`) files. Object files are saved with an "o" at the end of the extension, for example, `.ino`, or `.to`. Object files cannot be edited; the only way to change compiled objects is to recompile the include or script file. When you save a script or include file, a source file and an object file are saved. Object files are not platform-specific; you can use them on all platforms that Silk Test Classic supports.

In order for Silk Test Classic to run a script or include file that is in source form, it must compile it, which can be time-consuming. Object files, on the other hand, are ready to run.

Note: You cannot call objects that exist in the object file (`.to`) from a test plan; you must have the script file (`.t`).

To disable saving object files during compilation, the **AutoComplete** options on the **General Options** dialog box as well as the **Save object files during compilation** option on the **Runtime Options** dialog box need to be unchecked.

Silk Test Classic always uses object files if they are available. When you open a script file or an include file, Silk Test Classic loads the corresponding object file as well, if there is one. If the object file is not older than the source file, Silk Test Classic does not recompile the source file. The script is ready to run. If the source file is more recent, Silk Test Classic recompiles the source file before the script is run. If you then later save the source file, Silk Test Classic automatically saves a new object file.

If a file is loaded during compilation, that is, if you include a file in another file that is being compiled, Silk Test Classic loads only the object file, if it exists and is newer than the corresponding source file.

Object files may not be backward-compatible, although sometimes they will be. Specifically, object files will not work with versions of Silk Test Classic for which the list of GUI/browser types is different than for the version used to compile the object file. The list is in `4Test.inc`. For example, object files created before 'mswxp' was added as the GUI type for Windows XP cannot be used with ST5.5 SP3, which includes the 'mswxp' GUI type.

If you are using a `.ino` file, but during compilation Silk Test Classic displays a message that the corresponding .inc file is missing, then you may be experiencing the object file version incompatibility explained in the preceding paragraph.

Advantages of Object Files

Advantages of object files include:

- Because object files are ready to run, they do not need to be recompiled if the source file has not changed. This can save you a lot of time. If your object file is more recent than your source file, the source file does not need to be recompiled each time the file is first opened in a session; the object file is used as is.
- You can distribute object files without having to distribute the source equivalents. So if you have built some tests and include files that you want to distribute but don’t want others to see the sources, you can distribute only the object files.

Since an object file cannot be run directly:

- Define the code you want to "hide" in an include file, which will be compiled into an `.ino` object file.
- Call those functions from an ordinary script file.
- Distribute the `.t` script file and the compiled `.ino` include file. Users can open and run the script file as usual, through **File** > **Run**.

Here's a simple example of how you might distribute object files so that others cannot see the code.

In file `test.inc`, place the definition of a function called `TestFunction`. When you save the file, the entire include file is compiled into `test.ino`.

```
TestFunction ()
           ListPrint (Desktop.GetActive ())
```

In the file `test.t` use the `test.inc` include file. Silk Test Classic will load the `.ino` equivalent. Call `TestFunction`, which was defined in the include file.

```
use "test.inc"

        main ()
        TestFunction () // call the function
```

Distribute `test.t` and `test.ino`. Users can open `test.t` and run it but do not have access to the actual routine, which resides only in compiled form in `test.ino`.

Object File Locations

By default, an object file is read from and written to the same directory as its corresponding source file. But you can specify different directories for object files.

Specifying `d:\obj` in the **Objfile Path** text box of the **Runtime Options** dialog box tells Silk Test Classic to read and write all object files in the `d:\obj` directory, regardless of where the source files are located.

Specifying obj in the **Objfile Path** text box tells Silk Test Classic to read and write an object file in the directory obj that is a subdirectory of the directory containing the source file. In this scenario, each directory of source files will have a different directory of object files. For example, if a source file is in `d:\src`, its corresponding object file would be read from and written to `d:\src\obj`.

You can specify several directories in the **Objfile Path** text box. New files are written to the first directory specified. Silk Test Classic searches the directories in the order in which you have specified them to find existing files and will subsequently re-save existing files in the same directory where it found them.

Specifying where Object Files Should be Written To and Read From

By default, an object file is read from and written to the same directory as its corresponding source file. But you can specify different directories for object files. To specify where object files are written to and read from:

1. Click **Options** > **Runtime**.
2. Specify a directory in the **Objfile Path** text box.
   - Leave the text box empty if you want to store object files in the same directory as their corresponding source files.
   - Specify an absolute path if you want to store all object files in the same directory.
   - Specify a relative path if you want object files to be stored in a directory relative to the directory containing the source files.
3. Click **OK**.

Object files are saved in the location you specify here. In addition, Silk Test Classic will try to find object files in these locations. If it fails to find an object file, it will look in the directory containing the source file.

Declarations

This section describes declarations.

Generic Message Box Declaration

This functionality is available only for projects or scripts that use the Classic Agent.

When Silk Test Classic generates the window declarations for the main window of your application, it also includes a declaration for a generic object named `MessageBox`. Therefore, you do not have to record a declaration for each of the message boxes (potentially hundreds) in your application.

A message box is a dialog box that has static text and pushbuttons, but no other controls. Typically, message boxes are used to prompt users to verify an action, for example `Save changes before closing?`, or to alert users to an error.

The message box declaration is generic for three reasons:

- The tag of the dialog box specifies that its parent is the current active application.
- The most likely names for pushbuttons are accounted for: **OK**, **Cancel**, **Yes**, and **No**.
- The tag of the message is an index number, not the text of the message.

If your application contains message boxes that have extra pushbuttons or if your pushbuttons use different names, you need to add the declarations for those buttons to the declaration for the generic `MessageBox` object. For example, if a message box contains a Test pushbutton, you need to add the following lines to the recorded declaration:

```
PushButton Test
   tag "Test"
```

Here is the declaration for the generic message box:

```
window MessageBoxClass MessageBox
   tag "~ActiveApp/[DialogBox]$MessageBox"
   PushButton OK
      tag "OK"
   PushButton Cancel
      tag "Cancel"
   PushButton Yes
      tag "Yes"
   PushButton No
      tag "No"
   StaticText Message
      mswnt tag "#2"
      tag "#1"
```

GUI Specifiers

Where Silk Test Classic can detect a difference from one platform to the next, it automatically inserts a **GUI-specifier** in a window declaration to indicate the platform, for example `msw`.

For a complete list of the valid GUI specifiers, see *GUITYPE data type*.

Overview of Dialog Box Declarations

The declarations for the controls contained by a dialog box are nested within the declaration of the dialog box to show the GUI hierarchy.

The declarations for menus are nested (indented) within the declaration for the main window, and the declarations for the menu items are nested within their respective menus. This nesting denotes the hierarchical structure of the GUI, that is, the parent-child relationships between GUI objects. Although a dialog box is not physically contained by the main window, as is true for menus, the dialog box

nevertheless logically belongs to the main window. Therefore, a parent statement within each dialog box declaration is used to indicate that it belongs to the main window of the application.

In the sample Text Editor application, `MainWin` is the parent of the `File` menu. The `File` menu is considered a child of the `MainWin`. Similarly, all the menu items are child objects of their parent, the `File` menu. A child object belongs to its parent object, which means that it is either logically associated with the parent or physically contained by the parent.

Because child objects are nested within the declaration of their parent object, the declarations for the child objects do not need to begin with the reserved word window.

Classic Agent Example

The following example from the Text Editor application shows the declarations for the **Find** dialog box and its contained controls:

```
window DialogBox Find
   tag "Find"
   parent TextEditor
   StaticText FindWhatText
     multitag "Find What:"
       "$65535"
   TextField FindWhat
     multitag "Find What:"
       "$1152"
   CheckBox CaseSensitive
     multitag "Case sensitive"
       "$1041"
   StaticText DirectionText
     multitag "Direction"
       "$1072"
   RadioList Direction
     multitag "Direction"
       "$1056"
   PushButton FindNext
     multitag "Find Next"
       "$1"
   PushButton Cancel
     multitag "Cancel"
       "$2"
```

Open Agent Example

The following example from the Text Editor application shows the declarations for the **Find** dialog box and its contained controls:

```
window DialogBox Find
  locator "Find"
  parent TextEditor
  TextField FindWhat
    locator "@caption='Find What:' or @windowId='65535'"
  StaticText FindWhatText
    locator "@caption='Find What:' or @windowId='1152'"
  CheckBox CaseSensitive
    locator "@caption='Case sensitive' or @windowId='1041'"
  StaticText DirectionText
    locator "@caption='Direction' or @windowId='1072'"
  RadioList Direction
    locator "@caption='Direction' or @windowId='1056'"
  PushButton FindNext
    locator "@caption='Find Next' or @windowId='1'"
```

```
  PushButton Cancel
    locator "@caption='Cancel' or @windowId='2'"
```

Main Window and Menu Declarations

The main window declaration

The main window declaration begins with the 4Test reserved word window. The term window is historical, borrowed from operating systems and window manager software, where every GUI object, for example main windows, dialogs, menu items, and controls, is implemented as a window.

As is true for all window declarations, the declaration for the main window is composed of a class, identifier, and tag or locator.

Classic Agent Example

The following example shows the beginning of the default declaration for the main window of the Text Editor application:

```
window MainWin TextEditor
  multitag "Text Editor"
        "$C:\PROGRAMFILES\<SilkTest install directory>\SILKTEST
\TEXTEDIT.EXE"
```

Part of Declaration	Value for TextEditor's main window.
Class	MainWin
Identifier	TextEditor
Tag	Two components in the multiple tag: • " Text Editor "—The application's caption • " executable path "—The full path of the executable file that invoked the application

Open Agent Example

The following example shows the beginning of the default declaration for the main window of the Text Editor application:

```
window MainWin TextEditor
  locator "Text Editor"
```

Part of Declaration	Value for TextEditor's main window.
Class	MainWin

Part of Declaration	Value for TextEditor's main window.
Identifier	TextEditor
Locator	" Text Editor "—The application's caption

sCmdLine and wMainWindow constants

When you record the declaration for your application's main window and menus, the *sCmdLine* and *wMainWindow* constants are created. These constants allow your application to be started automatically when you run your test cases.

The *sCmdLine* constant specifies the path to your application's executable. The following example shows an *sCmdLine* constant for a Windows environment:

```
mswnt const sCmdLine = "c:\program files\<SilkTest install directory>\silktest
\textedit.exe"
```

The *wMainWindow* constant specifies the 4Test identifier for the main window of your application. For example, here is the definition for the *wMainWindow* constant of the Text Editor application on all platforms:

```
const wMainWindow = TextEditor
```

Menu declarations

When you are working with the Classic Agent, the following example from the Text Editor application shows the default main window declaration and a portion of the declarations for the File menu:

```
window MainWin TextEditor
  multitag "Text Editor"
    "$C:\PROGRAM FILES\<SilkTest install directory>\SILKTEST\TEXTEDIT.EXE"
    .
    .
    .
    Menu File
      tag "File"
      MenuItem New
        multitag "New"
          "$100"
```

Menus do not have window IDs, but menu items do, so by default menus are declared with the tag statement while menu items are declared with the `multitag` statement.

When you are working with the Open Agent, the following example from the Text Editor application shows the default main window declaration and a portion of the declarations for the File menu:

```
window MainWin TextEditor
  locator "Text Editor"
    .
    .
    .
    Menu File
      locator "File"
      MenuItem New
        locator "@caption='New' or windowId='100'"
```

Window Declarations

This section describes how you can use a window declaration to specify a cross-platform, logical name for a GUI object, called the identifier, and map the identifier to the object's actual name, called the tag or locator.

Overview of Window Declarations

A window declaration specifies a cross-platform, logical name for a GUI object, called the `identifier`, and maps the identifier to the object's actual name, called the `tag` or `locator`. Because your test cases use logical names, if the object's actual name changes on the current GUI, on another GUI, or in a localized version of the application, you only need to change the tag in the window declarations. You do not need to change any of your scripts.

You can add variables, functions, methods, and properties to the basic window declarations recorded by Silk Test Classic. For example, you can add variables to a dialog box declaration that specify what the tab sequence is, what the initial values are, and so on. You access the values of variables at runtime as you would a field in a record.

After you record window declarations for the GUI objects in your application and insert them into a declarations file, called an include file (`*.inc`), Silk Test Classic references the declarations in the include file to identify the objects named in your test scripts. You tell Silk Test Classic which include files to reference through the **Use Files** field in the **Runtime Options** dialog box.

Improving Silk Test Classic Window Declarations

The current methodology for identifying window declarations in Microsoft Windows-based applications during a recording session is usually successful. However, some applications may require an alternate approach of obtaining their declarations because their window objects are invisible to the Silk Test Recorder. You can try any of the following:

- Turning on Accessibility - use this if during a session started with the Recorder, Silk Test Classic is unable to recognize objects within a Microsoft Windows-based application. This functionality is available only for projects or scripts that use the Classic Agent.
- Defining a new window - use this if turning on Accessibility does not help Silk Test Classic to recognize the objects. This functionality is available only for projects or scripts that use the Classic Agent.
- Creating a test case that uses dynamic object recognition - use this to create test cases that use XPath queries to find and identify objects. Dynamic object recognition uses a **Find** or **FindAll** method to identify an object in a test case. This functionality is available only for projects or scripts that use the Open Agent.

Improving Object Recognition by Defining a New Window

If Silk Test Classic is having difficulty recognizing objects in Internet Explorer or Microsoft Office applications, try enabling Accessibility. If that does not help improve recognition, try defining a new window.

How defined windows works

When you use Defined Window, you use the mouse pointer to draw a rectangle around the object that Silk Test Classic cannot record and then assign a name to the object. When you save your work, Silk Test Classic stores the name and the object's coordinates in a test script. When you replay the script, Silk Test Classic uses a `Click()` method on the center of the area you have specified.

Notes

- Defining a new window is only available for projects or scripts that use the Classic Agent.
- Defining a new window is not available for Java applications or applets.

- Defined Window does not support nesting of defined objects.
- Defined Window is location-based and uses pixel coordinates to locate the object in the parent window. Thus, if the layout of your parent window changes and/or the object's coordinates change frequently, you may need to re-define the window in order for Silk Test Classic to correctly declare the object.
- If you draw a rectangle around an unrecognized object, but also include an object that Silk Test Classic easily recognizes, Silk Test Classic records both and lists the easily recognized object first.

Recording Window Declarations for the Main Window and Menu Hierarchy

1. Start your application.
2. Click **File** > **New** in Silk Test Classic.
3. Click **Test Frame** and then click **OK**. Silk Test Classic displays the **New Test Frame** dialog box.
4. If you are using the Open Agent, follow the appropriate wizard to select your application, depending on whether you want to test an application that uses a Web browser or not. When you have stepped through the wizard, the **Choose name and folder of the new frame file** dialog box opens.
5. In the **Frame filename** (Classic Agent) or the **File name** (Open Agent) text box, accept the default test frame name (`frame.inc`), or type a new name.

   By default, Silk Test Classic names the new test frame file `frame.inc`, denoting it is an include file that contains declarations. If you change the default name of the file, make sure to include the file extension .inc in the new file name. If you do not, the file is not identified to Silk Test Classic as an include file and Silk Test Classic will give it a .txt extension and report a compilation error when you click **OK** to create the file.
6. If you are using the Classic Agent, select your application from the **Application** list box.

   The **Application** list box displays all applications that are open and not minimized. If your test application is not listed, click **Cancel**, open your application, and click **File** > **New** again.
7. Click **OK** (Classic Agent) or **Save** (Open Agent). Silk Test Classic creates the new test frame file. Window declarations display in the test plan editor, which means that the declarations for individual GUI objects can be expanded to show detail, collapsed to hide detail, and edited if necessary.

Recording a Window Declaration for a Dialog Box

After you record your test application's main window and menus, you record all the dialog boxes you want to test. Use this procedure once for each dialog box in your application.

This functionality is available only for projects or scripts that use the Classic Agent.

1. Make sure that the test frame (`.inc`) file that contains the declarations for the application's main window is open.

   The dialog box declarations will be appended to this file.
2. Click **Record** > **Window Declarations**.
3. Make your application active and invoke one of its dialog boxes, referred to in this procedure as the target dialog box. If necessary, arrange windows so that you can see the target dialog box and position the cursor on the title bar of the target dialog box.

   Note: As you move the cursor toward the title bar, the contents of the **Window Declaration** list box change dynamically to reflect the object at which you're pointing, as well as any contained objects. When the cursor is positioned correctly, the **Window Detail** group box (upper left) shows the caption of the dialog box in the **Identifier** field.
4. Press **Ctrl+Alt**. The declaration is frozen in the lower half of the dialog box.
5. Close the target dialog box.
6. In the **Record Window Declarations** dialog box, click **Paste to Editor**. The information in the **Record Window Declarations** dialog box is cleared, and the newly recorded declarations are appended to the test frame after the last recorded declaration.

7. If you are finished recording declarations, click **Close** on the **Record Window Declarations** dialog box. Otherwise, click **Resume Tracking** to begin recording the declarations for another dialog box.

   Many applications begin with a login window, which is not accounted for when you record the test frame. Therefore, make sure that you invoke this window and record a declaration for it when you are recording the declarations for your application's dialog boxes.

Defining a New Window

Defining a new window can improve how Silk Test Classic records Microsoft Office-based and Internet Explorer applications.

You must have an include file open in order to define a new window.

This functionality is available only for projects or scripts that use the Classic Agent.

1. Click **Record** > **Defined Window**.
2. Click **Draw Rectangle**.
3. Click and drag to form a rectangle around the object you want Silk Test Classic to record.

   Note: The position of the rectangle is recorded in pixels in the **Window Rectangle** field.

4. Once you lift the mouse button, click **Add**.

   The **Update Window Declaration Detail** confirmation message box opens, containing a message similar to the following:
   The window declarations in the following files will be updated c:\program files\<SilkTest installation directory>\silktest\projects\aaa\frame.inc.
5. Click **OK**. The information is saved to the specified file.

The file opens. Scroll to see the new `DefinedWin` with the name you assigned and a tag with the coordinates of the rectangle you drew.

```
[-] DefinedWin StartWindow
[ ] tag "(295,380-997,642)"
```

Specifying Tags

This functionality is available only for projects or scripts that use the Classic Agent.

When you are recording declarations, you can select any combination of tags to record by selecting check boxes in the **Tag Information** group box in the **Record Window Declarations** dialog box. You can record different tags for different objects. You can also specify which tags you want recorded by default.

Default tags

You can record more than one tag for an object. Doing so makes scripts less sensitive to changes when the tests are run. For example, if you record a caption and a window ID for a control, then even if the caption on the control changes (such as the caption "Case sensitive" changing to "Case is significant"), Silk Test Classic can still find the control based on its window ID.

This is particularly an issue in situations where captions change dynamically, such as in MDI applications where the window title changes each time a different child window is made active.

By default, when you record window declarations, each object is given two tags: the caption (if there is one) and the Window ID (if there is one).

Note: Two tags are checked in the **Tag Information** box of the **Record Declarations** dialog box: Caption and Window ID.

For example, here is the default recorded declaration for the **Case sensitive** check box:

```
CheckBox CaseSensitive
   multitag "Case sensitive"
            "$1041"
```

Silk Test Classic specifies multiple tags in a declarations file using the multitag statement. In the previous example, the check box is declared with two tags:

- The string "Case sensitive", which is its caption.
- The string "$1041", which is its Window ID.

Using class-specific multiple tags

You can specify which multiple-tag types to use for an individual class. For example, maybe you don't want window ID used with a particular class, even though you want window ID used with all other classes. You can specify this by including a setting statement in the declaration for the class.

For additional information, see the *winclass Declaration*.

Multiple tags at runtime

When running your test cases, the Agent tries to resolve each part of a multiple tag from top to bottom until it finds an object that matches.

Consider this declaration:

```
CheckBox CaseSensitive
   multitag "Case sensitive"
            "#1"
```

When Silk Test Classic encounters a reference to `Find.CaseSensitive`, it first looks for a check box whose caption is "Case sensitive". If it finds one, it uses it. If it doesn't find one, it looks for the first check box in the dialog box (because of the index tag "#1"). If there is one, Silk Test Classic uses it. If none of the tags resolve, an exception is raised.

For complete information about tag resolution, see *multitag Statement*.

Changing tags

Sometimes you need to change tags from what Silk Test Classic named them by default.

Why change the tags

By default, the GUI object's caption and index are used for the tag, because they are the most portable. In most cases, these are what you want to use.

However, there are situations in which the default tag is not suitable.

Example: Changing a tag

You might want to provide more than one caption for a control if the control's caption can change dynamically. For example, if a push button sometimes says Yes and sometimes says Continue, you could change the tag as shown here:

```
PushButton Confirm
           multitag "Yes"
  "Continue"
```

The Agent would find the pushbutton if it had either caption.

To separate different tag components, use the pipe character: `|`.

Modify a Declaration in the Record Window Declarations Dialog Box

This functionality is supported only if you are using the Classic Agent.

You can modify the identifier or tag for a dialog box as you record it.

1. In the **Window declaration** list box at the bottom of the **Record Window Declarations** dialog box, click the line for the object containing the tag or identifier you want to change. Silk Test Classic updates the **Window Detail** group box in the upper left of the dialog box to include the information from the line you clicked.
2. To change the identifier, replace the existing identifier with one of your choice.
3. To change the tag, select the tag types you want to include in the generated `multitag`.

   You can edit the contents of each tag type in the text fields in the **Tag Information** group box.

   The **Window declaration** list box updates dynamically as you enter the new information.
4. When finished making modifications, click **Paste to Editor**. Silk Test Classic appends the declarations for the dialog box to the test frame file.

Changing the Tags Recorded by Default

This functionality is supported only if you are using the Classic Agent.

1. Click **Record** > **Window Declarations**.
2. Click **Options**.
3. On the **Record Window Declarations Options** dialog box, check and uncheck the check boxes in the **Default multitags** box as appropriate.
4. Click **OK**. The next time you record window declarations, Silk Test Classic will use the tag types you selected by default. You can always override the defaults for a particular object.

Turning Off Multiple Tag Recording

This functionality is supported only if you are using the Classic Agent.

1. Click **Record** > **Window Declarations**.
2. Click **Options**. Silk Test Classic displays the **Record Window Declarations Options** dialog box.
3. Uncheck the **Record multiple tags** check box. The check boxes in the **Default tag** group box become option buttons.
4. Select the tag type you want Silk Test Classic to use by default.
5. Click **OK**.

When you record window declarations, Silk Test Classic defaults to the tag type you selected and record the tag in a `tag` statement. You can always override the default for a particular object.

Use the member-of Operator to Access Data

Use the member-of operator (.) to reference the data defined in a window declaration. For example, if a script needs to know which control should have focus when the **Find** dialog box is first displayed, it can access this data from the window declaration with this expression:

```
Find.lwTabOrder[1]
```

Similarly, to set focus to the third control in the list:

```
Find.lwTabOrder[3].SetFocus ()
```

Identifiers and Tags

This section describes identifiers and tags.

Captions for Objects

This functionality is supported only if you are using the Classic Agent.

By default, Silk Test Classic follows these steps to create a **Caption** tag for an object:

1. Silk Test Classic uses the literal label or caption of the object, if there is one.
2. If the object has a sibling object with the same label or caption, Silk Test Classic appends the object's index number to the tag. The index number is the object's order of appearance in relation to other sibling objects of the same class, from top left to bottom right within the parent object.

   For example, if a dialog box has two objects labeled **Find**, the tag of the one nearest the top left of the dialog box is **Find[1]** and the tag of the one nearest the bottom right of the dialog box is **Find[2]**.
3. If the object does not have a label or caption, Silk Test Classic uses the index number.

   For example, if a dialog box contains two unnamed text boxes, the text box closest to the upper left corner of the dialog box has the tag **#1**, and the other has the tag **#2**.

Overview of Identifiers

When you record test cases, Silk Test Classic uses the window declarations in the test frame file to construct a unique identifier, called a fully qualified identifier, for each GUI object. The fully-qualified identifier consists of the identifier of the object, combined with the identifiers of the object's ancestors. In this way, the 4Test commands that are recorded can manipulate the correct object when you run your test cases.

If all identifiers were unique, this would not be necessary. However, because it is possible to have many GUI objects with the same identifier, for example the **OK** button, a method call must specify as many of the object's ancestors as are required to uniquely identify it.

The following table shows how fully qualified identifiers are constructed:

GUI Object	Fully-Qualified Identifier	Example
Main Window	The main window's identifier	`TextEdit.SetActive ()`
Dialog	The dialog's identifier	`Find.SetActive ()`
Control	The identifiers of the dialog and the control	`Find.Cancel.Click ()`
Menu item	The identifiers of the main window, the menu, and the menu item	`TextEditor.File.Open.Pick ()`

The fully qualified identifier for main windows and dialog boxes does not need to include ancestors because the declarations begin with the keyword window.

An identifier is the GUI object's logical name. By default, Silk Test Classic derives the identifier from the object's actual label or caption, removing any embedded spaces or special characters (such as accelerators). So, for example, the **Save As** label becomes the identifier **SaveAs**. Identifiers can contain single-byte international characters, such as é and ñ.

If the object does not have a label or caption, Silk Test Classic constructs an identifier by combining the class of the object with the object's index. When you are using the Classic Agent, the index is the object's order of appearance, from top left to bottom right, in relation to its sibling objects of the same class. For example, if a text box does not have a label or caption, and it is the first text box within its parent object, the default identifier is TextField1. When you are using the Open Agent, the index depends on the underlying technology of the application under test.

Note: The identifier is arbitrary, and you can change the generated one to the unique name of your choice.

Overview of Tags

This functionality is supported only if you are using the Classic Agent.

The tag is the actual name of the object, as opposed to the identifier, which is the logical name. Silk Test Classic uses the tag to identify objects in the application under test when recording and when executing test cases. Test cases never use the tag to refer to an object; they always use the identifier.

Alternatively, you can use locator keywords, rather than tags, to create scripts that use dynamic object recognition and window declarations. Or, you can include locators and tags in the same window declaration.

There are several types of tags:

Tag Type	Description
`Caption`	The caption or label as it appears to the user.
`Prior text`	Closest static text above or to the left of the object. Prior text tags begin with the ^ character.
`Index`	The order (from top left to bottom right) in relation to its sibling objects of the same class. Index tags must begin with the # character.
`Window ID`	The GUI-specific internal ID of the object. Window ID tags begin with the $ character.
`Location`	The physical location (coordinates) of the object. Location tags begin with the @ character.
`Attributes`	The attribute name(s) of the Html object. If the object is not an Html object, nothing is recorded.

Not all types of objects have all tags. Dialog boxes, for example, do not have window IDs, so they cannot have a `Window ID` tag.

In the **Record Window Declarations** dialog box, if you record declarations for the **Case sensitive** check box in the **Text Editor's Find** dialog box, the possible tags for the check box include:

Tag Type	Value	Comments
`Caption`	Case sensitive	
`Prior text`	^Find What:	"Find What" is the nearest static text above or to the left of the check box
`Index`	#1	The Case Sensitive check box is the first check box in the dialog
`Window ID`	$1041	
`Location`	@(57,75)	
`Attributes`	[blank]	Attributes are only recorded for Html objects.

These are the possible tags that can be used by Silk Test Classic to identify the **Case sensitive** check box when recording or executing test cases.

It is helpful to understand how Silk Test Classic identifies tags in browsers. For additional information, see *Comparison of DOM and VO*.

Save the Test Frame

To save a test frame, click **File** > **Save** when the test frame is the active window. If it is a new file, it is automatically named `frame.inc`. If you already have a `frame.inc` file, a number is appended to the file name. You can click **File** > **Save** to select another name.

If you are working within a project, Silk Test Classic automatically adds the new test frame (`.inc`) to the project.

When saving a file, Silk Test Classic does the following:

- Saves a source file, giving it the `.inc` extension. The source file is an ASCII text file, which you can edit. For example: `myframe.inc`.
- Saves an object file, giving it the `.ino` extension. The object file is a binary file that is executable, but not readable by you. For example: `myframe.ino`.

Specifying How a Dialog Box is Invoked

4Test provides two equivalent ways to invoke a dialog box:

- Use the `Pick` method to pick the menu item that invokes the dialog box. For example: `TextEditor.File.Open.Pick ()`
- Use the `Invoke` method: `Open.Invoke ()`

While both are equivalent, using the `Invoke` method makes your test cases more maintainable. For example, if the menu pick changes, you only have to change it in your window declarations, not in any of your test cases.

The Invoke method

To use the `Invoke` method, you should specify the *wInvoke* variable of the dialog box. The variable contains the identifier of the menu item or button that invokes the dialog box. For example:

```
window DialogBox Open
   tag "Open"
   parent TextEditor
WINDOW wInvoke = TextEditor.File.Open
```

Class Attributes

This section describes class attributes.

Attributes Tag Notation

This functionality is supported only if you are using the Classic Agent.

The attributes tag does not use the [n] notation to distinguish between windows with the same value of the tag. If the attributes tag is comprised only of an attribute that is not unique, then the same tag is recorded for multiple objects. The duplication is not detected until Silk Test Classic tries to match the tag at runtime, at which time an `E_WINDOW_NOT_UNIQUE` exception will be raised.

If you anticipate that multiple windows may have the same value for the attributes tag, then use `multitags` instead (the attributes tag will have precedence).

Enabling Class Attribute Recording

This functionality is supported only if you are using the Classic Agent.

You must turn on attribute recording in order for Silk Test Classic to capture Html class attributes.

1. Click **Record** > **Window Declarations**, then click **Options**.
2. On the **Record Window Declarations Options** dialog box, check the **Attributes** check box in the Default multi-tag area.

   As with other attributes, if you want to record only the Attributes tags, uncheck the **Record multiple tags** check box. As with previous versions of Silk Test Classic, if you want to record multiple tags, leave the **Record multiple tags** check box checked.

3. Click **OK** to save your selection.

Recording Existing Html Class Attributes and Specifying the Hierarchy of Attributes

This functionality is supported only if you are using the Classic Agent.

You must enable attribute recording to allow Silk Test Classic to capture Html class attribute information while recording.

1. Click **Record** > **Window Declarations**, then click **Edit Class Attributes**.
2. On the **Edit Class Attributes** dialog box, select **Browser DOM** from the **Set** list box.
3. Select a class from the **Class** list box.

   For example, select **HtmlCheckBox** to specify attributes within the `HtmlCheckBox` class that you want to record.
4. After you have selected a class, select the attribute you want to record from the list of attributes in the **Defined attributes** list box.
5. Click **>>** to move the attribute to the **Class attributes** pane.

   You can only select a single attribute at a time.
6. Select an attribute, then click **Move Up** or **Move Down** to indicate the order in which you want the attributes to appear when you paste the window declaration to the Editor.

   You may select only a single attribute at a time.
7. Click **OK** to save your work and return to the **Record Window Declarations** dialog box. Silk Test Classic records the attribute tags in the order you have specified.

You can record custom class attributes in a test that uses hierarchical object recognition and Silk Test Classic will record the attribute tags in the order you have specified.

For example, record four attributes for the `HtmlImage` class. First you select `id` from the **Defined Attributes** list box, then you click **>>** to move it to the **Class Attributes** list box. You repeat that process for the `name`, `rel`, and `src` attributes. Once the four attributes are listed in the **Class Attributes** list box, you use **Move Up** and **Move Down** so that the attributes display in the order you want them to display.

After you click **OK**, Silk Test Classic displays the **Record Window Declarations** dialog box. Whenever Silk Test Classic recognizes an `HtmlImage` object, Silk Test Classic records the attributes you specified on the **Edit Class Attributes** dialog box. The full string for the attributes tag information that Silk Test Classic records is:

```
&id='myButton';name='button';src='file:???D:?buttonnext.gif';rel='start'
```

Adding a New Class Attribute and Specifying the Hierarchy of Attributes

This functionality is supported only if you are using the Classic Agent.

You can add a new attribute to the list that Silk Test Classic records for an Html class and specify the order in which the attributes are recorded. Add custom attributes to a Web application to make a test more specific.

To capture class attribute information, you must first enable attribute recording.

1. Click **Record** > **Window Declarations**, then click **Edit Class Attributes**.
2. On the **Edit Class Attributes** dialog box, select **Browser DOM** from the **Set** list box.
3. Select a class from the **Class** list box.

   For example, select `HtmlCheckBox` to specify attributes within the `HtmlCheckBox` class that you want to record.

4. Type the name of the new attribute in the text box above **Add** and **Remove**, then click **Add**.

   There is a 62 character limit to attribute names. You may type in the name or you can copy and paste the name from the text editor. If you are typing in a long name, the field stops accepting characters after the 62nd character. If you paste in a long name, the name is truncated at 62 characters. The following characters are not allowed in attribute names: `(space) ~`!@#$%^&*()_-+={[}]|\:;"'<,>.?/`
5. Click **>>** to move the new attribute to the **Class Attributes** pane.
6. Select an attribute, then click **Move Up** or **Move Down** to indicate the order in which you want the attributes to display when you paste the window declaration to the Editor.

   You may select only a single attribute at a time.
7. Click **OK** to save your work and return to the **Record Window Declarations** dialog box. Silk Test Classic will now record the new attribute tags in the order you specified. To record a test that uses the custom Html class attribute, you must use the Classic Agent.

Deleting a Class Attribute

This functionality is supported only if you are using the Classic Agent.

You can delete an Html class attribute or a Java SWT custom class attribute if you want Silk Test Classic to avoid recording it.

1. Click **Record** > **Window Declarations**, then click **Edit Class Attributes**.
2. On the **Edit Class Attributes** dialog box, select the name of the attribute you want to delete from the **Defined attributes** list box.
3. Click **Remove**. The name is removed from the list of attributes. It is possible to delete an attribute from the **Defined attributes** list box and still have it display in the **Class attributes** list box. An attribute remains in the **Class attributes** list box until you move it to the **Defined attributes** list box and then delete it, if you like.
4. Click **OK** to save your work and return to the **Record Window Declarations** dialog box, or click **Cancel** to avoid deleting the attribute.

Improving Object Recognition with Microsoft Accessibility

You can use Microsoft Accessibility (Accessibility) to ease the recognition of objects at the class level. There are several objects in Internet Explorer and in Microsoft applications that Silk Test Classic can better recognize if you enable Accessibility. For example, without enabling Accessibility Silk Test Classic records only basic information about the menu bar in Microsoft Word and the tabs that appear. However, with Accessibility enabled, Silk Test Classic fully recognizes those objects.

Example

Without using Accessibility, Silk Test Classic cannot fully recognize a `DirectUIHwnd` control, because there is no public information about this control. Internet Explorer uses two `DirectUIHwnd` controls, one of which is a popup at the bottom of the browser window. This popup usually shows the following:

- The dialog box asking if you want to make Internet Explorer your default browser.
- The download options **Open**, **Save**, and **Cancel**.

When you start a project in Silk Test Classic and record locators against the `DirectUIHwnd` popup, with accessibility disabled, you will see only a single control. If you enable Accessibility you will get full recognition of the `DirectUIHwnd` control.

Enabling Accessibility

This functionality is supported only if you are using the Classic Agent.

If you are testing an application and Silk Test Classic cannot recognize objects, you should first enable Accessibility. Accessibility is designed to help Silk Test Classic recognize objects at the class level. If that does not help with recognition, then you should try defining a new window by clicking **Record** > **Defined Window**.

Accessibility is turned off by default, and you need to enable your extension as usual. There are two ways to enable accessibility:

- If you are using the **Basic Workflow**, Silk Test Classic is usually able to do this automatically when you check the **Enable Accessibility** check box on the **Extension Settings** dialog box.
- If you are configuring your extension manually, you can enable Accessibility by checking the **Accessibility** check box on the **Extension Enabler** and the **Extensions Option** dialog box.

Adding Accessibility Classes

This functionality is supported only if you are using the Classic Agent.

Use accessibility to help Silk Test Classic better identify unrecognizable objects. The information you add to the list of classes is stored in the `accex.inc` file, which is installed by default in the `<Silk Test installation directory>\Extend` or `<Project name>\extend` directory.

You cannot add duplicate or blank class names.

1. Open the application containing the unrecognizable objects.
2. Open Silk Test Classic, then click **Record** > **Class** > **Accessibility**.
3. On the **Windows Accessibility** dialog box, click and drag the **Finder tool** icon over the object you want to identify. A black rectangle displays around the edge of the control. When you release the mouse button, the object's information displays in the **Name** and **Class** text boxes.
4. Click **Add** to move the class name to the list of Accessibility classes that Silk Test Classic can identify, then click **OK**.

Improving Object Recognition with Accessibility

This functionality is supported only if you are using the Classic Agent.

There are several objects in Internet Explorer and Microsoft applications that Silk Test Classic can better recognize if you enable Accessibility. For example, without enabling Accessibility Silk Test Classic records only basic information about the menu bar in Microsoft Word and the tabs that appear in Internet Explorer 7.0. However, with Accessibility enabled, Silk Test Classic fully recognizes those objects.

Accessibility is not available for Java applications or applets.

Comparison of what Silk Test Classic records

Without Accessibility enabled on Microsoft Excel, Silk Test Classic records the following if the mouse points to the File command on the main toolbar:

```
[+] CustomWin MenuBar
[+] multitag "[MsoCommandBar]Menu Bar"
[ ] "[MsoCommandBar]$0[1]"
[+] CustomWin TypeAQuestionForHelp
[+] multitag "[RichEdit20W]Type a question for help"
[ ] "[RichEdit20W]$16075636"
[+] CustomWin Standard
[+] multitag "[MsoCommandBar]Standard"
```

```
[ ] "[MsoCommandBar]$0[2]"
...
```

However, if you record the same test case with Accessibility enabled, Silk Test Classic is able to record the following:

```
[+] AccObject WorksheetMenuBar
    [+] multitag "Worksheet Menu Bar"
        [ ] "$window"
[+] AccObject WorksheetMenuBar2
    [+] multitag "Worksheet Menu Bar[2]"
     [ ] "$menu bar[2]"
    [+] AccMenuItem File
        [+] multitag "File"
            [ ] "$menu item[1]"
    [+] AccMenuItem Edit
        [+] multitag "Edit"
         [ ] "$menu item[2]"
    [+] AccMenuItem View
        [+] multitag "View"
            [ ] "$menu item[3]"
    [+] AccMenuItem Insert
        [+] multitag "Insert"
            [ ] "$menu item[4]"
    [+] AccMenuItem Format
        [+] multitag "Format"
            [ ] "$menu item[5]"
…
```

With Accessibility enabled Silk Test Classic is able to record more than simple details about the File menu command.

Silk Test Classic stores the information about Accessibility classes in the `accex.inc` file which is installed by default in the `<Silk Test installation directory>/Extend` or `<Project name>\extend` directory. The `accex.inc` file comes pre-loaded with several classes, including the `MsoCommandBar`, the class of the Microsoft Office menu bar.

Removing Accessibility Classes

This functionality is supported only if you are using the Classic Agent.

You use Accessibility to help better identify unrecognizable objects. However, you may want to delete classes from the list of classes you created in order to clean up your `.inc` file or to prevent recognition of classes.

1. Click **Record** > **Class** > **Accessibility**.
2. On the **Windows Accessibility** dialog box, select the name of the class you want to remove from the list of Accessibility classes.
3. Click **Remove** to delete the class name, then click **OK**.

The information is removed from the list of classes in the `accex.inc` file. The `accex.inc` file is installed by default to the `<SilkTest installation directory>\Extend` or `<name of project>\extend` directory.

Calling Windows DLLs from 4Test

This section describes how you can call Windows DLLs from 4Test.

Note: The Open Agent supports DLL calling for both 32-bit and 64-bit DLL calls, while the Classic Agent supports DLL calling only for 32-bit calls.

Aliasing a DLL Name

If a DLL function has the same name as a 4Test reserved word, or the function does not have a name but an ordinal number, you need to rename the function within your 4Test declaration and use the 4Test alias statement to map the declared name to the actual name.

For example, the exit statement is reserved by the 4Test compiler. Therefore, to call a function named `exit`, you need to declare it with another name, and add an alias statement, as shown here:

```
dll "mydll.dll"
my_exit ()
alias "exit"
```

Calling a DLL from within a 4Test Script

A declaration for a DLL begins with the keyword dll. The general format is:

```
dll dllname.dll
prototype
[prototype]...
```

where `dllname` is the name of the dll file that contains the functions you want to call from your 4Test scripts and `prototype` is a function prototype of a DLL function you want to call.

Environment variables in the DLL path are automatically resolved. You do not have to use double backslashes (\\) in the code, single backslashes (\) are sufficient.

The Open Agent supports calling both 32bit and 64bit DLLs. You can specify which type of DLL the Open Agent should call by using the `SetDllCallPrecedence` method of the `AgentClass` class. If you do not know if the DLL is a 32bit DLL or a 64bit DLL, use the `GetDllCallPrecedence` function of the `AgentClass` Class. The Classic Agent provides support for calling 32bit DLLs only.

Prototype syntax

A function prototype has the following form:

```
return-type func-name ( [arg-list] )
```

where:

`return-type` The data type of the return value, if there is one.

`func-name` An identifier that specifies the name of the function.

`arg-list` A list of the arguments passed to the function, specified as follows:

```
[pass-mode] data-type identifier
```

where:

`pass-mode` Specifies whether the argument is passed into the function (`in`), passed out of the function (`out`), or both (`inout`). If omitted, `in` is the default.

To pass by value, make a function parameter an `in` parameter.

To pass by reference, use an `out` parameter if you only want to set the parameter's value; use an `inout` parameter if you want to get the parameter's value and have the function change the value and pass the new value out.

`data-type` The data type of the argument.

identifier The name of the argument.

You can call DLL functions from 4Test scripts, but you cannot call member functions in a DLL.

Example

The following example writes the text *hello world!* into a field by calling the `SendMessage` DLL function from the DLL `user32.dll`.

```
use "mswtype.inc"
use "mswmsg32.inc"

dll "user32.dll"
  inprocess ansicall INT SendMessage (HWND hWndParent, UINT
msg, WPARAM  wParam, LPARAM lParam) alias "SendMessageA"

testcase SetTextViaDllCall()
  SendMessage(UntitledNotepad.TextField.GetHandle(),
WM_SETTEXT, 0, "hello world! ")
```

Passing Arguments to DLL Functions

Valid data types for arguments passed to DLL functions

Since DLL functions are written in C, the arguments you pass to these functions must have the appropriate C data types. In addition to the standard 4Test data types, Silk Test Classic also supports the following C data types:

- char, int, short, and long
- unsigned char, unsigned int, unsigned short, and unsigned long
- float and double

Note: Any argument you pass must have one of these data types (or be a record that contains fields of these types).

Passing string arguments

The `char*` data type in C is represented by the 4Test `STRING` data type. The default string size is 256 bytes.

The following code fragments show how a char array declared in a C struct is declared as a `STRING` variable in a 4Test record:

```
// C declaration
typedef struct
{
...
char szName[32];
...
}

// 4Test declaration
type REC is record
...
STRING sName, size=32
...
```

To pass a NULL pointer to a STRING, use the NULL keyword in 4Test. If a DLL sets an out parameter of type char* to a value larger than 256 bytes, you need to initialize it in your 4Test script before you pass it to the DLL function. This will guarantee that the DLL does not corrupt memory when it writes to the

parameter. For example, to initialize an out parameter named `my_parameter`, include the following line of 4Test code before you pass my_parameter to a DLL:

```
my_parameter = space(1000)
```

If the user calls a DLL function with an output string buffer that is less then the minimum size of 256 characters, the original string buffer is resized to 256 characters and a warning is printed. This warning, `String buffer size was increased from x to 256 characters` (where x is the length of the given string plus one) alerts the user to a potential problem where the buffer used might be shorter than necessary.

Passing arguments to functions that expect pointers

When passing pointers to C functions, use these conventions:

- Pass a 4Test string variable to a DLL that requires a pointer to a character (null terminated).
- Pass a 4Test array or list of the appropriate type to a DLL that requires a pointer to a numerical array.
- Pass a 4Test record to a DLL that requires a pointer to a record. 4Test records are always passed by reference to a DLL.
- You cannot pass a pointer to a function to a DLL function.

Passing arguments that can be modified by the DLL function

An argument whose value will be modified by a DLL function needs to be declared using the `out` keyword. If an argument is sometimes modified and sometimes not modified, then declare the argument as `in` and then, in the actual call to the DLL, preface the argument with the `out` keyword, enclosed in brackets.

For example, the third argument (`lParam`) to the `SendMessage` DLL function can be either `in` or `out`. Therefore, it is declared as follows:

```
// the lParam argument is by default an in argument
dll "user.dll"
LRESULT
SendMessage (HWND hWnd, UINT uiMsg, WPARAM wParam, LPARAM lParam)
```

Then, to call the DLL with an out argument, you use the keyword `out`, enclosed within brackets:

```
SendMessage (Open.hWnd, WM_GETTEXT, 256, [out] sText)
```

Passing window handles to a DLL function

If a parameter takes a window handle, use the hwnd property or the `GetHandle` method of the `AnyWin` class to get the window handle you need.

Using DLL Support Files Installed with Silk Test Classic

Silk Test Classic is installed with the following include files that contain all the declarations, data types, and constants necessary for you to call hundreds of functions within the Windows API from your scripts.

`msw32.inc`

Contains use statements for the include files that apply to 32-bit Windows: `mswconst.inc`, `mswtype.inc`, `mswfun32.inc`, `mswmsg32.inc`, and `mswutil.inc`.

By including `msw32.inc` in your 4Test scripts, you have access to all the information in the other include files.

Note: The DLL functions declared in the files included in `msw32.inc` are aliased to the W (wide-character) functions.

mswconst.inc Declares constants you pass to DLL functions. These constants contain style bits, message box flags, codes used by the `GetSystemMetrics` function, flags used by the `GetWindow` function, window field offsets for the `GetWindowLong` and the `GetWindowWord` functions, class field offsets for the `GetClassLong` and `GetClassWord` functions, and menu function flags.

mswfun32.inc Contains 4Test declarations for 32-bit functions in the `user32.dll` and `kernel32.dll` files. The `mswfun32.inc` file provides wide character support. This means that you no longer have to edit `mswfun32.inc` in order to call Windows DLL functions. See the description of `mswfun32.inc` in the Dll declaration section.

mswmsg32.inc Declares 32-bit Microsoft Window messages, control messages, and notification codes.

mswtype.inc Declares many data types commonly used in the Windows API.

mswutil.inc Contains the following utility functions:

- `PrintWindowDetail`
- `GetStyleBitList`
- `PrintStyleBits`

Extending the Class Hierarchy

This section describes how you can extend the class hierarchy.

Classes

This section describes the 4Test classes.

Overview of Classes

The `class` indicates the type, or kind, of GUI object being declared.

Note: This is the 4Test class, not the class that the GUI itself uses internally. For example, although the class might be Label on one GUI and Text on another, 4Test uses the class name `StaticText` to refer to text strings that cannot be edited.

A class defines data and behavior

The class also defines methods (actions) and properties (data) that are inherited by the GUI object. For example, if you record a declaration for a pushbutton named OK, a test case can legally use a method like `Click` on the pushbutton because the `Click` method is defined at the class level. In other words, the definition of what it means to click on a pushbutton is included within the definition of the 4Test class itself, and this definition is inherited by each pushbutton in the GUI. If this were not true, you would have to define within each GUI object's window declaration all the methods you wanted to use on that object.

The class as recorded cannot be changed

The one exception is that if the recorded class is `CustomWin`, meaning that Silk Test Classic does not recognize the object. You can, when appropriate, map the class to one that is recognized.

Custom classes

Enable an application to perform functions specific to the application and to enhance standard class functionality. Custom classes are also easy to maintain and can be extended easily by developers. All custom objects default to the built-in class, `CustomWin`.

Custom objects fall into two general categories:

Visible objects	Objects that Silk Test Classic knows about, but cannot identify, for example, the icon in an About dialog box. Two further categories of visible objects include: • Common objects are those that look and behave like standard objects, for example, a third-party object that looks and acts like a `PushButton`, but is recorded as a `CustomWin`. • Uncommon objects, on the other hand, have no relation to the existing standard objects. For example, an Icon. there is no corresponding `Icon` class.
Invisible objects	Objects that Silk Test Classic cannot recognize at all.

Polymorphism

If a class defines its own version of a method or property, that method or property overrides the one inherited from an ancestor. This is referred to as polymorphism. For example, the `ListBox` class has its own `GetContents` method, which overrides the `GetContents` method inherited from the `AnyWin` class.

CursorClass, ClipboardClass, and AgentClass

The following three classes are not part of the `AnyWin` class hierarchy, because they define methods for objects that are not windows:

`CursorClass`	Defines the three methods you can use on the cursor: `GetPosition`, `GetType`, and `Wait`.
`ClipboardClass`	Defines the two methods you can use on the system clipboard: `GetText` and `SetText`.
`AgentClass`	Defines the methods you can use to set options in the 4Test Agent. The 4Test Agent is the component of Silk Test Classic that translates the method calls in your test cases into the appropriate GUI- specific event streams.

Predefined identifiers for Cursor, Clipboard, and Agent

You do not record declarations for the cursor, the clipboard, or the Agent. Instead, you use predefined identifiers for each of these objects when you want to use a method to act against the object. The predefined methods for each are:

- 4Test Agent: `Agent`
- clipboard: `Clipboard`
- cursor (mouse pointer): `Cursor`

For example, to set a 4Test Agent option, you use a call such as the following:

```
Agent.SetOption (OPT_VERIFY_COORD, TRUE)
```

Defining New Classes with the Classic Agent

This functionality is supported only if you are using the Classic Agent.

Consider the declarations for the **Open** and the **Save As** dialog boxes of the **Text Editor** application, which each contain exactly the same child windows:

window DialogBox Open

```
   tag "Open"
   parent TextEditor
   StaticText FileNameText
      tag "File Name:"
   TextField FileName1
```

```
      tag "File Name:"
   ListBox FileName2
      tag "File Name:"
   StaticText DirectoriesText
      tag "Directories:"
   StaticText PathText
      tag "#3"
   ListBox Path
      tag "#2"
   StaticText ListFilesOfTypeText
      tag "List Files of Type:"
   PopupList ListFilesOfType
      tag "List Files of Type:"
   StaticText DrivesText
      tag "Drives:"
   PopupList Drives
      tag "Drives:"
   PushButton OK
      tag "OK"
   PushButton Cancel
      tag "Cancel"
   PushButton Network
      tag "Network"
```

window DialogBox SaveAs

```
   tag "Save As"
   parent TextEditor
   StaticText FileNameText
      tag "File Name:"
   TextField FileName1
      tag "File Name:"
   ListBox FileName2
      tag "File Name:"
   StaticText DirectoriesText
      tag "Directories:"
   StaticText PathText
      tag "#3"
   ListBox Path
      tag "#2"
   StaticText ListFilesOfTypeText
      tag "List Files of Type:"
   PopupList ListFilesOfType
      tag "List Files of Type:"
   StaticText DrivesText
      tag "Drives:"
   PopupList Drives
      tag "Drives:"
   PushButton OK
      tag "OK"
   PushButton Cancel
      tag "Cancel"
   PushButton Network
      tag "Network"
```

It is not uncommon for an application to have multiple dialogs whose only difference is the caption: The child windows are all identical or nearly identical. Rather than recording declarations that repeat the same child objects, it is cleaner to create a new class that groups the common child objects.

For example, here is the class declaration for a new class called `FileDialog`, which is derived from the `DialogBox` class and declares each of the children that will be inherited by the **SaveAs** and **Open** dialog boxes:

```
winclass FileDialog : DialogBox
   parent TextEditor
```

```
   StaticText FileNameText
      tag "File Name:"
   TextField FileName1
      tag "File Name:"
   ListBox FileName2
      tag "File Name:"
   StaticText DirectoriesText
      tag "Directories:"
   StaticText PathText
      tag "#3"
   ListBox Path
      tag "#2"
   StaticText ListFilesOfTypeText
      tag "List Files of Type:"
   PopupList ListFilesOfType
      tag "List Files of Type:"
   StaticText DrivesText
      tag "Drives:"
   PopupList Drives
      tag "Drives:"
   PushButton OK
      tag "OK"
   PushButton Cancel
      tag "Cancel"
   PushButton Network
      tag "Network"
```

To make use of this new class, you must do the following:

1. Rewrite the declarations for the **Open** and **Save As** dialog boxes, changing the class to **FileDialog**.
2. Remove the declarations for the child objects inherited from the new class.

Here are the rewritten declarations for the **Open** and **Save As** dialog boxes:

```
window FileDialog SaveAs
   tag "Save As"
window FileDialog Open
   tag "Open"
```

For more information on the syntax used in declaring new classes, see the `winclass` declaration.

The default behavior of Silk Test Classic is to tag all instances of the parent class as the new class. So, if you record a window declaration against a standard object from which you have defined a new class, Silk Test Classic records that standard object's class as the new class. To have all instances declared by default as the original class, add the following statement to the declaration of your new class: `setting DontInheritClassTag = TRUE`. For example, let's say you define a new class called `FileDialog` and derive it from the `DialogBox` class. Then you record a window declaration against a dialog box. Silk Test Classic records the dialog box to be of the new `FileDialog` class, instead of the `DialogBox` class. To have Silk Test Classic declare the class of the dialog box as `DialogBox`, in the `FileDialog` definition, set DontInheritClassTag to TRUE. For example:

```
winclass FileDialog : DialogBox
  setting DontInheritClassTag = TRUE
```

Defining New Class Properties

You can define new properties for existing classes using the `property` declaration. You use these class properties to hold data about an object; you can use class properties anywhere in a script.

DesktopWin

Because the desktop is a GUI object, it derives from the `AnyWin` class. However, unlike other GUI objects, you do not have to record a declaration for the desktop. Instead, you use the predefined identifier `Desktop` when you want to use a method on the desktop.

For example, to call the `GetActive` method on the desktop, you use a call like the following:

```
wActive = Desktop.GetActive ()
```

Logical Classes

The `AnyWin`, `Control`, and `MoveableWin` classes are logical (virtual) classes that do not correspond to any actual GUI objects, but instead define methods common to the classes that derive from them. This means that Silk Test Classic never records a declaration that has one of these classes.

Furthermore, you cannot extend or override logical classes. If you try to extend a logical class, by adding a method, property or data member to it, that method, property, or data member is not inherited by classes derived from the class. You will get a compilation error saying that the method/property/data member is not defined for the window that tries to call it. Nor can you override the class, by rewriting existing methods, properties, or data members. Your modifications are not inherited by classes derived from the class.

Class Hierarchy (Classic Agent)

You can define your own methods and properties, as well as define your own classes. You can also define your own attributes, which are used in the verification stage in test cases.

The 4Test class hierarchy defines the methods and properties that enable you to query, manipulate, and verify the data or state of any GUI object in your application. You can define your own methods and properties, as well as define your own classes. You can also define your own attributes, which are used in the verification stage in test cases. The following schema shows a listing of the built-in class hierarchy for the core classes and the Classic Agent:

- `AgentClass`
- `AnyWin`
  - `Control`
    - `CheckBox`
    - `ComboBox`
    - `DynamicText`
    - `Header`
    - `ListBox`
    - `ListView`
    - `PageList`
    - `PopupList`
    - `PushButton`
    - `RadioList`
    - `Scale`
    - `ScrollBar`
      - `HorizontalScrollBar`
      - `VerticalScrollBar`
    - `StaticText`
    - `StatusBar`
    - `Table`
    - `TextField`
    - `ToolBar`
    - `TreeView`
      - `TreeViewEx`
    - `UpDown`
  - `ControlMultiWin`
  - `CustomWin`

    - DefinedWin
    - DesktopWin
    - Menu
        - MenuItem
        - SysMenu
        - PopupMenu
        - PopupStart
    - MoveableWin
        - ChildWin
        - DialogBox
            - MessageBoxClass
        - MainWin
    - TaskbarWin
    - WinPart
- ClipboardClass
- CursorClass

Verifying Attributes and Properties

This section describes how you can use attributes and properties to verify test cases.

Attribute Definition and Verification

When you record a test case, you can verify the test case using attributes.

You can choose to verify using either attributes or properties. Generally you will verify using properties, because property verification is more flexible.

For example, the attributes for the `DialogBox` class are `Caption`, `Contents`, `Default button`, `Enabled`, and `Focus`. The following 4Test code implements the `Default Button` attribute in the `winclass.inc` file:

```
attribute "Default button", VerifyDefaultButton, QueryDefaultButton
```

As this 4Test code shows, each attribute definition begins with the statement, followed by the following three comma-delimited values:

1. The text that you want to display in the Attribute panel of the **Verify Window** dialog box. This text must be a string.
2. The method Silk Test Classic should use to verify the value of the attribute at runtime.
3. The method Silk Test Classic should use to get the actual value of the attribute at runtime.

Defining a New Attribute for an Existing Class

To add one or more attributes to an existing class, use the following syntax:

```
winclass ExistingClass : ExistingClass...
attribute_definitions
```

Each attribute definition begins with the `attribute` statement, followed by the following three comma-delimited values:

1. The text that you want to display in the **Attribute** panel of the **Verify Window** dialog box. This text must be a string.
2. The method Silk Test Classic should use to verify the value of the attribute at runtime.
3. The method Silk Test Classic should use to get the actual value of the attribute at runtime.

Each attribute definition must begin and end on its own line. When you define a new attribute, you usually need to define two new methods (steps 2 and 3 above) if none of the built-in methods suffice.

Silk Test Classic allows you to add, delete, or edit the existing functionality of a class; this applies to both functions and variables of a class. However, we recommend that you do not override a function or a variable by declaring a function or variable of that same name. Furthermore, you should never override a variable that has a tag associated with it. You cannot have two variables with the same name in the same level of an object. If you do so, Silk Test Classic will display a compile error.

Defining New Verification Properties

You can perform verifications in your test cases using properties. These verification properties are different from class properties, which are defined using the property declaration. Verification properties are used only when verifying the state of your application in a test case. Silk Test Classic comes with built-in verification properties for all classes of GUI objects.

You can define your own verification properties, which will be added to the built-in properties listed in the **Verify Window** dialog box when you record a test case.

Syntax for Attributes

To add one or more attributes to an existing class, use the following syntax:

```
winclass ExistingClass : ExistingClass...
attribute_definitions
```

Each attribute definition must begin and end on its own line.

When you define a new attribute, you usually need to define two new methods if none of the built-in methods suffices.

For example, to add a new attribute to the `DialogBox` class that verifies the number of children in the dialog box, you add code like this to your test frame (or other include file):

```
winclass DialogBox:DialogBox

  attribute "Number of children", VerifyNumChild, GetNumChild

  integer GetNumChild()
     return ListCount (GetChildren ()) // return count of children of dialog

  hidecalls VerifyNumChild (integer iExpectedNum)
     Verify (GetNumChild (), iExpectedNum, "Child number test")
```

As this example shows, you use the hidecalls keyword when defining the verification method for the new attribute.

Hidecalls Keyword

The keyword hidecalls hides the method from the call stack listed in the results. Using hidecalls allows you to update the expected value of the verification method from the results. If you do not use hidecalls in a verification method, the results file will point to the frame file, where the method is defined, instead of to the script. We recommend that you use hidecalls in all verification methods so that you can update the expected values.

An Alternative to NumChildren as a Class Property

Instead of defining `NumChildren` as a class property, you could also define it as a variable, then initialize the variable in a script. For example, in your include file, you would have:

```
winclass DialogBox : DialogBox
INTEGER NumChild2
```

```
 // list of custom verification properties
LIST OF STRING lsPropertyNames = {"NumChild2"}
```

And in your script, before you do the verification, you would initialize the value for the dialog box under test, such as:

```
Find.NumChild2 = ListCount(Find.GetChildren ())
```

Defining Methods and Custom Properties

This section describes how you can define methods and custom verification properties.

Defining a New Method

To add a method to an existing class, you use the following syntax to begin the method definition:

```
winclass ExistingClass : ExistingClass
```

The syntax `ExistingClass : ExistingClass` means that the declaration that follows extends the existing class definition, instead of replacing it.

Note: Adding a method to an existing class adds the method to all instances of the class.

Example

To add a `SelectAll`() method to the `TextField` class, add the following code to your `frame.inc` file:

```
winclass TextField : TextField
 SelectAll()
   TypeKeys("<Ctrl+a>")
```

In your test cases, you can then use the `SelectAll` method like any other method in the `TextField` class.

```
UntitledNotepad.TextField.SelectAll()
```

Defining a New Method for a Single GUI Object

To define a new method to use on a single GUI object, not for an entire class of objects, you add the method definition to the window declaration for the individual object, not to the class. The syntax is exactly the same as when you define a method for a class.

To add a method to a single GUI object, for example to add the `SelectAll`() method to a specific `TextField` object, locate the GUI object in your `frame.inc` file, like described in the following code sample:

```
window MainWin UntitledNotepad

  ...

  TextField TextField
    locator "//TextField"
```

In your test cases, you can then use the `SelectAll` method like any other method of the `TextField` object:

```
window MainWin UntitledNotepad

  ...

  TextField TextField
    locator "//TextField"
```

```
    SelectAll()
      TypeKeys("<Ctrl+a>")
```

Note: Adding a method to a single GUI object adds the method only to the specific GUI object and not to other instances of the class.

Classic Agent Example

For example, suppose you want to create a method named `SetLineNum` for a dialog box named **GotoLine**, which performs the following actions:

- Invokes the dialog box.
- Enters a line number.
- Clicks **OK**.

The following 4Test code shows how to add the definition for the `SetLineNum` method to the declaration of the **GotoLine** dialog box.

```
window DialogBox GotoLine
   tag "Goto Line"
   parent TextEditor
   const wInvoke = TextEditor.Search.GotoLine

   void SetLineNum (STRING sLine)
     Invoke ()           // open dialog
     Line.SetText (sLine) // populate text field
                         // whose identifier is Line
     Accept ()           // close dialog, accept values

     //Then, to go to line 7 in the dialog, you use this method
call in your testcases:
     GotoLine.SetLineNum (7)
```

Recording a Method for a GUI Object

If you need to perform an action on an object, and the class does not provide a method for doing so, you can define your own method in the window declaration for the object. Then, in your scripts, you can use the method as though it were just another of the built-in methods of the class. You can hand-code methods or record them.

Before you can record a method, you must have already recorded window declarations.

1. Position the insertion point on the declaration of the GUI object to which you want to add a method.
2. Click **Record** > **Method**.
3. On the **Record Method** dialog box, name the method by typing the name or selecting one of the predefined methods: `BaseState`, `Close`, `Invoke`, or `Dismiss`.
4. Click **Start Recording**. Silk Test Classic is minimized and your application and the **SilkTest Record Status** dialog box open.
5. Perform and record the actions that you require.
6. On the **SilkTest Record Status** dialog box, click **Done**. The **Record Method** dialog box opens with the actions you recorded translated into 4Test statements.
7. On the **Record Method** dialog box, click **OK** to paste the code into your include file.
8. Edit the 4Test statements that were recorded, if necessary.

Note: To add a method to a class which is using the Open Agent, you can also manually code the new method into the script or copy the method into the script from a recorded test case.

Deriving a New Method from an Existing One

To derive a new method from an existing method, you can use the derived keyword followed by the scope resolution operator (::).

Use the following syntax:

```
new method : existing method
```

The following example defines a `GetCaption` method for `WPFNewTextBox` that prints the string `Caption as is` before calling the built-in `GetCaption` method (defined in the `AnyWin` class) and printing its return value:

```
winclass WPFNewTextBox : WPFTextBox
GetCaption ()
Print ("Caption as is: ")
Print (derived::GetCaption ())
```

Defining Custom Verification Properties

1. In a class declaration or in the declaration for an individual object, define the variable `lsPropertyNames` as follows:

   ```
   LIST OF STRING lsPropertyNames
   ```

2. Specify each of your custom verification properties as elements of the list `lsPropertyNames`. Custom verification properties can be either:
   - Class properties, defined using the property statement.
   - Variables of the class or individual object.

Any properties you define in `lsPropertyNames` will override built-in properties with the same name. With your custom verification properties listed as elements in `lsPropertyNames`, when you record and run a test case, those additional properties will be available during verification.

Redefining a Method

There may be some instances in which you want to redefine an existing method. For example, to redefine the `GetCaption` method of the `Anywin` class, you use this 4Test code:

```
winclass AnyWin : AnyWin
   GetCaption ()
      // insert method definition here
```

Confirming the Property List

You can use the `GetPropertyList` method to confirm the list of verification properties for an object. For example, the following simple test case prints the list of all the verification properties of the **Find** dialog to the results file:

```
testcase FindDialogPropertyConfirm ()
TextEditor.Search.Find.Pick ()
ListPrint (Find.GetPropertyList ())
Find.Cancel.Click ()
```

Examples

This section provides examples for defining methods and custom verification properties.

Example: Adding a Method to TextField Class

This example adds to the `TextField` class a method that selects all of the text in the text box.

```
winclass TextField : TextField
   SelectAll ()
```

```
      STRING sKey1, sKey2
      switch (GetGUIType ())
         case mswnt, msw2003
            sKey1 = "<Ctrl-Home>"
            sKey2 = "<Shift-Ctrl-End>"
         case mswvista
            sKey1 = "<Ctrl-Up>"
            sKey2 = "<Shift-Cmd-Down>"
      // return cursor to 1,1
      this.TypeKeys (sKey1)
      // highlight all text
      this.TypeKeys (sKey2)
```

The keyword this refers to the object the method is being called on.

The preceding method first decides which keys to press, based on the GUI. It then presses the key that brings the cursor to the beginning of the field. It next presses the key that highlights (selects) all the text in the field.

Example: Adding Tab Method to DialogBox Class

To add a `Tab` method to the `DialogBox` class, you could add the following 4Test code to your `frame.inc` file (or other include file):

```
winclass DialogBox : DialogBox
Tab (INTEGER iTimes optional)
if (iTimes == NULL)
 iTimes = 1
this.TypeKeys ("<tab {iTimes}>")
```

Example: Defining a Custom Verification Property

Let's look at an example of defining a custom verification property. Say you want to test a dialog box. Dialog boxes come with the following built-in verification properties:

- Caption
- Children
- DefaultButton
- Enabled
- Focus
- Rect
- State

And let's say that you have defined a class property, NumChildren, that you want to make available to the verification system.

Here is the class property definition:

```
property NumChildren
INTEGER Get ()
return ListCount (GetChildren ())
```

That property returns the number of children in the object, as follows:

- The built-in method GetChildren returns the children in the dialog box in a list.
- The built-in function ListCount returns the number of elements in the list returned by GetChildren.

To make the NumChildren class property available to the verification system (that is, to also make it a verification property) you list it as an element in the variable lsPropertyNames. So here is part of the extended DialogBox declaration that you would define in an include file:

```
winclass DialogBox : DialogBox
  // user-defined property
  property NumChildren
    INTEGER Get ()
```

```
      return ListCount (GetChildren ())
  // list of custom verification properties
  LIST OF STRING lsPropertyNames = {"NumChildren"}
```

Now when you verify a dialog box in a test case, you can verify your custom property since it will display in the list of DialogBox properties to verify.

Note: As an alternative, instead of defining NumChildren as a class property, you could also define it as a variable, then initialize the variable in a script. For example, in your include file, you would have:

```
winclass DialogBox : DialogBox
  INTEGER NumChild2
  // list of custom verification properties
  LIST OF STRING lsPropertyNames = {"NumChild2"}
```

And in your script-before you do the verification-you would initialize the value for the dialog box under test, such as:

```
Find.NumChild2 = ListCount (Find.GetChildren ())
```

Porting Tests to Other GUIs

This section describes how you can port tests to other GUIs.

Handling Differences Among GUIs

This section describes how you can handle differences between GUIs when porting tests to other GUIs.

Creating a Class that Maps to Several Silk Test Classic Classes

This functionality is supported only if you are using the Classic Agent.

Consider the Direction control in the **Find** dialog box of the Text Editor application, which allows a user to specify the direction (up or down) of searches. Suppose that this control is implemented as a check box on one GUI, but as a radio list on all other GUIs. As a radio list, the user clicks either the **Up** or the **Down** option button. As a check box, the user checks the check box to select Up, and leaves the check box unchecked to select Down.

The first step in solving this portability scenario is to create a new window class that you will use for the object on all platforms. The class you need to define, in effect, generalizes several distinct 4Test classes into one logical class.

To achieve this generalization, you:

- Derive the new class from the 4Test `Control` class, since both radio lists and check boxes derive from this class.
- Define the class with a GUI-specific tag statement for each platform. Each tag statement states the actual class of the control on the particular GUI. This allows Silk Test Classic to know what the actual class on the control will be at runtime on each of the GUIs.
- Define generalized methods that use a switch statement to branch to the 4Test code that implements the method on the particular GUI.

Here is the class declaration, which is arbitrarily named DirButton:

```
// The class is derived from Control
winclass DirButton : Control
   tag "[RadioList]"
   msw9x tag "[CheckBox]"
   void Select (LISTITEM Item optional)
      BOOLEAN bState
      switch (GetGUIType ())
         case msw9x
            bState = (Item == "Up")
```

```
            CheckBox (WndTag).SetState (bState)
         default
            RadioList (WndTag).Select (Item)
```

Note:

- The `Select` method acts against the control, regardless of whether it is a RadioList or CheckBox. The method contains a switch statement which executes the `SetState` method if the control is a check box, and the `Select` method if the control is a radio list. The `Select` method also takes an optional parameter, as indicated by the keyword optional.
- Because the tag of the object differ on each GUI, rather than specifying an identifier in the `SetState` and `Select` method calls, you use the `WndTag` property. By doing this, you force Silk Test Classic to construct a dynamic identifier for the object at runtime which will uniquely identify the object as a check box in the one case and as a radio list in all other cases.

The next step is to change your window declarations so that the control has the new class.

Continuing the example, you change the class of the control named `Direction` to **DirButton**.

```
window DialogBox Find
   tag "Find"
   parent TextEditor
   DirButton Direction
      tag "Direction"
```

Creating GUI-Specific Tags

This functionality is supported only if you are using the Classic Agent.

To close a file on one operating system, you click, for example, **File** > **Quit**, whereas on all other platforms you click **File** > **Exit**. The following window declaration accounts for these differences with two tag statements:

```
MenuItem Exit
   tag "Exit"
   [other OS] tag "Quit"
```

With this declaration, the `Exit` identifier can be used to refer to the menu item regardless of the actual label.

Conditionally Loading Include Files

If you are testing different versions of an application, such as versions that run on different platforms or versions in different languages, you probably have different include files for the different versions. For example, if your applications run under different languages, you might have text strings that display in windows defined in different include files, one per language. You want Silk Test Classic to load the proper include file for the version of the application you are currently testing.

Load Different Include Files for Different Versions of the Test Application

1. Define a compiler constant.

   For example, you might define a constant named `MyIncludeFile`.
2. Insert the following statement into your 4Test file: `use constant`.

   For example, if you defined a constant `MyIncludeFile`, insert the following statement: `use MyIncludeFile`. In this example, constant can also be an expression that evaluates to a constant at compile time.
3. When you are ready to compile your 4Test files, specify the file name of the include file you want loaded as the value of the constant in the **Compiler Constants** dialog box.

   Be sure to enclose the value in quotation marks if it is a string.

4. Compile your code.

Silk Test Classic evaluates all compiler constants and substitutes their values for the constants in your code. In this case, the constant `MyIncludeFile` will be evaluated to a file, which will be loaded through the `use` statement.

Deciding which Form of Tag to Use

This functionality is supported only if you are using the Classic Agent.

When an object's caption or label changes on a different GUI, it is usually preferable to use multiple tags, each based on the GUI-specific label or caption, instead of using the index. Not only does it make your declarations easier to understand, but it shields your test cases from changes to the sequence of child objects. For example, if the Exit item changes so that it is the fourth item and not the fifth, your test cases will still run.

Different Error Messages

The `VerifyErrorBox` function, shown below, illustrates how to solve the problem of different error messages on each GUI platform. For example, if a GUI platform always adds the prefix "Error:" to its message, while the other platforms do not, you might use or create a GUI Specifier for that platform and then use the `VerifyErrorBox` function as follows:

```
VerifyErrorBox (STRING sMsg)
   // verifies that the error box has the correct error
   // message, then dismisses the error box

   const ERROR_PREFIX = "ERROR: "
   const ERROR_PREFIX_LEN = Len (ERROR_PREFIX)
   STRING sActMsg = MessageBox.Message.GetText ()

   // strip prefix "ERROR: " from GUI Specifier for that platform error
messages
   if (GetGUIType () == GUI Specifier for that platform)
      sActMsg = SubStr (sActMsg, ERROR_PREFIX_LEN + 1)

   Verify (sActMsg, sMsg)
   MessageBox.Accept ()
```

One Logical Control can Have Two Implementations

Consider the case where the same logical control in your application is implemented using different classes on different GUIs.

If the kinds of actions you can perform against the object classes are similar, and if Silk Test Classic uses the same method names for the actions, then you do not have a portability problem to address.

For example, the methods for the `RadioList` and `PopupList` classes have identical names, because the actions being performed by the methods are similar. Therefore, if a control in your application is a popup list on one GUI and a radio list on another, your scripts are already portable.

If the two object classes do not have similar methods, or if the methods have different names, then you need to port your scripts.

Options Sets and Porting

Options sets save all current options except General Options. Options sets can be very useful when trying to use the same scripts on different operating systems. The primary differences between the two may be compiler constants.

For example, you might use the compiler constant *sCmdLine*. Usually, the command line to invoke an application differs between the PC operating systems. You could create a compiler constant (note that there is a string limit on compiler constants) for use in the *sCmdLine* constant to differentiate between the

platforms' command lines. You might also use a compiler constant for methods that work slightly differently on the two operating systems, such as the `Pick()` methods.

Specifying Options Sets

In a test plan, you can specify options sets to be used with the test plan or parts of it. You use options sets to automatically run different tests that require different options without having to manually open options sets.

To ensure that everyone working on a project has the same options settings (such as class mapping), do one of the following:

- Open an Options Set.
- Set these option values at runtime.
- Specify the following statement in the test plan: `optionset: filename.opt`.

Dependent test cases will run with the specified options set opened. The options set will be closed when it passes out of scope. If you don't specify a full path name, the file is considered to be in a directory relative to the directory containing the current test plan or sub-plan.

Remember:

- Options can also be set at runtime in a test script by using the `Agent` method, `SetOption`, and passing in the name of the option and its value.
- Many Agent options and their values are found in the **Agent Options** dialog box.
- Agent options can be set in a `testcase/` function.
- Class map settings, set at runtime, are best set before any tests are executed (for example, in ScriptEnter) and after each test case (for example TestcaseExit) in case any have been changed in the course of a test case.
- Class mappings set at runtime using the Agent method `SetOption` are only in effect during test execution; these settings are not available to the recorders.

Supporting Differences in Application Behavior

Although you can account for differences in the appearance of your application in the window declarations, if the application's behavior is fundamentally different when ported, you need to modify your test cases themselves. To modify your test cases, you write sections of 4Test code that are platform-specific, and then branch to the correct section of code using the return value from the `GetGUIType` built-in function.

This topic shows how to use the `GetGUIType` function in conjunction with `if` statements and the `switch` statements.

Switch statements

You can use GUI specifiers before an entire `switch` statement and before individual statements within a `case` clause, but you cannot use GUI specifiers before entire `case` clauses.

```
testcase GUISwitchExample()
INTEGER i
FOR i=1 to 5
mswxp, mswnt switch(i)

// legal:
mswxp, mswnt switch (i)
   case 1
      mswxp Print ("hello")
      mswnt Print ("goodbye")
   case 2
      mswxp raise 1, "error"
      mswnt Print ("continue")
   default
      mswxp Print ("ok")
```

```
// NOT legal:
switch (i)
   mswxp case 1
      Print ("hello")
   mswnt case 1
      Print ("goodbye")
```

If statements

You can use GUI specifiers in if statements, as long as GUI specifiers used within the statement are subsets of any GUI specifiers that enclose the entire `if` statement.

```
// legal because no GUI specifier
// enclosing entire if statement:
if (i==j)
   msw32, mswnt Print ("hi")
   msw2000 Print ("bye")

// legal because msw is a subset of enclosing specifier:
msw32, msw2000 if (i==j)
   mswnt Print("hi")

// legal for the same reason as preceding example:
msw32, msw2000 if (i==j)
   Print ("hi")
mswnt else
   Print ("Not the same")

// NOT legal because msw2000 is not a subset
// of the enclosing GUI specifier msw:
msw32 if (i==j)
   msw2000 Print ("bye")  // Invalid GUI type
```

If you are trying to test multiple conditions, then you should use a `select` or `switch` block. You could use nested `if.else` statements, but if you have more than two or three conditions, the levels of indentation will become cumbersome.

You should not use an `if..else if..else` block. Although `if..else if..else` will work, it will be difficult to troubleshoot exceptions that occur because the results file will always point to the first `if` statement even if it was actually a subsequent `if` statement that raised the exception.

For example, in the following test case, the third string, `Not a date`, will raise the exception:

*** Error: Incompatible types -- 'Not a date' is not a valid date

The exception actually occurs in the lines containing:

```
GetDateTimePart ([DATETIME]sVal, DTP_YEAR) == 2006
```

For the nested `if..else` and the `select` blocks, the results file points to those lines as the sources of the exceptions. However, for the `if..else if..else` block, the results file points to the first `if` statement, in other words to the line:

```
[-] if IsNull (sVal)
```

even though that line clearly is not the source of the exception because it does not concern DATETIME values.

```
[+] testcase IfElseIfElse ()
[-] LIST OF STRING lsVals = {...}
[ ] "2006-05-20"
[ ] "2006-11-07"
[ ] "Not a date"
[ ] STRING sVal
[ ]
[-] for each sVal in lsVals
```

```
[-] do
[-] if IsNull (sVal)
[ ] Print ("No date given")
[-] else if sVal == FormatDateTime (GetDateTime (), "yyyy-mm-dd")
[ ] Print ("The date is today")
[-] else if GetDateTimePart ([DATETIME]sVal, DTP_YEAR) == 2006
[ ] Print ("The year is this year")
[-] else
[ ] Print ("Some other year")
[-] except
[ ] ExceptLog ()
[ ]
[-] do
[-] if IsNull (sVal)
[ ] Print ("No date given")
[-] else
[-] if sVal == FormatDateTime (GetDateTime (), "yyyy-mm-dd")
[ ] Print ("The date is today")
[-] else
[-] if GetDateTimePart ([DATETIME]sVal, DTP_YEAR) == 2006
[ ] Print ("The year is this year")
[-] else
[ ] Print ("Some other year")
[-] except
[ ] ExceptLog ()
[ ]
[-] do
[-] select
[-] case IsNull (sVal)
[ ] Print ("No date given")
[-] case sVal == FormatDateTime (GetDateTime (), "yyyy-mm-dd")
[ ] Print ("The date is today")
[-] case GetDateTimePart ([DATETIME]sVal, DTP_YEAR) == 2006
[ ] Print ("The year is this year")
[-] default
[ ] Print ("Some other year")
[-] except
[ ] ExceptLog ()
[ ]
```

Text Box Requires Return Keystroke

On some GUIs, the `Enter`/`Return` key must be pressed after data is entered into a text box. Suppose you want to create a test case that enters invalid data into the text box, and then checks if the application detects the error. After the test case enters the invalid data, it needs to use the `GetGUIType` function to determine the GUI, and then press the `Return` key if the GUI requires it.

For example:

```
// code to enter an invalid string into field
if (GetGUIType () == mswnt)
   MyTextField.TypeKeys ("<Return>")
// code to verify that application detected error
```

Using Cross-Platform Methods in Your Scripts

In scripts, you can use your cross-platform method names. The window declarations map the cross-platform method names you use in your scripts to the actual methods required to carry out the actions you want on each of the GUIs.

Continuing the example from *Creating a Class that Maps to Several Silk Test Classic Classes*, you use the `Select` method in your code to select the control named `Direction`.

```
testcase SearchBackward ()

   LISTITEM Item
   Item = "Up"
   Find.Invoke ()
   Find.Direction.Select (Item)
   .
   .
   .
   Find.Dismiss ()
```

Note: The script does not indicate that anything unusual is happening. All of the steps necessary to make the `Select` method work properly, regardless of the class of the object, are encapsulated in the class and window declarations.

Using the Index as the Tag

This functionality is supported only if you are using the Classic Agent.

If you are certain that an object's position in relation to its sibling objects of the same class will remain the same when the application is ported, you can use the index form for the tag.

Repeating the example from the preceding section, because the Exit/Quit menu item is the fifth menu item on the **File** menu (on all platforms), you can use the index form for the tag (#5) as shown here:

```
MenuItem Exit
   tag "#5"
```

About GUI Specifiers

This section describes GUI specifiers.

Class Declarations

Be careful using GUI specifiers before class declarations; they can be ambiguous. Any ambiguities must be resolvable at compile-time.

```
// bad style:
msw winclass myclass
mswnt winclass myclass
window myclass inst // Ambiguous. Is it an instance of
                    // the msw class or the mswnt class?
```

The preceding example's ambiguity can be resolved by specifying a GUI target with conditional compilation (so that, for example, only code for msw gets compiled, in which case inst would be an instance of the msw class or by explicitly using a GUI specifier for the window, as follows:

```
// good style:
msw winclass myclass
mswnt winclass myclass
msw window myclass inst
```

Conditional Compilation

If you have GUI-specific code in your scripts and declarations, you can have Silk Test Classic conditionally compile your code based on the values of the GUI specifiers - only code specific to a particular GUI is compiled (as well, of course, as all code that is not GUI-specific). This has the following two advantages:

- The compilation is faster.
- The resulting code is smaller and requires less memory to run.

You can also cause conditional compilation by using constants, which are evaluated at compile time.

Constants are not restricted to conditional compilation. You can use constants for any value that you want resolved at compile time.

Conditionally Compile Code

1. Prefix any 4Test statements that are GUI-specific with the appropriate GUI specifier.
2. Specify the platforms that you want to compile for by entering the appropriate GUI specifiers in the **GUI Targets** field in the **Runtime Options** dialog box. You can specify as many GUI targets as you want; separate each GUI specifier by a comma.

   Setting a GUI target affects which classes are listed in the **Library Browser**.
3. To conditionalize code based on the value of constants you define, do the following:
   1. Click **Compiler Constants** in the **Runtime Options** dialog box.
   2. The **Compiler Constants** dialog box is displayed.
   3. Define a constant and specify its value.
   4. Use the constant in your code anywhere you can specify an expression.
4. Click **OK** to close the **Runtime Options** dialog box.

GUI with Inheritance

When using GUI specifiers for parent classes, you must explicitly use the GUI specifiers with the descendants:

```
mswxp winclass newclass
mswxp winclass subclass : newclass
mswxp window subclass inst
```

GUI with Global Variables

Be careful when using GUI specifiers with global variables, because Silk Test Classic initializes global variables before connecting to an Agent. This might not give you the results you want if you are doing distributed testing.

Let's say that you are running tests on a remote machine that is listed in the **Runtime Options** dialog box. Because Silk Test Classic initializes all global variables before connecting to an Agent, any GUI specifier at the global level will initialize to the host machine, not the target machine you want to test against.

For example, say the host machine is running a different operating system than the target machine. Consider the following script:

```
mswxp STRING sVar1 = SYS_GetEnv("UserName")

mswxp STRING sVar1 = SYS_GetRegistryValue
   (HKEY_LOCAL_MACHINE,     "System\CurrentControlSet\Control", "Current
User")

main()
   print(sVar1)
```

This script fails, with the error message:

```
*** Error: Registry entry 'Current User' not found
```

because `sVar1` is initialized to the value for the host system, not the target system.

Constants behave similarly to global variables if you use a GUI specifier to initialize the variable (or constant). It is a good idea to use GUI specifiers in the main function, under **Options** > **Runtime** or another function that is called after the Agent is contacted.

Marking 4Test Code as GUI Specific

Using Silk Test Classic, you can create portable test cases that will test your application on any of the supported GUIs. The reason for this is that your test cases use logical names, called identifiers, to refer to

the GUI objects, and not actual names, called tags. Therefore, if there are differences in the ported application's appearance, you need only change the window declarations, not the test cases themselves.

The porting scenarios described section use 4Test keywords called GUI specifiers to indicate that portions of include files or script files are specific to a particular GUI. Before studying these scenarios, you should understand which GUI specifiers are available and how to use them in your include files and script files.

4Test includes a long list of GUI specifiers.

Syntax of a GUI Specifier

A GUI specifier has this syntax:

```
[[gui-type [,gui-type]] | [!gui-type]]
```

`gui-type` is the GUI. You can express this in one of two mutually exclusive ways. For example, you can specify one or more GUIs separated by commas, as in:

```
mswxp, mswin7
```

Or you can specify all but one GUI, as in the following, which indicates that what follows applies to all environments except Windows NT-based operating systems:

```
! mswnt
```

What Happens when the Code is Compiled

Only code relevant to the GUI environments specified in the GUI Targets field (plus all common code) will be compiled. If you do not list any GUI specifiers in the GUI Targets field, all code will be compiled; at runtime, code not relevant to the platform the application is running on will be skipped.

The constants you have defined are evaluated and used to compile the code. You can use this feature to conditionally load include files.

Where You Use GUI Specifiers

A GUI specifier can be located before any 4Test declaration or statement except the `use` statement, which must be evaluated at compile time, with the following exceptions:

- `Switch` statements
- `If` statements
- `Type` statements
- `Do… except` statements
- `Class` declarations
- GUI with inheritance
- GUI with global variables

If you try to use a browser specifier instead of a GUI specifier to specify a window, Silk Test Classic will generate an error. The primary use of browser specifiers is to address differences in window declarations between different browsers. Each Agent connection maintains its own browser type, allowing different threads to interact with different browsers.

do...except Statements

You can use GUI specifiers to enclose an entire `do...except` statement before individual statements, but you cannot use GUI specifiers before the `except` clause.

```
// legal:
do
   mswxp Verify (expr1,expr2)
   mswin7 Verify (expr3,expr4)
except
   mswin7 reraise
   mswxp if (ExceptNum () == 1)
```

```
      Print ("err, etc.")
// NOT legal:
mswin7 do
   Verify (expr,expr)
mswxp except
   reraise
```

Type Statements

You can use a GUI specifier before a type `type ... is` enum or `type ... is` set statement, but not before an individual value within the type declaration.

Supporting GUI-Specific Objects

This section describes how Silk Test Classic supports testing GUI-specific objects.

Supporting GUI-Specific Captions

Classic Agent

When you are using the Classic Agent, by default Silk Test Classic bases the tag for an object on the actual caption or label of the object. If the captions or labels change when the application is ported to a different GUI, you have two options:

- You can have multiple tags, each based on the platform-specific caption or label.
- You can have a single tag, using the index form of the tag, as long the relative position of the object is the same in the ported versions of the application.

Then, in your test cases, you can use the same identifier to refer to the object regardless of what the object's actual label or caption is.

Open Agent

When you are using the Open Agent, Silk Test Classic creates locator keywords in an INC file to create scripts that use dynamic object recognition and window declarations. The locator is the actual name of the object, as opposed to the identifier, which is the logical name. Silk Test Classic uses the locator to identify objects in the application when executing test cases. Test cases never use the locator to refer to an object; they always use the identifier.

The advantages of using locators with an INC file include:

- You combine the advantages of INC files with the advantages of dynamic object recognition. For example, scripts can use window names in the same manner as traditional, Silk Test Classic tag-based scripts and leverage the power of XPath queries.
- Enhancing legacy INC files with locators facilitates a smooth transition from using hierarchical object recognition to new scripts that use dynamic object recognition. You use dynamic object recognition but your scripts look and feel like traditional, Silk Test Classic tag-based scripts that use hierarchical object recognition.
- You can use `AutoComplete` to assist in script creation. `AutoComplete` requires an INC file.

Supporting GUI-Specific Executables

The command to start the application will almost always be different on each GUI. The `Invoke` method of Silk Test Classic expects to find the command in the constant *sCmdLine*, which is defined in the main window declaration of your application. You should declare as many *sCmdLine* variables as there are GUIs on which your application runs, beginning each declaration with the appropriate GUI specifier.

For example, the following constants specify how Silk Test Classic should start the Text Editor application on Windows and Windows Vista:

```
msw32 const sCmdLine = "c:\program files\<SilkTest install directory>\silktest
\textedit.exe"
mswvista const sCmdLine = "{SYS_GetEnv('SEGUE_APPS')}/SilkTest/demo/textedit"
```

Supporting GUI-Specific Menu Hierarchies

When an application is ported, there are two common structural differences in the menu hierarchy:

- The menu bar contains a platform-specific menu.
- A menu contains different menu items.

To illustrate the case of the platform-specific menu, consider the Microsoft Windows system menu or a Vista menu (for example). Silk Test Classic recognizes these kinds of standard GUI- specific menus and includes the appropriate GUI specifier for them when you record declarations.

For menus that Silk Test Classic does not recognize as platform-specific, you should preface the window declaration with the appropriate GUI specifier.

Different menu items - example

To illustrate the case of different menu items, suppose that the **Edit** menu for the Text Editor application has a menu item named **Clear** which displays on the Windows version only. The declaration for the **Edit** menu should look like the following:

Classic Agent	Open Agent

`Menu Edit`
`  tag "Edit"`
`  msw32 MenuItem Clear`
`  tag "Clear"`
`  MenuItem Undo`
`  tag "Undo"`
	`Menu Edit`
	`  locator "Edit"`
	`  msw32 MenuItem Clear`
	`  locator "Clear"`
	`  MenuItem Undo`
	`  locator "Undo"`

Supporting Custom Controls

This sections describes how Silk Test Classic supports custom controls.

Why Silk Test Classic Sees Controls as Custom Controls

A control is defined by the following:

- The actual class name of the control.
- The underlying software code that creates and manipulates the control.

Whenever the definition of a control varies from the standard, Silk Test Classic defines the control as a custom control. During recording, Silk Test Classic attempts to identify the class of each control in your GUI and to assign the appropriate class from the built-in class hierarchy. If a control does not correspond to one of the built-in classes, Silk Test Classic designates the control as a custom control.

- When you are using the Classic Agent, Silk Test Classic assigns custom controls to the `CustomWin` class.
- When you are using the Open Agent, Silk Test Classic assigns custom controls to the `Control` class or another class.

Classic Agent Example

For example, Silk Test Classic supports the standard MFC library, which is a library of functions that allow for the creation of controls and the mechanism of interaction with them. In supporting these libraries, Silk Test Classic contains algorithms to interrogate the controls based upon the standard libraries. When these algorithms do not work, Silk Test Classic reports the control as a `CustomWin`.

Suppose that you see a text box in a window in your application under test. It looks like a normal text field, but Silk Test Classic calls it a control of the class `CustomWin`.

Reasons Why Silk Test Classic Sees the Control as a Custom Control

For the following reasons Silk Test Classic might recognize a control as a custom control:

- The control is not named with the standard name upon the definition of the control in the application under test. For example, when a **TextField** is named **EnterTextRegion**. If this is the only reason why Silk Test Classic recognizes the control as a custom control, then you can class map the control to the standard name.

  The class mapping might not work. The class mapping will work if the control is not really a custom control, but rather a standard control with a non-standard name. Try this as your first attempt at dealing with a custom control.
- If the class mapping does not work the control truly is a custom control. The software in the application under test that creates and manipulates the control is not from the standard library. That means that the Silk Test Classic algorithms written to interrogate this kind of control will not work, and other approaches will have to be used to manipulate the control.

When you are using the Classic Agent, the support for custom controls depends on whether the control is a graphical control, such as a tool bar, or a non-graphical control, such as a text box.

Supporting Graphical Controls

If an application contains a graphical area, for example a tool bar, which is actually composed of a discrete number of graphical controls, Silk Test Classic records a single declaration for the entire graphical area; it does not understand that the area contains individual controls.

Custom Controls (Classic Agent)

This section describes how the Classic Agent supports custom controls.

Mapping Custom Classes to Standard Classes

This functionality is supported only if you are using the Classic Agent.

When a control shows up in the **Record Window Declarations** dialog box as a `CustomWin`, but is actually a standard GUI object which the application developers have renamed, you should map the custom class name to the built-in class name while in the dialog box.

Among the standard classes the Silk Test Classic displays in the **Class Map** dialog box, there are the following three classes that can be described as "meta" classes:

- `BUTTON` causes the Agent to treat the object as a kind of button, whether it be an instance of `PushButton`, `CheckBox`, or `RadioButton`. The kind of button depends on the object's style bits.
- `STATIC` causes the Agent to treat the object as Static Text if the appropriate style bits (for example, `SS_LEFT` and `SS_CENTER`) are set. Otherwise, only the methods of the `AnyWin` class apply.

- `MDI` client windows are containers that sit between application frame windows and MDI document windows. Mapping a custom object to `MDICLIENT` means you do not need a tag for it in order to refer to one of its children. You cannot perform operations on them.

Perform a Class Mapping when a Declaration for a CustomWin Displays in the Record Window Declaration Dialog

This functionality is supported only if you are using the Classic Agent.

When a declaration for a `CustomWin` displays in the **Record Window Declarations** dialog box, perform the following steps:

1. Press **Ctrl+Alt**. The declarations are frozen in the **Window Declaration** list box in the lower half of the **Record Window Declarations** dialog box.
2. In the **Window Declarations** list box, click the line containing the declaration for the custom object. The line is highlighted and the declaration for the `CustomWin` displays in the **Window Detail** group box.
3. In the **Window Detail** group box, click **Class Map**. The **Class Map** dialog box opens. The name of the custom class is displayed in the **Custom Class** text box.
4. In the **Standard Class** field, enter the name of the built-in 4Test class to which you are mapping the custom class.
5. Click **Add**. Silk Test Classic adds the class name.
6. Click **OK**.

When you resume recording, the object has the standard 4Test class. If Silk Test Classic encounters a similar object, it automatically maps the object to the correct 4Test classes. You must modify any prerecorded declarations containing these objects to use the standard class.

Non-Graphical Custom Controls

This functionality is supported only if you are using the Classic Agent.

Non-graphical custom controls are controls which are not owner drawn. If your application uses a non-graphical control that does not map to any of the controls that are supported by Silk Test Classic, you have the following options:

- If the developer of the application has created DLLs to interact with the custom object, you can call the DLL functions from a script.
- Otherwise, and only if you are working with the Classic Agent, you can add partial support for the non-graphical custom object by creating a new class, derived from `AnyWin`. Then, to implement the methods for the non-graphical control, you can write methods that use the primitive methods of the `AnyWin` class, like `TypeKeys` and `MoveMouse`. You have to manually change the scripts for your application, because the custom methods cannot be recorded.

Example

In the following sample code, the class *MyCustomTextField*, which is a custom control in the application *myApplication*, is derived from the `AnyWin` class. The method `TypeText` is added to *MyCustomTextField*, and performs a `Click` and a `TypeKeys`.

```
const wDynamicMainWindow = MyApplication

window MainWin MyApplication
        locator "/MainWin[@caption=My Application']"

  // The working directory of the application when it is invoked
  const sDir = "%USERPROFILE%"

  // The command line used to invoke the application
  const sCmdLine = "C:\myApplication.exe"
```

```
  MyCustomTextField TextField
    tag "TextField"

winclass MyCustomTextField : AnyWin
  void TypeText(string text)
    Click(1, 0)
    TypeKeys(text)
```

The test case *Test1* calls the `TypeText` method.

```
testcase Test1 ()
     recording
          MyApplication.SetActive()
          MyApplication.TextField.TypeText("test")
```

Adding xy Coordinates to a Declaration

This functionality is supported only if you are using the Classic Agent.

Record the x, y coordinates of a graphical control, such as a toolbar, to add them to a window declaration:

1. Position the cursor in the window declaration at the end of the tag to which you want to add coordinates.
2. Type a slash character `/`.
3. Click **Record** > **Window Locations** to open the **Record Window Locations** dialog box.
4. Track the cursor over the object.

   The dialog box displays the name of the object and its x,y coordinates relative to the screen, the frame, which is the main window and its window decoration, and the client, which is the main window minus its window decoration.
5. Press **Ctrl+Alt** to freeze the declaration.
6. Since this procedure is appending the location to a window declaration, click **Client option**.
7. Click **Paste to Editor**, and then click **Close**.

Modify Declarations for Each of the Icons Contained in an Evenly Sized and Spaced Tool Bar

This functionality is supported only if you are using the Classic Agent.

1. In the window declarations file, make as many copies of this recorded declaration as there are discrete objects.
2. You can retain the original class (`CustomWin`) if the functionality inherited from the `AnyWin` class is sufficient. Or you can specify the name of a class you create that contains the added functionality you need.
3. Change the identifier to some string that represents the icon's action.
4. Append the tag with the icon's location suffix in the tool bar. You express the location using this syntax:

   ```
   (column:total-columns, row:total-rows)
   ```

   For example, you specify the icon in the third column, first row, like this:

   ```
    (3:26, 1:1)
   ```

   You append this location to the tag with the forward slash (/) character.

Adding a Location Suffix to the Tag of a Declaration

This functionality is supported only if you are using the Classic Agent.

You can, however, create declarations for each discrete object. To do this, make as many copies of the original recorded declaration as there are discrete objects. Then add a location suffix to the tag in each declaration, which is the location of the object within the graphical area.

Silk Test Classic provides two ways to specify the location suffix of contained graphical objects, depending on the size and spacing of the control.

Controls that are sized and spaced evenly in a grid

If a group of graphical controls are equal in size and evenly spaced in a grid, you can specify the location of each control as column y of the total number of columns and row x of the total number of rows. This syntax is both cross-platform and resolution independent.

Controls that are sized and spaced irregularly in a grid

If the graphical controls in a group are not the same size or are not evenly spaced in a grid, you need to specify in the declaration the location suffix of each control as an exact x,y point. This x,y point typically corresponds to the center of the object. This syntax is not necessarily cross-platform or resolution independent.

You specify a location in its declaration as an x,y coordinate using the following syntax:

```
(x, y)
```

You append this location to the tag with the forward slash (/) character.

Silk Test Classic Does Not Recognize the Class of a Control

This functionality is supported only if you are using the Classic Agent.

While using the **Record Window Declarations** dialog box, you will on occasion notice that the recorded class is `CustomWin`, indicating that Silk Test Classic does not recognize the class of the object. The object is interpreted as a custom object. Custom objects often look and act the same as their corresponding known, standard objects and they may correspond to built-in, known classes.

If the object is in fact a custom object, to be able to record and run test cases that interact with the custom object, see *Mapping Custom Classes to Standard Classes*.

However, if the object is not actually a custom object, but is instead a standard object that your application's developers have renamed, you can record and run test cases merely by establishing a class map between the renamed class and the standard 4Test class. You can also filter out unneeded custom classes from the class hierarchy.

Class mapping only works for objects that are created with standard API calls but are given non-standard names.

Supporting Custom Text Fields

This functionality is supported only if you are using the Classic Agent.

Suppose your application has an object that acts like a text box, but which is not implemented using your GUI's standard text box object. The following example illustrates how you can derive a new class from `AnyWin` and define methods for the custom object. The example defines the `ClearText`, `GetMultiText`, `SetMultiText`, and `GetMultiSelText` methods.

```
// This new class defines methods that re-implement
// the methods of the TextField class so that they will
// work on custom text boxes. To be able to use these
// methods, you must change the class of object in the
// declarations from CustomWin to CustomTextField.

winclass CustomTextField : AnyWin

   // This method clears the text box by moving the
```

```
// cursor to the start of field, selecting the text
// to the end of the file, and deleting the selected
// text

void ClearText ()
   TypeKeys ("<Ctrl-Home>")
   TypeKeys ("<Ctrl-Shift-End>")
   TypeKeys ("<Backspace>")

// This method writes text to the text field.
// It first calls ClearText and then uses TypeKeys to
// input the text passed in.

void SetMultiText (STRING sText)
   ClearText ()
   TypeKeys (sText)

// This copies the currently selected text to the
// clipboard and returns the clipboard contents.

LIST OF STRING GetMultiSelText ()
   Clipboard.SetText () // Clear the clipboard
   TypeKeys ("<Ctrl-Insert>")
   return (Clipboard.GetText ())

// This method highlights all of the text in the
// text field, copies the highlighted text to the
// clipboard, and returns the clipboard contents.

LIST OF STRING GetMultiText ()
   Clipboard.SetText () // Clear the clipboard
   TypeKeys ("<Ctrl-Home>")
   TypeKeys ("<Ctrl-Shift-End>")
   TypeKeys ("<Ctrl-Insert>")
   TypeKeys ("<Left>")
   return (Clipboard.GetText ())
```

Supporting Custom List Boxes

This functionality is supported only if you are using the Classic Agent.

To support custom list boxes, you can write "low-level" methods. Or, a second approach is to implement the necessary Microsoft Windows messages and set the necessary Windows style bits so that you can use the standard list box methods on your custom list box. The Microsoft Windows messages which you must implement are:

- `LB_GETCOUNT`
- `LB_GETTEXT`
- `LB_GETITEMRECT`
- `LB_GETTOPINDEX`
- `LB_GETSEL`
- `LB_GETTEXTLEN`

And the Windows style bits that you must set are:

- `WS_VSCROLL`
- `LBS_EXTENDEDSEL`
- `LBS_MULTIPLESEL`

Using Clipboard Methods

If you are having trouble getting or setting information with a custom object that contains text, you might want to try the 4Test Clipboard methods. For example, assume you have a class, `CustomTextBuffer`,

which is similar to a `TextField`, but using the `GetText` and `SetText` methods of the `TextField` does not work with the `CustomTextBuffer`. In such a case, you can use the `GetText` and `SetText` methods of the `ClipboardClass`.

Get and Set Text Sample Code

The following sample code retrieves the contents of the `CustomTextBuffer` by placing it on the **Clipboard**, then printing the **Clipboard** contents:

```
// Go to beginning of text field
CustomTextBuffer.TypeKeys ("<Ctrl-Home>")
// Highlight it
CustomTextBuffer.TypeKeys ("<Ctrl-Shift-End>")
// Copy it to the Clipboard
CustomTextBuffer.TypeKeys ("<Ctrl-Insert>")
// Print the contents of the Clipboard
Print (Clipboard.GetText())
```

Setting text

Similarly, the following sample code inserts text into the custom object by pasting it from the Clipboard:

```
// Go to beginning of text field
CustomTextBuffer.TypeKeys ("<Ctrl-Home>")
// Highlight it
CustomTextBuffer.TypeKeys ("<Ctrl-Shift-End>")
// Paste the Clipboard contents into the text field
CustomTextBuffer.TypeKeys ("<Shift-Insert>")
```

You can wrap this functionality in `GetText` and `SetText` methods you define for your custom class, similar to what was shown in supporting custom text boxes.

Using the Modified Declaration

Once you create window declarations like these for the graphical objects in your application, you can manipulate them as you would any other object. For example, if the tool bar was contained in an application named `MyApp`, to click on the **FileOpen** icon in the tool bar, you use the following command:

```
MyApp.FileOpen.Click()
```

You need to write this statement, and others that access the objects declared above, such as Save and Printer, by hand. **Record** > **Testcase** and **Record** > **Actions** will not use these identifiers.

Filtering Custom Classes

This section describes how you can filter custom classes.

Using Class Mapping to Filter Custom Classes

This functionality is supported only if you are using the Classic Agent.

You can use 4Test to filter out unnecessary classes, such as invisible containers. Ignoring these unnecessary classes simplifies the object hierarchy and shortens the length of the lines of code in your test scripts and functions. Container classes or ‘frames’ are common in GUI development, but might not be necessary for testing.

You enable Silk Test Classic to ignore instances of a container class through the **Class Map** dialog box. When you enable Silk Test Classic to ignore these classes, the classes are not recorded.

You can suppress the controls for certain classes for .NET, Java SWT, and Windows API-based applications. Other extensions do not support this type of class-mapping for window classes, so you must modify the extension .ini file in order to ignore window classes in the Java or ActiveX/VB extensions.

After filtering out a custom container class, you may lose the ability to see its child objects.

You can also map a class to LookUnder to look ‘through’ the class, seeing the objects under it. For example, there is a `BlackFrame` class in Visual Basic that is a 3-D black border around controls. LookUnder is not in the standard classes list, so if you want to use it you must type it in.

You can use class mapping to filter out the following:

- Object layers to ignore non-logical windows.
- Extra or "invisible" window layers in the object hierarchy.
- An overlaying window that obstructs an object under it.

Overview of Style-Bits

This functionality is supported only if you are using the Classic Agent.

Classes may allow different styles of instantiation. Style-bits determine the styles that can be applied to an object. For example, a PushButton, CheckBox, and RadioButton are all variants of the native Windows Button class. They are all the same class, but each has different style-bit to determine the specific look and behavior of the button.

You can map not only an entire class, but also specific ‘styles’ of one class to another known class.

Class Mapping with Style-Bits

This functionality is supported only if you are using the Classic Agent.

1. In the **Custom Class** text box of the **Class Map** dialog box, enter the class name, style-bit, and style-mask.

   The style-mask tells Silk Test Classic which parts of the style-bit to use. As a general rule, repeat the style-bit.
2. In the **Standard Class** text box, enter the known class.

   This can be any standard 4Test class or a user-defined class.
3. Click **Add** and then click **OK**.

Note: The style-mask can also take the format `0xFFFFFFFF`, where ‘F’ includes the bit and ‘0’ turns it off. For example, one custom PageList might look like `0x12345678` and another of the same type might look like `0x12345679`. You can class map it like:

```
newpageclass,0x12345678,0x12345678=PageList
newpageclass,0x12345679,0x12345679=PageList
```

And on and on for each one like it, or…

```
newpageclass,0x12345678,0xFFFFFFF0=PageList
```

What this says is that every instance of `newpageclass 0x1234567?` should be class mapped to PageList. The last bit, being turned off, is ignored.

Invisible Containers

Sometimes a window contains an invisible dialog box that contains controls. You can set these "dialog box containers" to Ignore using class mapping and style-bits in order to avoid making all of the dialog boxes disappear.

See the following examples for details.

Example: WordPad with No Class Mappings

```
[-] window MainWin WordPad
[+] multitag "*WordPad"
[+] Menu File
[+] Menu Edit
[+] Menu View
[+] Menu Insert
```

```
[+] Menu Format
[+] Menu Help
// toolbars seen, but are nested
[+] CustomWin BottomStatusBar
[-] CustomWin Frame
[+] CustomWin FormatBar
[+] ComboBox ComboBox1
[+] ComboBox ComboBox2
[+] CustomWin StandardBar
[+] CustomWin Ruler
[-] main ()
WordPad.Frame.FormatBar.ComboBox1.Select ("Arial")
```

Example: WordPad with AfxControlBar Ignored

```
[-] window MainWin WordPad
[+] multitag "*WordPad"
[+] Menu File
[+] Menu Edit
[+] Menu View
[+] Menu Insert
[+] Menu Format
[+] Menu Help
// toolbars, ruler, and statusbar not seen
[+] ComboBox ComboBox1
[+] ComboBox ComboBox2
[+] TextField Document
[-] main ()
WordPad.ComboBox1.Select ("Arial")
```

Example: Class Mapping Using Style-Bits

```
[-] MainWin MyApp
[+] multitag "My App"

// the following are default declarations; these are really
// a push button, checkbox, and a radio button
[-] CustomWin AVButton
tag "[AVButton]#1"
[-] CustomWin AVButton2
tag "[AVButton]#2"
[-] CustomWin AVButton3
tag "[AVButton]#3"

// in the Class Map Dialog set up class mapping
// using style bits:
// class map AVButton,0x0000,0x0000 to PushButton
// class map AVButton,0x0003,0x0003 to CheckBox
// class map AVButton,0x0004,0x0004 to RadioButton
[-] MainWin MyApp
[+] multitag "My App"

// notice that the following windows are now declared correctly
[-] PushButton Close
tag "Close"
[-] CheckBox PrintReport
tag "Print Report"
[-] RadioButton PieChart
tag "Pie Chart"
```

OCR Support

This functionality is supported only if you are using the Classic Agent.

Silk Test Classic provides a 4Test include file, `OCR.inc`, which contains two 4Test functions that are used to perform optical character recognition (OCR). One function converts bitmap files to text. The other allows you to pass in a window identifier and extract the text from the window (or a region of the window). To use the 4Test OCR functions, include `OCR.inc` in your test script or include file, or add the include file through the **Use Files** text box in the **Runtime Options** dialog box. To include the function documentation in the library browser, add `OCR.txt` through the **Help files for library browser** text box in the **General Options** dialog box.

The 4Test functions call functions in a Silk DLL file that extends the third-party Textract DLL file from Structu Rise. The Textract DLL uses a font pattern database file to recognize text of certain sizes and styles for fonts that are specified in an initialization file. Although a default version of the font pattern database is installed with Silk Test Classic, we recommend that you configure the font pattern database to include the fonts used by your application.

The Silk Test Classic OCR Module

This functionality is supported only if you are using the Classic Agent.

The following files comprise the OCR module provided with Silk Test Classic. All of the files must reside in the Silk Test directory:

Exgui.exe	Utility for generating the font pattern database. Can also be used as a standalone utility for text recognition.
OCR.inc	The 4Test include file that provides high-level 4Test functions based on the functions in `SgOcrLib.dll`.
SgOcrLib.dll	Borland extension of `Textract.dll` that provides high-level functions.
SgOcrLib.inc	Declares the functions in `SgOcrLib.dll` so that they can be called in 4Test.
SgOcrPattern.pat	Font pattern database that controls text conversion.
Textract.dll	Textract OCR DLL provided by Structu Rise.
Textract.ini	Initialization file for the Textract OCR DLL. The following two sections contain settings that may require modification: • In the `[Options]` section, the `Database Path` setting must point to the OCR pattern file, `<SilkTest directory>\SgOcrPattern.pat`. • The `[Recognition]` section contains settings that control which fonts are used to generate the pattern file.

The 4Test OCR Functions

This functionality is supported only if you are using the Classic Agent.

`OCR.inc` includes the following 4Test functions for OCR:

Function	Description	Parameters	Syntax
`OcrGetTextFromBmp`	Converts a bitmap file into text. The conversion uses the pre-configured pattern file, which is specified in the `Database Path`	**iRet** Result length. If conversion fails, then an E_OCR	`iRet = OcrGetTextFromBmp (sOcrText, sBitmapFile)`

Function	Description	Parameters		Syntax
	setting in the `textract.ini` file.		exception will be raised. INTEGER.	
		`sOcrText`	The result of converting the bitmap to text. NULL if conversion failed. OUT. STRING.	
		`sBitmapFile`	The bitmap (.bmp) file to convert. STRING.	
`OcrGetTextFromWnd`	Converts a bitmap of a window, or an area within a window, into text. The conversion uses the pre-configured pattern file, which is specified in the `Database Path` setting in the `textract.ini` file.	**iRet**	Result length. If conversion fails, then an E_OCR exception will be raised. INTEGER.	`iRet` = `OcrGetTextFromWnd` (`sText`, `wWindow`[, `rCapture`])
		`sText`	The result of converting a bitmap of the window to text. NULL if conversion failed. OUT. STRING.	
		`wWindow`	The window that will be the source of the bitmap to be converted to text. WINDOW.	
		`rCapture`	The capture region. OPTIONAL. RECT.	

Example

The sample test script (ocrtest.t) includes a test case (shown below) that extracts the text from a Microsoft Word document. Microsoft Office controls are recognized as custom windows (CustomWin) by Silk Test Classic, so you cannot use the 4Test `GetText()` method to get the text. However, you can use the `OcrGetTextFromWnd()` function to capture a bitmap of the document window and convert it to text. Notice that, if necessary, the test case will scroll through the document and capture multiple bitmaps.

```
testcase GetOcrAPIDocText (STRING sDocument)
  MSWord.SetActive ()

  // Open the specified document.
  MSWord.OpenDoc (sDocument)

  // Capture each page in succession.
  // Start at the top and page down until the bottom is reached
  LIST OF STRING lsResults = {}
  STRING sResult = NULL
  INTEGER iMaxPos, iCurPos, iLastPos = -1
  INTEGER iResLen, iTotalLen = 0

  withoptions
    BindAgentOption (OPT_REQUIRE_ACTIVE, FALSE)
    BindAgentOption (OPT_VERIFY_ACTIVE, FALSE)
    TheDoc.ScrollBarV.ScrollToMin ()
    iCurPos = TheDoc.ScrollBarV.GetPosition ()
    iMaxPos = TheDoc.ScrollBarV.GetRange ().iMax

  while TRUE
    // If we are capturing the first page, then eliminate the
flashing cursor
    // by highlighting the current character. Otherwise, page
down to capture
    // the next page.
    MSWord.SetActive ()
    if sResult == NULL
      // First page
      MSWord.TypeKeys ("<Shift-Right>")
    else
      // Page down
      withoptions
        BindAgentOption (OPT_REQUIRE_ACTIVE, FALSE)
        BindAgentOption (OPT_VERIFY_ACTIVE, FALSE)
        TheDoc.ScrollBarV.ScrollByPage (1)
        iLastPos = iCurPos
        iCurPos = TheDoc.ScrollBarV.GetPosition ()
      // If scrolling did not change the scrollbar position,
then
      // we have reached the bottom. Also, if we scrolled up
instead
      // of down by paging down, then we have reached the
bottom.
      if iCurPos <= iLastPos
        break

    // Convert the bitmap for the current view.
    iResLen = OcrGetTextFromWnd (sResult, TheDoc.CurrentView)
    if sResult != NULL
      ListAppend (lsResults, sResult)
      iTotalLen = iTotalLen + Len (sResult)
```

```
ResPrintList ("Document text ({iTotalLen} chars)", lsResults)
```

If neither of the 4Test functions in `OCR.inc` provides the functionality that you need, you can call the DLL functions in the Borland DLL, `SgOcrLib.dll`, through directly using the function declarations in `SgOcrLib.inc`. The DLL functions are documented at the top of `SgOcrLib.inc`. The functions in `OCR.inc` can serve as an example of how to use the DLL functions.

Instructions for Generating the Font Pattern Database

This functionality is supported only if you are using the Classic Agent.

Use the `Exgui.exe` utility to generate the pattern file `SgOcrPattern.pat`. The text conversion is performed using Windows fonts. Before conversion, the required fonts must be processed into the pattern file. Before generating the pattern file, the required fonts, font sizes, and font styles must be configured in the `Textract.ini` file. Open `Textract.ini` and adjust the following settings in the `[Recognition]` section:

Include1	List of fonts that are to be converted.
Exclude	List of fonts that are not to be converted.
Italic	1 - Convert italic characters; 0 - Exclude italic characters.
Bold	1 - Convert bold characters; 0 - Exclude bold characters.
Underlined	1 - Convert underlined characters; 0 - Exclude underlined characters.
Sizes	Range of font sizes, <min>-<max>, that are to be converted.

Once `Textract.ini` has been configured, open the `Exgui.exe` application and click **Build font pattern database**. When the **Textract - Build Font Base** dialog box opens, click **OK** and wait for the pattern file to be generated. The file name and path will be saved based on the `Database Path` setting in the `[Options]` section of the `textract.ini` file.

More Information about SGOCRLIB.DLL

This functionality is supported only if you are using the Classic Agent.

Copy the following files into the directory where the executable that calls the dll, for example `Partner.exe`, resides:

- `SgOcrLib.dll`
- `SgOcrPattern.pat`
- `Textract.dll`
- `Textract.ini`

For convenient pattern file generation, we recommend that you also copy the `exgui.exe` file to this directory.

Open the `Textract.ini` file and modify the `Database Path` setting in the `[Options]` section to point to the correct location of `SgOcrPattern.pat`.

Have the application, for example Silk Test Classic, load `SgOcrLib.dll`, which contains the functions explained in the following section.

Pattern File Generation

This functionality is supported only if you are using the Classic Agent.

Use the `exgui.exe` application to generate the appropriate pattern file. The text conversion is performed using Windows fonts. The required fonts must be processed into the pattern file before any conversion is

performed. The required fonts, font sizes, and font styles must first be configured in the `textract.ini` file. Open `textract.ini` and adjust the following settings:

Include1	List of fonts that are to be converted.
Exclude	List of fonts that are not to be converted.
Italic	1 - Convert italic characters; 0 - Exclude italic characters.
Bold	1 - Convert bold characters; 0 - Exclude bold characters.
Underlined	1 - Convert underlined characters; 0 - Exclude underlined characters.
Sizes	Range of font sizes, <min>-<max>, that are to be converted.

Then open the `exgui.exe` application and click **Build font pattern database**. Click **OK** and wait for the pattern file to be generated. The file name and path will be saved based on the `Database Path` setting in the `textract.ini` file.

Supporting Internationalized Objects

This section describes how you can work with internationalized objects.

Overview of Silk Test Classic Support of Unicode Content

Silk Test Classic is Unicode-enabled, which means that Silk Test Classic is able to recognize double-byte (wide) languages. We have enabled components within the application to deal with Unicode content. The Silk Test Classic GUI supports the display and input of wide text. The 4Test language processor has been enhanced to support wide text. All 4Test library functions have been widened. The extensions have been enhanced to support the input and output of wide text.

We have added and modified 4Test functions to deal with internationalization issues. With Silk Test Classic you can test applications that contain content in double-byte languages such as Chinese, Korean, or Japanese (Kanji) characters, or any combination of these. You can also name Silk Test Classic files using internationalized characters. Silk Test Classic supports three text file formats: ANSI, Unicode and UTF-8.

Silk Test Classic supports the following:

- Localized versions of Windows.
- International keyboards and native language Input Method Editors (IME).
- Passing international strings as parameters to test cases, methods, and so on, and comparing strings.
- Accessing databases through direct ODBC standard access.
- Reading and writing text files in multiple formats: ANSI, Unicode, and UTF-8.

For information about new features, supported platforms and versions, known issues, and work-arounds, refer to the *Release Notes*.

Before testing double-byte characters with Silk Test Classic

Testing an internationalized application, particularly one that contains double-byte characters, is more complicated than testing an application that contains strictly English single-byte characters. Testing an internationalized application requires that you understand a variety of issues, from operating system support, to language packs, to fonts, to working with IMEs and complex languages.

Before you begin testing your application using Silk Test Classic, you must do the following:

- Meet the needs of your application under test (AUT) for any necessary localized OS, regional settings, and required language packs.

- Install the fonts necessary to display your AUT.
- If you are testing an application that requires an IME for data input, install the appropriate IME.

Using DB Tester with Unicode Content

To use DB Tester with Unicode characters:

- You must have a Unicode-capable driver (ODBC version 3.5 or higher) associated with the data source name you are using in your test plan.
- The database must be Unicode capable (SQL Server 7 and 2000, Oracle 8 and higher).

Issues Displaying Double-Byte Characters

When you are dealing with internationalized content, being able to display the content of your application is critical. Carefully consider the following:

Operating system	Your operating system needs to be capable of displaying double-byte characters in the system dialog boxes and menus used by your application.
Silk Test Classic	You need to be concerned about displaying your content in the Silk Test Editor and the Silk Test Classic dialog boxes.
Application under test	You need to have a font installed that is capable of displaying the content of your application. If you have multiple languages represented in your application, you will need a font that spans these languages.
Browser	If your application is web-based, make sure that you are using a browser that supports your content, that the browser is configured to display your content, and that you have the necessary fonts installed to display your application.
Complex scripts (languages)	Silk Test Classic does not support complex scripts such as the bi-directional languages Hebrew and Arabic. These are languages that require special processing to display and edit because the characters are not laid out in a simple linear progression from left to right, as are most western European characters.

Learning More About Internationalization

There are a variety of online sites that provide general information about internationalization issues. You may find the following Web sites useful if you are learning about internationalization, localization or Unicode. They include:

- Microsoft's Professional Developer's Site for Software Globalization Information (*http://www.microsoft.com/globaldev/default.asp*)
- The definitive word on the W3C's Web site (*http://www.w3.org/international*)
- The Unicode Consortium, a non-profit organization founded to develop, extend and promote use of the Unicode Standard (*http://www.unicode.org*)
- IBM's International Components for Unicode (*http://oss.software.ibm.com/icu/userguide/index.html*)
- A tutorial from Sun on how to internationalize Java applications (*http://java.sun.com/docs/books/tutorial/i18n*)

Silk Test Classic File Formats

Silk Test Classic gives you the ability to specify the file format of text files and .ini files. Before Silk Test Classic 5.5, all files were in the ANSI file format. You can create the following formats:

ANSI	For Silk Test Classic purposes, ANSI is defined as the Microsoft Windows ANSI character set from Code Page 1252 Windows Latin 1.

Unicode Is an extended form of ASCII that provides the ability to represent data in a uniform plaintext format that can be sorted and processed efficiently. Unicode encompasses nearly all characters used in computers today.

UTF-8 Unicode Transformation Format (UTF) Is a multi-byte encoding that can handle all Unicode characters. It is used to compress Unicode, minimize storage consumption and maximize transmission.

You have the ability to save text files in any of three file formats: ANSI, UTF-8, and Unicode. By default all files are saved in UTF-8 format. The **Save As** dialog boxes throughout include a list box from which you can select the file format in which you want to save your file.

- ANSI files cannot contain non ANSI characters
- The file formats available will depend on the content of your text file. If your file contains characters not available on code page 1252, ANSI will not display in the list box. If you are working with an existing ANSI file and add non-ANSI characters, the **Save As** dialog box will open when you attempt to save the file. In order to save the changes you will need to change the file format and click **Save**.
- The title bar indicates the file format: When you have a file open, the format of that file is indicated on the title bar.
- Silk Test Classic uses the Microsoft standard Byte Order Marked (BOM) to determine the file type for UTF-8 and Unicode files. If a Unicode file does not have the BOM marker then Silk Test Classic sees the file as an ANSI file, and the Unicode characters cannot be displayed.

Reusing Silk Test Classic Single-Byte Files as Double-Byte

If you have existing single-byte Silk Test Classic text files, such as `*.pln`, `*.inc` or `*.t`, that you want to use in double-byte testing, the files must:

- Be compatible with Silk Test Classic, such as files created using the IE 5.x DOM extension for testing a Web application.
- Be recompiled in Silk Test Classic because the object files, `*.ino` and `*.to`, are not compatible.

Opening an existing Silk Test Classic file as a double-byte file

Choose one of the following:

- Copy the file you want to re-use to a new directory. Do not copy the associated object (`*.ino` or `*.to`) files. In Silk Test Classic, open this new file.
- In the existing directory, delete the object files associated with the file you want to re-use. In Silk Test Classic, open the desired file.

When the Silk Test Classic file is compiled, new objects files are created. If you enter double-byte content into the file, when you try to close the file you will be prompted to save the file in a compatible file format, Unicode or UTF-8.

Specifying File Formats for Existing Files with Unicode Content

If you want to save an existing file in a different file format, choose one of the following:

Overwriting the file

If the file is already referenced from other files, you may want to change the format without changing the name or its location. As you cannot have two files with the same name saved in the same directory, even in different formats, the only option is to overwrite the file.

1. Make sure the file is the active window. Click **File** > **Save As** and select the file from the list.
2. From the **Save as format** list box, select the file format. ANSI is not available if the file contains characters outside of the ANSI character set.
3. Click **Save**. A dialog box displays asking if you want to overwrite the file.

4. Click **Yes**.

Saving in the same directory

If you want to have versions of a file in various formats within the same directory, you must save each file with a different name.

1. Make sure the file is the active window. Click **File** > **Save As**.
2. In the **File name** text box, enter the new name of the file.
3. From the **Save as format** list box, select the file format. ANSI is not available if the file contains characters outside of the ANSI character set.
4. Click **Save**.

Saving in a different directory

If you would like to keep the name of the file but change the format, you must save the file in a different directory.

1. Make sure the file is the active window. Click **File** > **Save As** and select the file from the list.
2. Navigate to the directory in which you want to save the file.
3. From the **Save as format** list box, select the file format. ANSI is not available if the file contains characters outside of the ANSI character set.
4. Click **Save**.

If you modify an ANSI text file and the modifications include characters outside of the ANSI characters set, when you try to save your changes, the Save As dialog box will open and you need to either overwrite the ANSI file with a file of the same name but in a different format, or rename the file and save in Unicode or UTF-8 format .

Specifying File Formats for New Files with Unicode content

This topic contains instructions on specifying the file format for:

With the exception of test frames, to specify the file format of a new file:

1. Click **File** > **New**.
2. On the **New** dialog box, select the file type.
3. Click **OK**. The untitled file opens.
4. Click **File** > **Save As**. The **Save As** dialog box opens.
5. Navigate to where you want to store the file and enter the name of the file in the **File name** text box.
6. Select a file format (UTF-8 is the default) from the **Save as format** list box. ANSI is not available if the file contains characters outside of the ANSI character set.
7. Click **Save**.

To specify the file format for a new test frame:

1. Click **File** > **New**.
2. On the **New** dialog box, select the file type **Test Frame** and click **OK**. The **New Test Frame** dialog box opens.
3. To select a file format, click **Browse**. The **Save As** dialog box opens. The default file format for test frames is UTF-8. If you simply type the path and file name in the **File name** text box of the **New Test Frame** dialog box and click **OK**, the file is saved in UTF-8.
4. Navigate to where you want to store the file and enter the name the file in the **File name** text box.
5. Select the file format from the **Save as format** list box. If you select ANSI and if the file contains characters outside of the ANSI character set, when you try to save the file you will need to change the file format to a compatible format, Unicode or UTF-8.
6. Click **Save**. The **New Test Frame** dialog box regains focus.

7. On the **New Test Frame** dialog box, select the application and proceed as normal.

If you modify an ANSI text file and the modifications include characters outside of the ANSI characters set, when you try to save your changes, the **Save As** dialog box will open and you need to either overwrite the ANSI file with a file of the same name but in a different format, or rename the file and save in Unicode or UTF-8 format .

Working with Bi-Directional Languages

Silk Test Classic supports bi-directional languages to the extent that the operating system does. Silk Test Classic captures static text in all Unicode languages. However, scripting, playback and many string functions are not fully supported for complex languages, the most common of these being the bi-directional languages Hebrew and Arabic. The problems you may encounter are discussed below.

Silk Test Classic with bi-directional languages on Windows XP

Windows XP is a multi-lingual operating system and is capable of handling bi-directional languages when configured properly.

On Windows XP if you input characters from RIGHT to LEFT (`CBA `) provided that the default system locale is set for a bi-directional language, Silk Test Classic will correctly record and playback the characters as they were entered and display, from RIGHT to LEFT. When you use a 4Test string function such as `StrPos` (string position) to return the third element, 4Test correctly counts from right to left and returns `"C"`

Once you have set a default system locale, the operating system continues to be able to read and write that language properly, even after another locale has been set as the default. This works only if the language is not unchecked from the **Language Settings** area after another default is set. Once a language is unchecked, the ability to read and write in that language will be gone when you reboot your system. You would need to reset it as the default to restore the capability.

Recording Identifiers for International Applications

This functionality is supported only if you are using the Classic Agent.

By default, Silk Test Classic records an object's caption text as its identifier (ID). With Silk Test Classic you can override this default and specify ASCII-only IDs, where the ID is based on the object's class and index. This helps to automatically declare English IDs while keeping the tags native.

The identifier has the form ClassnameIndex, where Classname is the 4Test class of the object and Index is an internally generated integer that ensures that identifiers within a window are unique.

To select ASCII-only identifiers:

1. On the **Record Window Declarations** dialog box, click **Options**. The **Record Window Declarations Options** dialog box opens.
2. In the **Window declaration identifiers** area, click **Use the 4Test Class**.
3. Click **OK**. ASCII-only IDs will now be captured when you record new window declarations.

Configuring Your Environment

This section describes how you can configure your environment for internationalized objects.

Configuring Your Microsoft Windows XP PC for Unicode Content

If you have already configured your Windows XP PC to run your internationalized application, you may be able to disregard this topic and see *Recording Identifiers for International Applications*.

On Microsoft Windows XP you may need to do all or some of the following:

- Install language support required by your application through modifications in the **Regional and Language Options** dialog box.
- If your application contains content that is in a large-character-set language, such as simplified Chinese, you may need to install an Input Method Editor (IME) if you want to input data in this language. For additional information about IMEs, refer to the Microsoft support site.

Fonts

To display the content of your application in Silk Test Classic you will need to have an appropriate font installed and specify this font in the system registry and in the Silk Test Classic Options/Editor Font.

Installing Language Support

You must have administrator privileges to install language packs or set the system default locale.

Microsoft Windows XP provides built-in support for many double-byte languages. Enabling this support can be done at the time of install or after setup through the **Regional and Language Options** dialog box. If you enable language support after setup, you may need files from the Microsoft Windows XP installation CD. Configurations will vary depending on your needs and how your system has been configured previously. The following instructions are intended only to be general information to get you started:

1. Click **Start** > **Settings** > **Control Panel** > **Regional and Language Options**.
2. If you are testing East Asian languages, select the **Languages** tab, and then check the **Install files for East Asian languages** check box.

   You may be prompted to insert the Microsoft Windows XP CD for the necessary files.
3. Click the **Advanced** tab on the **Regional and Language Options** dialog box.
4. Select the language that matches the language of the non-Unicode programs you want to use.

   For example Chinese (PRC).
5. Click **OK**.
6. Reboot your computer for the changes to take effect.

   After you restart your computer, if you want to input data in a language other than the default language, you must click the Language bar icon in your system tray and select the language from the multi-lingual indicator.

Setting Up Your Input Method Editor

If you want to use an Input Method Editor (IME) to input data in the language you selected, you may need to set up your IME.

1. Click **Start** > **Settings** > **Control Panel** > **Regional and Language Options**.
2. Click the **Languages** tab.
3. Click **Details** in the **Text Services and Input Language** area.
4. In the **Settings** tab on the **Text Services and Input Language** dialog box, select the language you want to use as your default input language.
5. In the **Preferences** section of the **Settings** tab, click **Language Bar**, make sure the **Show the Language Bar on the desktop** check box is checked, and then click **OK** on the **Settings** tab.

   This default will enable your system to display this language in dialog boxes and menus. We recommend setting the default to the language of the AUT.

Displaying Double-Byte Characters

While Silk Test Classic can process Unicode, displaying double-byte characters is not automatic. Keep the following in mind:

- Is your operating system configured to display your content?

- Is Silk Test Classic configured to display double-byte content in its dialog boxes?
- Do you have the right font set to display your content in the Editor?

Displaying Double-Byte Characters in Dialog Boxes

If Silk Test Classic is rendering squares or pipes in dialog boxes where you expect double-byte characters, you may need to make a simple modification to Silk Test Classic using a script we have provided. This script is located in `<SilkTest Installation directory>\Tools`.

1. In Silk Test Classic, click **File** > **Open**.
2. In the `Tools` directory, open `font.t`.
3. Click **Run** > **Testcase**. The **Run Testcase** dialog box opens.
4. In the **arguments** area, type the name of the font in quotes.

   For example, `Arial Unicode MS`. It is not necessary to include the type of font, for example `Arial Unicode MS (True Type)`.
5. Click **Run**.
6. Reboot your computer for the changes to take effect.

Displaying Double-Byte Characters in the Editor

In order for the Editor to display double-byte characters, such as those captured in your test frame, you must select a font that is able to display these characters.

1. In Silk Test Classic, click **Options** > **Editor Font**.
2. From the available fonts, select one that is able to display the language of your application.

If your application contains multiple languages, make sure that you have a font installed that is capable of rendering all the languages, as the Editor does not display multiple fonts. Licensed Microsoft Office 2000 users can freely download the Arial Unicode MS font from Microsoft.

Localized Browser Support

This functionality is supported only if you are using the Classic Agent.

Silk Test Classic provides support for several internationalized versions of Internet Explorer and Mozilla Firefox; localized browser include files are provided.

For information about new features, supported platforms and versions, known issues, and work-arounds, refer to the *Release Notes*.

The following files, in ANSI format, are available in the `<SilkTest Installation directory>\Locale`:

- `...\French\browser.inc`
- `...\French\explorer.inc`
- `...\French\firefox.inc`
- `...\German\browser.inc`
- `...\German\explorer.inc`
- `...\German\firefox.inc`

Silk Test Classic also contains the following files in UTF-8 format in the `<SilkTest Installation directory>\Locale`:

- `...\japanese\browser.inc`
- `...\japanese\explorer.inc`
- `...\simplified_chinese\browser.inc`
- `...\simplified_chinese\explorer.inc`

- `...\simplified_chinese\Firefox.inc`

Changing the Default Browser Include Files

This functionality is supported only if you are using the Classic Agent.

To change the default English-US browser include files to one of the supported localized browsers:

1. Navigate to `<SilkTest Installation directory>\Locale`.
2. In the `Locale` directory, locate the language of the localized browser you are using.
3. Copy the files contained in the `<Language directory>` directory.
4. Go to `<SilkTest Installation directory>\Extend` and paste the files, overwriting the existing files.

Resetting Browser Support to Default

This functionality is supported only if you are using the Classic Agent.

To reset support for the English-US browsers to the defaults when you no longer want to test a localized version of the browser:

1. Navigate to `<SilkTest Installation directory>\Locale`.
2. Copy the files from within the US directory.
3. Navigate to `<SilkTest Installation directory>\Extend` and paste the files, overwriting the existing files.

Note: Use statements cannot be used to swap browser include files. You must overwrite the files within the `Extend` directory.

Using an IME with Silk Test Classic

Silk Test Classic supports IMEs. The IME is enabled only after you have installed an Asian language package. The IME will work once you have installed it, enabled it, and are in an application with IME support. In Silk Test Classic, the IME is only available when a file, such as an include or script, is active.

For additional information about IMEs and for downloads, see the Microsoft support site.

Troubleshooting Unicode Content

This section contains topics to help troubleshoot unicode content.

Display Issues

This section describes how you can troubleshoot display issues in Unicode contents.

Why Are My Window Declarations Recording Only Pipes?

If your window declarations record only pipes (|), You've probably forgotten to set the **Options** > **Font Editor** to a font that can display the language of your AUT.

What Are Pipes and Squares Anyway?

The pipes and squares, or even question marks (?), display in place of characters which the system has not yet been configured to display. A font that does not support the language is being used in the dialog boxes and menus. Whether or not you see pipes or squares depends on what font is used and what language you are trying to display.

Why Can I Only Enter Pipes Into a Silk Test Classic File?

If you can only enter pipes into a file, for example a frame file or an include file, the Silk Test Classic Editor font is not set to display the language of your AUT.

Why Do I See Pipes and Squares in the Project Tab?

Pipes, squares, and questions marks (?) display in place of characters which the system has not yet been configured to display. A font that does not support the language is being used in the dialog boxes and menus. Whether or not you see pipes or squares depends on what font is used and what language you are trying to display.

You must configure your system and make sure that you have set the regional settings.

Why Cannot My System Dialog Boxes Display Multiple Languages?

If you are testing an application whose content contains multiple languages, meaning that it has several character sets represented, you may need to:

- Make sure that you have a font installed on your machine that can display all the languages.
- Configure Silk Test Classic to use a font that can display your content.

Why Do I See Pipes and Squares in My Win32 AUT?

If you start up your application under test and see pipes and squares in the title bar, menus, or dialog boxes, it may mean that the operating system cannot support your application or that your system is not properly configured to display your content.

Why Do the Fonts on My System Look so Different?

Fonts that display in your menus, title bars and so on, are controlled by the registry settings and the **Display Properties** > **Appearance** settings of your computer.

If your fonts display too large or too small, you may have incorrectly set the appearance for an item:

1. Navigate to **Start** > **Settings** > **Control Panel** > **Display.**
2. Navigate to the **Appearance** tab and select **Windows** standard in the **Scheme** field.
3. Click **OK**.

Your desktop should now display normal.

Why Do Unicode Characters Not Display in the Silk Test Project Explorer

To view Unicode characters in the Silk Test Project Explorer, you must have installed a language pack with Unicode characters.

Why Is My Web Application Not Displaying Characters Properly?

If your Web application is not displaying the characters properly, or strange symbols or character are mixed in with your content, you may need to change a setting in your browser.

Internet Explorer Users

Check the settings for Encoding:

1. In Internet Explorer, click **View** > **Encoding**.
2. Select one of the following:
   - From the listed encodings, select one that meets the requirements of your application.
   - Click **More**, then select an encoding that meets the requirements of your application.
   - Click **Auto-Select**.

Mozilla Firefox Users

Check the settings for Character Coding:

1. In Mozilla Firefox, click **Settings** > **Content**.
2. In the **Fonts & Colors** section, click **Advanced**.
3. Select a character coding that meets the requirements of your application.

If you still have problems, ensure that your system locale is set for the language of your application under test.

File Formats

This section describes how you can troubleshoot issues with file formats in Unicode contents.

Considerations for VB/ActiveX Applications

This functionality is supported only if you are using the Classic Agent.

If your VB/ActiveX application uses richtextbox controls, you need to base that control on `riched32.dll` version 5.00.213.4.1 which supports Unicode text characters. Before you begin using Silk Test Classic with Unicode content, check to make sure you do not have the older `riched32.dll` version (4.00.993.4). To do this:

1. Using Windows Explorer, locate the `riched32.dll` on your system.
2. Right-click the file and select **Properties**.
3. In the **riched32.dll Properties** dialog box, click the **Version** tab. Make sure you have version 5.00.213.4.1.

Why Am I Getting Compile Errors?

You may be trying to compile a file with an incompatible file format. Silk Test Classic supports three file formats: ANSI, UTF-8, and Unicode. If you try to compile files in Silk Test Classic that are in other formats, such as DBCS, you will get compile errors.

Workaround: In a Unicode-enabled text editor, save the file in one of the Silk Test Classic supported file formats: ANSI, UTF-8 or Unicode.

Why Does Silk Test Classic Open Up the Save As Dialog Box when I Try to Save an Existing File?

You have likely added content to the file that is incompatible with the file's existing file format. For example, you could have added Japanese characters to a frame file that was previously saved in ANSI format.

You must save the existing file in a compatible format.

Working with Input Method Editors

This section describes how you can troubleshoot issues when working with Input Method Editors (IMEs).

Why is English the Only Language Listed when I Click the Language Bar Icon?

You must be running an application, or area within the application, that supports an IME for a language other than English to be displayed in the Language bar icon. Applications that support IME include elements of Silk Test Classic such as include files and script files, Outlook, and Internet Explorer.

Why Does This IME Look so Different from Other IMEs I Have Used

IMEs can look different, depending on the operating system you are using and the particular IME you have accessed. For more information about IMEs, see Microsoft's support site.

Using Autocomplete

This section describes how you can automatically complete functions, members, application states, and data types.

Overview of AutoComplete

AutoComplete makes it easier to work with 4Test, significantly reducing scripting errors and decreasing the need to type text into your 4Test files by automatically completing functions, members, application states, and data types. There are four AutoComplete options:

Option	Description
Function Tip	Provides the function signature in a tooltip.
MemberList	Displays window children, properties, methods, and variables available to your 4Test file.
AppStateList	Displays a list of the currently defined application states.
DataTypeList	Displays a list of built-in and user-defined data types.

AutoComplete works with both Silk Test Classic-defined and user-defined 4Test files.

If you create a new 4Test file, you must name and save it as either a .t , .g.t, or .inc file in order for AutoComplete to work. After a 4Test file is saved, AutoComplete recognizes any changes you make to this file in the 4Test Editor and includes files that you reference through a 4Test use statement or the **Use Files** text box on the **Runtime Options** dialog box. When working with an existing 4Test file, you do not need to save or compile in order to access newly defined functions, methods, or members.

AutoComplete only works with 4Test files, which are .t, .g.t, and .inc files, that use hierarchical object recognition or dynamic object recognition with locator keywords.

AutoComplete does not work on comment lines or within plan, suite, or text files. AutoComplete does not support global variables of type window. However, AutoComplete supports Unicode content.

AutoComplete does not distinguish between Silk Test Classic Agents. As a result, AutoComplete displays all methods, properties, variables, and data types regardless of the Silk Test Classic Agent that you are using. For example, if you are using the Open Agent, functions and data types that work only with the Classic Agent are also displayed when you use AutoComplete. For details about which methods are supported for each Silk Test Classic Agent, review the corresponding .inc file, such as the `winclass.inc` file.

Customizing your MemberList

The members that you see in the MemberList depend on the MemberList options that you select. You can specify which members display in your MemberList. The members are window children, methods, properties, and variables. You can also determine how much detail is displayed in the MemberList by specifying the inheritance level and deciding whether you want to view class, data type, and function return type for methods in your MemberList.

All member options are enabled by default and the default inheritance level is below `AnyWin` class, meaning that methods for any class derived from the `AnyWin` class display in the MemberList. For additional information about the inheritance level, see the *General Options Dialog Box*.

Note: Methods that are defined in and above the `AnyWin` class, such as `Click` and `Exist`, which are defined in the `Winclass`, will not display in the MemberList. You can type these methods into your script, but they will not display in the MemberList unless you change the inheritance level to `All`.

To customize your MemberList:

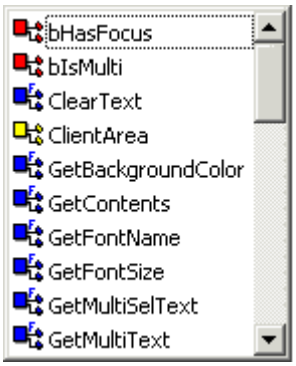
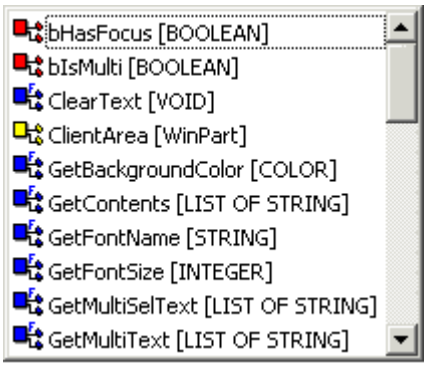
1. Open Silk Test Classic and choose **Options** > **General**.
2. In the **AutoComplete** area of the **General Options** dialog box, make sure MemberList is selected.
3. In the **MemberList Options** area, select the members that you want to display in your MemberList. For example, if you want to view only properties and variables, uncheck the **Methods** and **Window Children** check boxes.
4. Select the appropriate Inheritance Level for the selected methods.

   You can choose one of the following:

Below AnyWin Class	Displays methods for any class derived from the `AnyWin` class. **Below AnyWin Class** is the default.
All	Displays the complete inheritance for members all the way up through `AnyWin` and the control classes, including the `Winclass`.
None	Displays only those members defined in the class of the current object and window declaration.

5. If you want to view attributes for the selected members, such as the class for window children, the data type for properties and variables, and the return type for method functions in your MemberList, check the **Member Type** check box.

   Member Type is not checked by default. The following is a sample MemberList with and without **Member Type** checked.

Default MemberList	MemberList with Member Type Selected
	

6. Click **OK** on the **General Options** dialog box to save your changes.

Frequently Asked Questions about AutoComplete

Why isn't AutoComplete working?

AutoComplete only works with 4Test files with extension .t, .g.t, and .inc. If (untitled) is displayed in the title bar of your 4Test file, the file has not been saved yet. Save the file as .t, .g.t, or .inc.

After a 4Test file is saved, AutoComplete recognizes any changes you make to this file in the 4Test Editor and include files that you reference through a 4Test use statement or the **Use Files** text box on the **SilkTest Runtime Options** dialog box. Once you save a new file as a .t, .g.t, or .inc, you do not need to save or compile in order to access newly defined functions, methods, or members.

AutoComplete does not work on comment lines or within plan, suite, or text files.

Why doesn't a member display in my MemberList?

There a few reasons you may not see a member in your MemberList. Here's what you should do:

1. On the **General Options** dialog box, make sure that you chose to show members of this type in the **MemberList Options** section. For additional information, see *Customizing your MemberList*.

2. Make sure the member you want to see is included in the inheritance level you selected. Below `AnyWin` class is the default; you might need to change your inheritance level to `All`. For additional information, see *Customizing your MemberList*.
3. Name and save your file with a .t, .g.t, or .inc extension.
4. Compile your file and fix any scripting errors. Anything following a compile error is not displayed in the MemberList or FunctionTip.

What happens if there is a syntax error in the current file?

Everything, based on the AutoComplete options you have selected, prior to the syntax error will display in your MemberList and/or FunctionTip. Anything following the syntax error will not display in your MemberList and/or FunctionTip. For additional information, see *Customizing your MemberList*.

What if I type something and AutoComplete does not find a match?

AutoComplete might not find a match for a number of reasons, for example because of the AutoComplete options you have specified or because of a compile error in your file. For information about fixing some of these issues, see *Customizing your MemberList* and *Turning AutoComplete Options Off*.

When AutoComplete does not find a match in the MemberList, focus remains on the first item in the list.

Note: If you perform any of the selection methods, which means if you press Return, Tab, or click, the item will be pasted to the Editor.

You can simply type any function, method, or member in your 4Test files; AutoComplete does not restrict you in any way from typing in 4Test files.

Note: You must dismiss the MemberList or FunctionTip before you can type in the Editor.

If you plan to use AutoComplete extensively, we recommend that you rename your identifiers in your window declarations. Knowing your identifier names helps, especially when working with long lists.

Why doesn't list of record type display in the FunctionTip?

This is a known limitation. FunctionTip does not support list of record types.

Why does AutoComplete show methods that are not valid for a 4Test class?

When using AutoComplete, the member list occasionally may reveal methods that are not valid for the 4Test class. The compiler will not catch these usage problems, but at Runtime the following exception is raised when the script is played back: `Error: Function <invalid method> is not defined for <window class>.`

Why does AutoComplete show methods, properties, variables, and data types that are not supported for the Silk Test Agent that I am using?

AutoComplete does not distinguish between Silk Test Agents. As a result, AutoComplete displays all methods, properties, variables, and data types regardless of the Silk Test Agent that you are using. For example, if you are using the Open Agent, functions and data types that work only with the Classic Agent are also displayed when you use AutoComplete. For detailed information about which methods are supported for each Silk Test Agent, review the corresponding .inc file, such as the `winclass.inc` file.

Turning AutoComplete Options Off

This topic contains instructions on how to disable AppStateList, DataTypeList, FunctionTip, and MemberList.

To turn off AutoComplete options:

1. Open Silk Test Classic and click **Options** > **General**.
2. In the **AutoComplete** area of the **General Options** dialog box, uncheck the check box for each of the AutoComplete options that you want to disable, and then click **OK**.

Using AppStateList

To display a list of currently defined application states:

1. Within your script, .t or .g.t, or within the include file, type your test case declaration, followed by the keyword appstate and then press **space**.

   For example `testcase foo ( ) appstate` .

   A list of currently defined application states displays. You can also type the keyword basedon followed by a **space**. For example `appstate MyAppState () basedon` .
2. Use one of the following methods to select the appropriate member and paste it to the Editor.

   - Type the first letter or the first few letters of the member and then press **Enter** or **Tab**.
   - Use your arrow keys to locate the member and then press **Enter** or **Tab**.
   - Scroll through the list and click on a member to select it.

Using DataTypeList

To display a list of built-in and user-defined data types:

1. Within your script, .t or .g.t, or include file, type `array` or `varargs`, as appropriate, followed by the of keyword and a **space**.

   For example, `list of`.

   The current list of built-in and user-defined data types appears. You can also view the list of data types by pressing **F11**.
2. Use one of the following methods to select the appropriate member and paste it to the Editor:

   - Type the first letter or the first few letters of the member and then press **Enter** or **Tab**.
   - Use your arrow keys to locate the member and then press **Enter** or **Tab**.
   - Scroll through the list and click on a member to select it.

Using FunctionTip

To display the function signature for a function, test case, or method.

1. Within your script, .t or .g.t, or include file, type the function, test case, or method name, followed by an open parenthesis " (".

   For example `SetUpMachine(`. The function signature displays in a tooltip with the first argument, if any, in bold text. The function signature includes the return argument type, pass-mode, data type, name of the argument, and null and optional attributes, as they are defined.
2. Type the argument.

   The FunctionTip containing the function signature remains on top and highlights the argument you are expected to enter in bold text. As you enter each argument and then type a comma, the next argument that you are expected to type is highlighted. The expected argument is always indicated with bold text; if you backspace or delete an argument within your function, the expected argument is updated accordingly in the FunctionTip. The FunctionTip disappears when you type the close parenthesis ") " to complete the function call.

If you want to dismiss the FunctionTip, press **Escape**. FunctionTip is enabled by default. See *Turning AutoComplete Options Off* if you want to disable FunctionTip.

Using MemberList

This topic contains instructions on how to use MemberList to view and select a list of members.

To view a list of members:

1. Customize the member list so that it displays the information you require.

   You can choose to display any or all of the following members:

Member	Description
Window children	Displays all window objects of type WINDOW that are defined in window declarations in the referenced .t, .g.t, and .inc files. Indicated in the MemberList with a yellow icon.
Methods	Displays all methods defined in the referenced .t, .g.t, and .inc files. Indicated in the MemberList with a blue icon.
Properties	Displays all properties defined in the referenced .t, .g.t, and .inc files. Indicated in the MemberList with a red icon.
Variables	Displays all defined variables in the referenced .t, .g.t, and .inc files, including native data types, data, and records. Fields defined for records and nested records also display in the list. Indicated in the MemberList with a red icon.

2. Within your script or include file, type the member name and then type a period (.).

   For example `Find.`.

   The MemberList displays. Depending on the MemberList Options and the Inheritance Level you select, the types of members that display in the MemberList will vary.
3. Use one of the following methods to select the appropriate member and paste it to the Editor:

   - Type the first letter or the first few letters of the member and then press **Enter** or **Tab**.
   - Use your arrow keys to locate the member and then press **Enter** or **Tab**.
   - Scroll through the list and click on a member to select it.

The MemberList is case sensitive. If you type the correct case of the member, it is automatically highlighted in the MemberList; press **Enter** or **Tab** once to paste it to the Editor. If you do not type the correct case, the member has focus, but is not highlighted; press **Enter** or **Tab** twice to select the member and paste it to the Editor. To dismiss the MemberList, press **Escape**.

Overview of the Library Browser

Click **Help** > **Library Browser** to access the **Library Browser**. It provides online documentation for:

- Built-in 4Test methods, properties, and functions: the **Library Browser** shows the name and class of the method, one line of descriptive text, syntax, and a list of parameters, including a description.
- User-defined methods: the **Library Browser** shows the name and class of the method, syntax, and a list of parameters. It displays User defined as the method description and displays the data type for each parameter.
- User-defined Properties: As with user-defined methods, the description for user-defined properties by default is User defined.

The **Library Browser** does not, by default, provide documentation for your user-defined functions. You can add to the contents of the **Library Browser** to provide descriptive text for your user-defined methods, properties, and functions.

Library Browser Source File

The core contents of the **Library Browser** are based on a standard Silk Test Classic text file, `4test.txt`, which contains information for the built-in methods, properties, and functions.

You can edit `4test.txt` to include your user-defined information, or define your site-specific information in one or more separate files, and then have Silk Test Classic compile the file (creating `4test.hlp`) to make it available to the **Library Browser**. Information about methods in `4test.hlp` is also used in the **Verify Window** dialog box for methods.

Silk Test Classic does not update `4test.txt` with user-defined information; instead it populates the **Library Browser** from information it receives when include files are compiled in memory. You modify `4test.txt` to override the default information displayed for user-defined objects.

Simply looking through `4test.txt` should give you all the help you need about how to structure the information in the file. The following table lists all the keywords and describes how they are used in 4test.txt. You should edit a copy of `4test.txt` to add the information you want.

Keywords

Keywords are followed by a colon and one or more spaces.

class	Name of the class.
function	Name of the function. Specify the full syntax. If the function returns a value, specify: `return_value = function_name (parameters)` Otherwise, specify: `function_name (parameters)`
group	Name of the function category.
method	Description of the method. Specify the full syntax. If the method returns a value, specify: `return_value = method_name (parameters)` Otherwise, specify: `method_name (parameters)`
notes	Description of the method, property, or function, up to 240 characters. Do not split the description into multiple **notes** fields, since only the first one is displayed.
parameter	Name and description of a method or function parameter. Each parameter is listed on its own line. Specify the name, followed by a colon, followed by the description of the parameter.
property	Name of the property.
returns	Type and description of the return value of the method or function. Specify the name, followed by a colon, followed by the description of the return value.
#	Comment.

Adding Information to the Library Browser

1. Make a backup copy of the default `4test.txt file`, which is in the directory where you installed Silk Test Classic, and store your backup copy in a different directory.
2. In an ASCII text editor, open `4test.txt` in your Silk Test Classic installation directory and edit the file. See examples for methods, properties, and functions, if necessary.

3. Quit Silk Test Classic.
4. Place your modified `4test.txt` file in the Silk Test Classic installation directory.
5. Restart Silk Test Classic. Silk Test Classic sees that your source file is more recent than `4test.hlp` and automatically compiles `4test.txt`, creating an updated `4test.hlp`. If there are errors, Silk Test Classic opens a window listing them and continues to use the previous `4test.hlp` file for the Library Browser. If there were errors, fix them in `4test.txt` and restart Silk Test Classic. Your new definitions are displayed in the **Library Browser** (assuming that the files containing the declarations for your custom classes, methods, properties, and functions are loaded in memory).

There is another approach to updating the **Library Browser**: maintain information in different source files.

If the **Library Browser** isn't displaying your user-defined objects, close the **Library Browser**, recompile the include files that contain your user-defined objects, then reopen the **Library Browser**.

Add User-Defined Files to the Library Browser with Silk Test Classic

1. Create a text file that includes information for all your custom classes and functions, using the formats described in the **Library Browser** source file. If you have added methods or properties to built-in classes, you should add that information in the appropriate places in `4test.txt`, as described above. Only document your custom classes and functions in your own help file.
2. Click **Options** > **General** and add your help file to the list in the **Help Files For Library Browser** field. Separate the files in this list with commas.
3. Click **OK**. Silk Test Classic recompiles `4test.hlp` to include the information in all the files listed in the **Help Files For Library Browser** field. If there are errors, Silk Test Classic opens a window listing them and continues to use the previous `4test.hlp` file for the **Library Browser**. If you had errors, fix them in your source file, then quit and restart Silk Test Classic. Silk Test Classic recompiles `4test.hlp` using your modified source file.

Viewing Functions in the Library Browser

To view information about built-in 4Test functions in the **Library Browser**:

1. Click **Help** > **Library Browser**, and then click the **Functions** tab.
2. Select the category of functions you want in the **Groups** list box. To see all built-in 4Test functions, check the **Include all** check box.

   Functions are listed in the **Functions** list box.
3. Select the function for which you want information.

Viewing Methods for a Class in the Library Browser

4Test classes have methods and properties. When you select the **Methods** or **Properties** tabs in the **Library Browser**, you see a list of all the built-in and user-defined classes in hierarchical form.

To see the methods or properties for a class:

1. Click **Help** > **Library Browser**, and then click the **Methods** or **Properties** tab.
2. Select the class in the **Classes** list box.

   Double-click a + box to expand the hierarchy. Double-click a – box to collapse the hierarchy. The methods or properties for the selected class are displayed. By default, only those methods or properties that are defined by the class are displayed. To see all methods or properties that are available to the class (that is, methods or properties also defined by an ancestor of the class), select the **Include inherited** check box. To see all methods or properties (even those not available to the selected class), select the **Include all** check box.

3. Select a method or property. Information about the selected method or property is displayed.

If the **Library Browser** is not displaying your user-defined objects, close the **Library Browser**, recompile the include files that contain your user-defined objects (**Run** > **Compile**), and then re-open the **Library Browser**.

Examples of Documenting User-Defined Methods

This topic contains examples of adding user-defined methods, properties, and functions to the **Library Browser**.

```
#************************************************
class:     DialogBox
...
#** custom method
method:    VerifyNumChild (iExpectedNum)
parameter: iExpectedNum: The expected number of child objects (INTEGER).
notes:     Verifies the number of child objects in a dialog box.

Documenting user-defined properties: Add the property descriptions to the
appropriate class section in 4test.txt, such as:
#*********************************************
class:    DialogBox
...

#** custom property
property: iNumChild
notes:    The number of child objects in the dialog box.

Documenting user-defined functions: Create a group called User-defined
functions and document your functions, such as:
group:     User-defined functions

function:  FileOpen (sFileName)
parameter: sFileName = "myFile": The name of the file to open.
notes:     Opens a file from the application.

function:  FileSave (sFileName)
parameter: sFileName = "myFile": The name of the file to save.
notes:     Saves a file from the application.
```

Web Classes Not Displayed in Library Browser

This functionality is supported only if you are using the Classic Agent.

Problem

The class hierarchy in the **Library Browser** does not include the Web classes, which are `BrowserChild`, `HtmlText`, and so on.

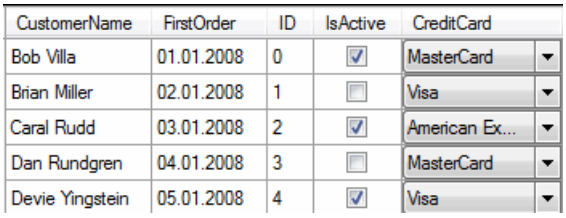
Possible Causes and Solutions

No browser extension is enabled.	Make sure that at least one browser extension is enabled.
Enhanced support for Visual Basic is enabled.	Disable Visual Basic by un-checking the **ActiveX** check box for the Visual Basic application in the **Extension Enabler** and **Extensions** dialog boxes.

Text Recognition Support

Text recognition methods enable you to conveniently interact with test applications that contain highly customized controls, which cannot be identified using object recognition. You can use *text clicks* instead of coordinate-based clicks to click on a specified text string within a control.

For example, you can simulate selecting the first cell in the second row of the following table:



CustomerName	FirstOrder	ID	IsActive	CreditCard
Bob Villa	01.01.2008	0	☑	MasterCard
Brian Miller	02.01.2008	1	☐	Visa
Caral Rudd	03.01.2008	2	☑	American Ex...
Dan Rundgren	04.01.2008	3	☐	MasterCard
Devie Yingstein	05.01.2008	4	☑	Visa

Specifying the text of the cell results in the following code line:

```
table.TextClick("Brian Miller")
```

Text recognition methods are supported for the following technology domains:

- Win32.
- WPF.
- Windows Forms.
- Java SWT and Eclipse.
- Java AWT/Swing.

  Note: For Java Applets, and for Swing applications with Java versions prior to version 1.6.10, text recognition is supported out-of-the-box. For Swing applications with Java version 1.6.10 or later, you have to add the following command-line element when starting the application:

  ```
  -Dsun.java2d.d3d=false
  ```

  For example:

  ```
  javaw.exe -Dsun.java2d.d3d=false -jar mySwingApplication.jar
  ```
- xBrowser.

Text recognition methods

The following methods enable you to interact with the text of a control:

`TextCapture`	Returns the text that is within a control. Also returns text from child controls.
`TextClick`	Clicks on a specified text within a control. Waits until the text is found or the *Object resolve timeout*, which you can define in the synchronization options, is over.
`TextRectangle`	Returns the rectangle of a certain text within a control or a region of a control.
`TextExists`	Determines whether a given text exists within a control or a region of a control.

Text click recording

Text click recording is enabled by default. To disable text click recording, click **Options** > **Recorder** > **Recording** and uncheck the **OPT_RECORD_TEXT_CLICK** check box.

When text click recording is enabled, Silk Test Classic records `TextClick` methods instead of clicks with relative coordinates. Use this approach for controls where `TextClick` recording produces better results than normal coordinate-based clicks. You can insert text clicks in your script for any control, even if the text clicks are not recorded.

If you do not whish to record a `TextClick` action, you can turn off text click recording and record normal clicks.

The text recognition methods prefer whole word matches over partially matched words. Silk Test Classic recognizes occurrences of whole words previously than partially matched words, even if the partially matched words are displayed before the whole word matches on the screen. If there is no whole word found, the partly matched words will be used in the order in which they are displayed on the screen.

Example

The user interface displays the text *the hostname is the name of the host*. The following code clicks on *host* instead of *hostname*, although *hostname* is displayed before *host* on the screen:

```
control.TextClick("host")
```

The following code clicks on the substring *host* in the word *hostname* by specifying the second occurrence:

```
control.TextClick("host", 2)
```

Running Tests and Interpreting Results

This section describes how you can run your tests and interpret the generated results.

Running Tests

This section describes how you can run your tests with Silk Test Classic.

Creating a suite

After you have created a number of script files, you might want to collect them into a test suite. A suite is a file that names any number of scripts. Instead of running each script individually, you run the suite, which executes in turn each of your scripts and all the testcases they contain. Suite files have a `.s` extension.

1. Click **File** > **New**.
2. Select the **Suite** radio button and click **OK**. An untitled suite file is displayed.
3. Enter the names of the script files in the order you want them executed. For example, the following suite file executes the `find.t` script first, the `goto.t` script second, and the `open.t` script third:

   ```
   find.t
   goto.t
   open.t
   ```
4. Click **File** > **Save** to save the file.
5. If you are working within a project, you are prompted to add the file to the project. Click **Yes** if you want to add the file to the open project, or **No** if you do not want to add this file to the project.

Passing Arguments To a Script

You can pass arguments to a script. For example, you might want to pass in the number of iterations to perform or the name of a data file. All functions and test cases in the script have access to the arguments.

How to pass arguments to a script

All arguments are passed in as strings, separated by spaces, such as: Bob Emily Craig

If an argument is more than one word, enclose it with quotation marks. For example, the following passes in three arguments: `"Bob H" "Emily M" "Craig J"`

You can pass arguments to a script using the following methods:

- Specify them in the **Arguments** field in the **Runtime Options** dialog box(**Options** > **Runtime** from the menu bar).
- The **Arguments** field in the **Run Testcase** dialog box is used to pass arguments to a testcase, not to an entire script.
- Specify them in a suite file after a script name, such as: `find.t arg1 arg2`
- Provide arguments when you invoke Silk Test Classic from the command line.
- If you pass arguments in the command line, the arguments provided in the command line are used and any arguments specified in the currently loaded options set are not used. To use the arguments in the currently loaded options set, do not specify arguments in the command line.

Processing arguments passed into a test script

You use the `GetArgs` function to process arguments passed into a script. `GetArgs` returns a list of strings with each string being one of the passed arguments. Any testcase or function in a script can call `GetArgs` to access the arguments.

Example: passed arguments

The following testcase prints a list of all the passed arguments:

```
testcase ProcessArgs ( )
LIST OF STRING lsArgs
lsArgs = GetArgs ( )
ListPrint (lsArgs)

//You can also process the arguments individually. The following test case
prints the second argument passed:
testcase ProcessSecondArg ( )
LIST OF STRING lsArgs
lsArgs = GetArgs ( )
Print (lsArgs[2])

//The following testcase adds the first two arguments:
testcase AddArgs ()
LIST OF STRING lsArgs
lsArgs = GetArgs ( )
NUMBER nArgSum

nArgSum = Val (lsArgs[1]) + Val (lsArgs[2])
Print (nArgSum)
```

You can use the `Val` function to convert the arguments (which are always passed as strings) into numbers.

The `Val` function was used to specifying arguments `10 20 30` results in the following:

```
Script scr_args.t (10, 20, 30) - Passed
Passed:  1 test (100%)
Failed:  0 tests (0%)
Totals:  1 test, 0 errors, 0 warnings

Testcase AddArgs - Passed

30
```

Running a Test Case

When you run a test case, Silk Test Classic interacts with the application by executing all the actions you specified in the test case and testing whether all the features of the application performed as expected.

Silk Test Classic always saves the suite, script, or test plan before running it if you made any changes to it since the last time you saved it. By default, Silk Test Classic also saves all other open modified files whenever you run a script, suite, or test plan. To prevent this automatic saving of other open modified files, uncheck the **Save Files Before Running** check box in the **General Options** dialog box.

1. Make sure that the test case that you want to run is in the active window.
2. Click **Run Testcase** on the **Basic Workflow** bar.

   If the workflow bar is not visible, choose **Workflows** > **Basic** to enable it.

   Silk Test Classic displays the **Run Testcase** dialog box, which lists all the test cases contained in the current script.
3. Select a test case and specify arguments, if necessary, in the **Arguments** field.

   Remember to separate multiple arguments with commas.

4. To wait one second after each interaction with the application under test is executed, check the **Animated Run Mode (Slow-Motion)** check box.

   Typically, you will only use this check box if you want to watch the test case run. For instance, if you want to demonstrate a test case to someone else, you might want to check this check box. Executions of the default base state and functions that include one of the following strings are not delayed:

   - `BaseStateExecutionFinished`
   - `Connecting`
   - `Verify`
   - `Exists`
   - `Is`
   - `Get`
   - `Set`
   - `Print`
   - `ForceActiveXEnum`
   - `Wait`
   - `Sleep`
5. To view results using the TrueLog Explorer, check the **Enable TrueLog** check box. Click **TrueLog Options** to set the options you want to record.
6. Click **Run**. Silk Test Classic runs the test case and generates a results file.

   For the Classic Agent, multiple tags are supported. If you are running test cases using other agents, you can run scripts that use declarations with multiple tags. To do this, check the **Disable Multiple Tag Feature** check box in the **Agent Options** dialog box on the **Compatibility** tab. When you turn off multiple-tag support, 4Test discards all segments of a multiple tag except the first one.

Running a testplan

Before running a testplan, make sure that the window declarations file for the testplan is correctly specified in the **Runtime Options** dialog and that the testplan is in the active window.

- To run the entire testplan, click **Run** > **Run All Tests**. Silk Test Classic runs each testcase in the plan and generates a results file.
- To run only tests that are marked, click **Run** > **Run Marked Tests**. Silk Test Classic runs each marked test and generates a results file.

You can also run a single testcase without marking it.

If your testplan is structured as a master plan and associated subplans, Silk Test Classic automatically opens any closed subplans before running. SilkTest always saves the suite, script, or testplan before running it if you made any changes to it since the last time you saved it. By default, SilkTest also saves all other open modified files whenever you run a script, suite, or testplan. To prevent this automatic saving of other open modified files, uncheck the **Save Files Before Running** check box in the **General Options** dialog.

Running the currently active script or suite

1. Make sure the script or suite you want to run is in the active window.
2. Choose **Run** > **Run**. Silk Test Classic runs all the testcases in the script or suite and generates a results file.

Stopping a Running Testcase Before it Completes

To stop running a testcase before it completes:

- If your test application is on a target machine other than the host machine, click **Run** > **Abort**.

- If your test application is running on your host machine, press `Shift+Shift`.

Setting a Test Case to Use Animation Mode

To slow down a test case during playback so that it can be observed, set the test case to use *animation mode*. For instance, if you want to demonstrate a test case to someone else, you might want to use animation mode.

You can specify the animation mode when you run a test case, or you can specify the animation mode in the **Runtime Options** dialog.

To specify the animation mode using the **Runtime Options** dialog:

1. From the main menu, click **Options** > **Runtime**.
2. In the **Runtime Options** dialog, check the **Animated Run Mode (Slow-Motion)** check box.
3. Click **OK**.

Interpreting Results

This section describes how you can use the Difference Viewer, the results file, and the reports to interpret the results of your tests.

Overview of the Results File

A results file provides information about the execution of the test case, script, suite, or test plan. By default, the results file has the same name as the executed script, suite, or test plan, but with a `.res` extension (for example, `find.res`).

Whenever you run tests, Silk Test Classic generates a results file, which indicates how many tests passed and how many failed, describes why tests failed, and provides summary information. You can invoke comparison tools from within the results file that pinpoint exactly how the runtime results differ from your known baselines. Test-plan results files offer additional features, such as the ability to generate a Pass/Fail report or compare different runs of the test plan. When Silk Test Classic displays a results file, on the menu bar it includes the **Results** menu, which allows you to manipulate the results file and locate errors. The **Results** menu appears only when the active window displays a results file.

TrueLog Explorer

Silk Test Classic also provides the TrueLog Explorer to help you analyze test results files. You must configure Silk Test Classic to use the TrueLog Explorer and specify what you want to capture.

Multiple User Environments

A `.res` file can be opened by multiple users, as long as no test is in process. This means you cannot have two users run tests at the same time and write to the same results file. You can run a test on the machine while the file is open on the other machine. However, you must not add comments to the file on the other machine, or you will corrupt the .res file and will not be able to report the results of the test. If you add comments to the file on both machines, the comments will be saved only for the file that is closed (and therefore saved) first.

Default Settings

By default, the results file displays an overall summary at the top of the file, including the name of the script, suite, or testplan; the machine the tests were run on; the number of tests run; the number of errors and warnings; actual errors; and timing information. To hide the overall summary, click the summary and click **Results** > **Hide Summary**. For a script or suite results file, the individual test summaries contain

timing information and errors or warnings. For a testplan results file, the individual test summaries contain the same information as in the overall summary plus the name of the testcase and script file.

While Silk Test Classic displays the most current version of the script, suite, or testplan, by default Silk Test Classic saves the last five sets of results for each script, suite, or testplan executed. (To change the default number, use the **Runtime Options** dialog.) As results files grow after repeated testing, a lot of unused space can accumulate in the files. You can reduce a results file's size with the Compact menu option.

The format for the rest of a testplan results file follows the hierarchy of test descriptions that were present in the testplan. Test statements in the testplan that are preceded by a pound sign (#) as well as comments (using the `comment` statement) are also printed in the results file, in context with the test descriptions.

To change the default name and directory of the results file, edit the **Runtime Options** dialog.

Note: If you provide a local or remote path when you specify the name of a Results file in the **Directory/Field** field on the **Runtime Options** dialog, the path cannot be validated until script execution time.

Viewing Test Results

Whenever you run tests, a results file is generated which indicates how many tests passed and how many failed, describes why tests failed, and provides summary information.

1. Click **Explore Results** on the **Basic Workflow** or the **Data Driven Workflow** bars.
2. On the **Results Files** dialog box, navigate to the file name that you want to review and click **Open**.

By default, the results file has the same name as the executed script, suite, or test plan. To review a file in the TrueLog Explorer, open a `.xlg` file. To review a results file, open a `.res` file.

Difference Viewer Overview

To evaluate application logic errors, use the **Difference Viewer**, which you can invoke by clicking the box icon following an error message relating to an application's behavior.

Some expanded error messages are preceded by a box icon and three asterisks. What happens when you click the box icon depends on the error message.

If the error message relates to an application's:

- Appearance, as in bitmaps have different sizes, Silk Test Classic opens the **Bitmap Tool** for your platform. The **Bitmap Tool** compares baseline and results bitmaps.
- Behavior, as in Verify selected text failed, Silk Test Classic opens the **Difference Viewer**. The **Difference Viewer** compares actual and expected values for a given test case. It lists every expected (baseline) value in the left pane and the corresponding actual value in the right pane. Differences are marked with red, blue, or green lines, which denote different types of differences, for example deleted, changed, and added items.

You can use **Results** > **Next Result Difference** to find the next difference and update the values using **Results** > **Update Expected Value**.

Note: The **Difference Viewer** does not work for remote agent tests, because the compared values must be available on the local machine.

Errors And the Results File

You can expand the text of an error message or have Silk Test Classic find the error messages for you. To navigate from a test plan test description in a results file to the actual test in the test plan, click the test description and select **Results** > **Goto Source**.

Navigating to errors in the script

There are several ways to move from the results file to the actual error in the script:

- Double-click in the margin next to an error line to go to the script file that contains the 4Test statement that failed.
- Click an error message and select **Results** > **Goto Source**.
- Click an error message and press **Enter**.

What the box icon means

Some expanded error messages are preceded by a box icon and three asterisks.

If the error message relates to an application's behavior, as in `Verify selected text failed`, Silk Test Classic opens the **Difference Viewer**. The **Difference Viewer** compares actual and expected values for a given test case.

Application appearance errors

When you click a box icon followed by a bitmap-related error message, the bitmap tool starts, reads in the baseline and result bitmaps, and opens a **Differences** window and **Zoom** window.

Bitmap tool

In the **Bitmap Tool**:

- The baseline bitmap is the bitmap that is expected, which means the baseline for comparison.
- The results bitmap is the actual bitmap that is captured.
- The **Differences** window shows the differences between the baseline and result bitmap.

The **Bitmap Tool** supports several comparison commands, which let you closely inspect the differences between the baseline and results bitmaps.

Finding application logic errors

To evaluate application logic errors, use the **Difference Viewer**, which you can open by clicking the box icon following an error message relating to an application's behavior.

The Difference viewer

Clicking the box icon opens the **Difference Viewer**'s double-pane display-only window. It lists every expected (baseline) value in the left pane and the corresponding actual value in the right pane.

All occurrences are highlighted where expected and actual values differ. On color monitors, differences are marked with red, blue, or green lines, which denote different types of differences, for example, deleted, changed, and added items.

When you have more than one screen of values or are using a black-and-white monitor, use **Results** > **Next Result Difference** to find the next difference. Use **Update Expected Values**, described next, to resolve the differences.

Updating expected values

You might notice upon inspecting the **Difference Viewer** or an error message in a results file that the expected values are not correct. For example, when the caption of a dialog changes and you forget to update a script that verifies that caption, errors are logged when you run the test case. To have your test case run cleanly the next time, you can modify the expected values with the **Update Expected Value** command.

Note: The **Update Expected Value** command updates data within a test case, not data passed in from the test plan.

Debugging tools

You might need to use the debugger to explore and fix errors in your script. In the debugger, you can use the special commands available on the **Breakpoint**, **Debug**, and **View** menus.

Marking failed test cases

When a test plan results file shows test case failures, you might choose to fix and then rerun them one at a time. You might also choose to rerun the failed test cases at a slower pace, without debugging them, simply to watch their execution more carefully.

To identify the failed test cases, make the results file active and select **Results** > **Mark Failures in Plan**. All failed test cases are marked and test plan file is made the active file.

Testplan Pass/Fail Report and Chart

A **Pass/Fail** report lists the number and percentage of tests that have passed during a given execution of the testplan. The report can be subtotaled by an attribute, for example, by Developer.

After you generate a **Pass/Fail** report, you can take these actions:

- Print the report.
- Export the report to a comma-delimited ASCII file.
- Chart a generated Pass/Fail report—that is, produce report information as a graph—or you can directly graph the testplan results information without a preexisting report.

You can mark manual tests as having passed or failed in the **Update Manual Tests** dialog. The **Pass/Fail** report includes in its statistics the manual tests that you have documented as having passed or failed.

Merging testplan results overview

Results files consist of a series of results sets, one set for each testplan run. You can merge different results sets in a results file. Merging results sets is useful when:

- Sections of the testplan are run separately (either by one person or by several people) and you need to create a single report on the testing process. That is, you want one results set that includes the different runs.
- The testplan is updated with new tests or subplans and you want a single results set to reflect the execution of the additional tests or subplans.

the two results sets are combined by merging the results set you selected in the **Merge Results** dialog into the currently open results set. The open results set is altered. No additional results set is created. The date and time of the altered results set reflect the more recent test run.

For example, let’s say that yesterday you ran a section of the testplan consisting of 20 tests and today you ran a different section of the testplan consisting of 10 tests. The merged results set would have today's date and would consist of the results of 30 tests.

Analyzing Results with the Silk TrueLog Explorer

This section describes how you can analyze results with the Silk TrueLog Explorer (TrueLog Explorer).

For additional information about TrueLog Explorer, refer to the *Silk TrueLog Explorer User Guide*, located in **Start** > **Programs** > **Silk** > **Silk Test** > **Documentation**.

TrueLog Explorer

The TrueLog Explorer helps you analyze test results files and can capture screenshots before and after each action, and when an error occurs. TrueLog Explorer writes the test result files and screenshots into a TrueLog file.

You can additionally use the **Difference Viewer** to analyze results for test cases that use the Open Agent.

You can enable or disable TrueLog Explorer:

- For all test cases using the **TrueLog Options** dialog box.
- Each time you run a specific test case using the **Run Testcase** dialog box.
- At runtime using the test script.

When you enable or disable TrueLog Explorer in the **Run Testcase** dialog box, Silk Test Classic makes the same change in the **TrueLog Options** dialog box. Likewise, when you enable or disable TrueLog Explorer in the **TrueLog Options** dialog box, Silk Test Classic makes the same change in the **Run Testcase** dialog box.

Note: By default, TrueLog Explorer is enabled when you are using the Open Agent, and disabled when you are using the Classic Agent. When TrueLog Explorer is enabled, the default setting is that screenshots are only created when an error occurs in the script and only test cases with errors are logged.

For additional information about TrueLog Explorer, refer to the *Silk TrueLog Explorer User Guide*, located in **Start** > **Programs** > **Silk** > **Silk Test** > **Documentation**.

TrueLog Limitations and Prerequisites

When you are using TrueLog with Silk Test Classic, the following limitations and prerequisites apply:

Remote agents	When you are using a remote agent, the TrueLog file is also written on the remote machine.
Suites	TrueLog is not supported when you are executing suites.
Mixed-agent scripts	TrueLog is not supported when you are executing mixed-agent scripts, which are scripts that are using both agents.
Multiple-agent scripts	TrueLog is supported only for one local or remote agent in a script. When you are using a remote agent, the TrueLog file is also written on the remote machine.
Open Agent scripts	To use TrueLog Explorer with Open Agent scripts, set the default agent in the toolbar to the Open Agent.
Classic Agent scripts	To use TrueLog Explorer with Classic Agent scripts, set the default agent in the toolbar to the Classic Agent.

Why is TrueLog Not Displaying Non-ASCII Characters Correctly?

TrueLog Explorer is a MBCS-based application, meaning that to be displayed correctly, every string must be encoded in MBCS format. When TrueLog Explorer visualizes and customizes data, many string conversion operations may be involved before the data is displayed.

Sometimes when testing UTF-8 encoded Web sites, data containing characters cannot be converted to the active Windows system code page. In such cases, TrueLog Explorer will replace the non-convertible characters, which are the non-ASCII characters, with a configurable replacement character, which usually is '?'.

To enable TrueLog Explorer to accurately display non-ASCII characters, set the system code page to the appropriate language, for example Japanese.

Opening the TrueLog Options Dialog Box

Use the TrueLog options to enable the TrueLog Explorer and to customize the test result information that TrueLog collects.

- To open the **TrueLog Options** dialog box from the main menu, click **Options** > **TrueLog**.
- To open the **TrueLog Options** dialog box from a test case, click **Run Testcase** on the **Basic Workflow** bar. If the workflow bar is not visible, click **Workflows** > **Basic** to enable it. In the **Run Testcase** dialog box, check the **Enable TrueLog** check box and then click **TrueLog Options**.

Setting TrueLog Options

Use the TrueLog options to enable TrueLog and to customize the test result information that the TrueLog collects.

Logging bitmaps and controls in a TrueLog may adversely affect performance. Because capturing bitmaps and logging information can result in large TrueLog files, you may want to log test cases with errors only and then adjust the TrueLog options for test cases where more information is needed.

1. Click **Options** > **TrueLog** to open the **TrueLog Options** dialog box.
2. To capture TrueLog data and activate logging settings, check the **Enable TrueLog** check box and then choose to capture data for:

All Testcases	Logs activity for all test cases, both successful and failed. This setting may result in large TrueLog files.
Testcases with errors	Logs activity only for test cases with errors. This is the default setting.

3. In the **TrueLog File** field, specify the location and name of the TrueLog file.

   This path is relative to the machine on which the Silk Test Classic Agent is running. The name defaults to the name used for the results file, with an `.xlg` extension. The location defaults to the same folder as the test case `.res` file.

   Note: If you provide a local or remote path in this field, the path cannot be validated until script execution time.
4. Only when you are using the Classic Agent, choose one of the following to set pre-determined logging levels in the **TrueLog Presets** section:

Minimal	Enables bitmap capture of desktop on error; does not log any actions.
Default	Enables bitmap capture of window on error; logs data for Select and SetText actions; enables bitmap capture for Select and SetText actions.
Full	Logs all control information; logs all events for browsers except for MouseMove events; enables bitmap capture of the window on error; captures bitmaps for all actions.

   If you enable Full logs and encounter a `Window Not Found` error, you may need to manually edit your script.
5. Only when you are using the Classic Agent, in the **Log the following for controls** section, specify the types of information about the controls on the active window or page to log.
6. Only when you are using the Classic Agent, in the **Log the following for browsers** section, specify the browser events that you want to capture.
7. Specify the amount of time you want to allow Windows to draw the application window before a bitmap is taken.
   - When you are using the Classic Agent, specify the delay in the **TrueLog Delay** field.
   - When you are using the Open Agent, specify the delay in the **Delay** field in the **Screenshot mode** section.

The delay can be used for browser testing. You can insert a `Browser.WaitForReady` call in your script to ensure that the DocumentComplete events are seen and processed. If WindowActive nodes are missing from the TrueLog, you need to add a `Browser.WaitForReady` call. You can also use the delay to optimize script performance. Set the delay as small as possible to get the correct behavior and have the smallest impact on script execution time. The default setting is 0.

8. To capture screenshots of the application under test:
   - When you are using the Classic Agent, check the **Enable Bitmap Capture** check box and then choose to capture bitmaps.
   - When you are using the Open Agent, determine how Silk Test Classic captures screenshots in the **Screenshot mode** section.
9. Only when you are using the Classic Agent, click the **Action Settings** tab to select the scripted actions you want to include in the TrueLog.

   When enabled, these actions appear as nodes in the Tree List view of the TrueLog.
10. Only when you are using the Classic Agent, in the **Select Actions to Log** section, check the **Enable** check box to include the corresponding 4Test action in the log. Each action corresponds to a 4Test method, except for `Click` and `Select`.
11. Only when you are using the Classic Agent, in the **Select Actions to Log** section, from the **Bitmap** list box, select the point in time that you want bitmaps to be captured.
12. Click **OK**.

Toggle TrueLog at Runtime Using a Script

This functionality is supported only if you are using the Classic Agent.

Toggle the TrueLog Explorer at runtime to analyze test results, capture screen-shots before and after each action, and capture screen-shots when an error occurs.

Use the test script to toggle TrueLog Explorer multiple times during the execution of a test case. For example, if you run a single test case to test multiple user interface menus, you can turn TrueLog on and off several times during the script to capture bitmaps for only a portion of the menus.

1. Set the TrueLog Explorer options to define what you want the TrueLog Explorer to capture.
2. Create or open the script that you want to modify.
3. Navigate to the portion of the script that you want to turn on or off.
4. To turn TrueLog off, type: `SetOption(OPT_PAUSE_TRUELOG, TRUE)`.
5. To turn TrueLog on, type: `SetOption(OPT_PAUSE_TRUELOG, FALSE)`.
6. Click **File** > **Save** to save the script.

Viewing Results Using the TrueLog Explorer

Use the TrueLog Explorer to analyze test results files, capture screenshots before and after each action, and capture screenshots upon error.

1. Set the TrueLog Explorer options.
2. Run a test case.
3. Choose one of the following:
   - Click **Results** > **Launch TrueLog Explorer**.
   - Click the **Explore Results** button on the **Basic Workflow** or the **Data Driven Workflow** bars.
4. On the **Results Files** dialog box navigate to the file name that you want to review and click **Open**.

By default, the results file has the same name as the executed script, suite, or testplan. To review a file in the **TrueLog Explorer**, open a `.xlg` file. To review a Silk Test Classic results file in Silk Test Classic, open a `.res` file.

Modifying Your Script to Resolve Window Not Found Exceptions When Using TrueLog

This functionality is supported only if you are using the Classic Agent.

When you run a script and get a `Window 'name' was not found` error, you can modify your script to resolve the issue. Use this procedure if all of the following options are set in the **TrueLog Options - Classic Agent** dialog box:

- The action **PressKeys** is enabled.
- Bitmaps are captured after or before and after the **PressKeys** action.
- **PressKeys** actions are logged.

The preceding settings are set by default if you select **Full** as the TrueLog preset.

To resolve this error, in your test case, use `FlushEvents()` after a `PressKeys()` and `ReleaseKeys()` pair. Or, you can use `TypeKeys()` instead.

There is no need to add `sleep()` calls in the script or to change timeouts.

```
testcase one()
  Browser.SetActive()
  // Google.PressKeys("<ALT-T>")
  // Google.ReleaseKeys("<ALT-T>")
  Google.TypeKeys("<ALT-T>")
  Agent.FlushEvents ()
  Google.TypeKeys("O")
  Agent.FlushEvents ()

  //recording
  IE_Options.SetActive ()
  IE_Options.PageList.Select ("Security")
  IE_Options.Security.SecurityLevelIndicator.SetPosition (2)
  BrowserMessage.SetActive ()
  BrowserMessage.OK.Click ()
  IE_Options.SetActive ()
  IE_Options.OK.Click()
```

Analyzing Bitmaps

This section describes how you can analyze bitmaps with the **Bitmap Tool**.

Overview of the Bitmap Tool

This topic contains a brief overview of the **Bitmap Tool**. To access more information about the **Bitmap Tool**, launch it and press `F1` or choose **Help** > **Help Topics**.

The **Bitmap Tool** is an application that allows you to test and correct your Windows application's appearance by comparing two or more bitmaps and identifying the differences between them. It is especially useful for testing inherently graphical applications, like drawing programs, but you can also check the graphical elements of other applications. For example, you might want to compare the fonts you expect to see in a dialog with the fonts actually displayed, or you might want to verify that the pictures in toolbar buttons have not changed.

It can be used as a stand-alone product, in which you create and compare bitmaps of entire windows, client areas, the desktop, or selected areas of the screen. More commonly, however, you use the tool in

conjunction with Silk Test Classic. Bitmaps captured can be opened in the **Bitmap Tool** where you can compare them using the tool's comparison features. Conversely, bitmaps captured by the bitmap tool can be compared by Silk Test Classic bitmap functions.

You can compare a baseline bitmap captured in the **Bitmap Tool** with one captured in a Silk Test Classic test case of your application.

- If you write test cases by hand, you can use Silk Test Classic built-in bitmap functions.
- If you prefer to record test cases through **Record** > **Testcase**, the **Verify Window** dialog box allows you to record a bitmap-related verification statement.

The **Bitmap Tool** can only recognize an operating system's native windows. In the case of the Abstract Windowing Toolkit (AWT), included with Sun Microsystems Java Development Kit (JDK), each control has its own window, since AWT controls are native Microsoft windows. As a result, the **Bitmap Tool** will only see the top level dialog box.

When to use the Bitmap Tool

You might want to use the **Bitmap Tool** in these situations:

- To compare a baseline bitmap against a bitmap generated during testing.
- To compare two bitmaps from a failed test.

For example, suppose during your first round of testing you create a bitmap using one of Silk Test Classic's built-in bitmap functions, `CaptureBItmap`. Assume that a second round of testing generates another bitmap, which your test script compares to the first. If the testcase fails, Silk Test Classic raises an exception but cannot specifically identify the ways in which the two images differ. At this point, you can open the **Bitmap Tool** from the results file to inspect both bitmaps.

Capturing Bitmaps with the Bitmap Tool

You can capture bitmaps by embedding bitmap functions and methods in a test case or by using the **Bitmap Tool**. This section explains how to capture bitmaps in the **Bitmap Tool**.

Use the **Capture** menu to capture a bitmap for any of the following in your application:

- A window.
- The client area of a window, which means the working area, without borders or controls.
- A selected rectangular area of the screen. This is especially useful for capturing controls within a window.
- The desktop.

Capturing a Bitmap with the Bitmap Tool

1. Start the application in which you want to capture bitmaps and set up the window or area to capture.
2. Start the **Bitmap Tool**.
3. If you want to change the current behavior of the tool window, click **Capture** > **Hide Window on Capture**.

   By default, the tool window is hidden during capture.
4. Choose a window or screen area to capture:

Window	Choose **Capture** > **Window**. Click the window you want to capture.
Client area	Choose **Capture** > **Client Area**. Click the client area you want to capture.
Selected rectangular area	Choose **Capture** > **Rectangle**. **1.** Move the mouse cursor to desired location to begin capture.

2. While pressing and holding the left mouse button, drag the mouse to outline a rectangle, and then release the mouse button to capture it. During outlining, the size of the rectangle is shown in pixels.

Desktop Click **Capture** > **Desktop**.

The **Bitmap Tool** creates a new MDI child window containing the newly captured bitmap. The title bar reads **Bitmap - (Untitled)** and the status line at the bottom right of the window gives the dimensions of the bitmap (height by width), and the number of colors.

5. Repeat steps 3 and 4 to capture another bitmap. Alternatively, open an existing bitmap file.
6. Save the bitmap.

   Now you are ready to compare the two bitmaps or create a mask for the baseline bitmap.

The Open Agent and Classic Agent capture bitmaps in different color depths. By default, the Open Agent captures bitmaps using the current desktop settings for color depth. The Classic Agent captures bitmaps as 24-bit images. If you create a mask file for the captured bitmap using the Bitmap tool, the mask file is saved as a 24-bit bitmap. If the bitmap mask file does not have the same color depth as the bitmap that you capture, an error occurs. To ensure that `VerifyBitmap` functions do not fail due to different color-depth settings between the captured image and the mask image, ensure that the bitmaps have the same color depth.

Capturing a Bitmap During Recording

1. Open the dialog box by pointing at the object you want to capture and pressing `Ctrl+Alt`.
2. Click the **Bitmap** tab.
3. Enter a file name in the **Bitmap File Name** field. Use the **Browse** button to select a directory name.

   The default path is based on the current directory. The default file name for the first bitmap is `bitmap.bmp`. Click **Browse** if you need help choosing a new path or name.
4. Choose whether to copy the **Entire Window**, **Client Area of Window**, or **Portion of Window**, and click **OK**.

   To capture a portion of the window, move the mouse cursor to the location where you want to begin. While pressing the left mouse button, drag the mouse to outline a rectangle, and then release the mouse button to capture the bitmap.

Silk Test Classic always adds a bitmap footer to the bitmap file. This means that the physical size of the bitmap will be slightly bigger than if you capture the bitmap in the **Bitmap Tool**. The bitmap footer always contains the window tag for a given bitmap.

The Open Agent and Classic Agent capture bitmaps in different color depths. By default, the Open Agent captures bitmaps using the current desktop settings for color depth. The Classic Agent captures bitmaps as 24-bit images. If you create a mask file for the captured bitmap using the Bitmap tool, the mask file is saved as a 24-bit bitmap. If the bitmap mask file does not have the same color depth as the bitmap that you capture, an error occurs. To ensure that `VerifyBitmap` functions do not fail due to different color-depth settings between the captured image and the mask image, ensure that the bitmaps have the same color depth.

Capturing All or Part of the Zoom Window in Scan Mode

1. Make sure the **Capture** > **Hide Window** is unchecked.

   If necessary, select the item to uncheck the check mark.
2. Click **Next** or **Previous** until the **Zoom** window contains the difference you want to capture.
3. Perform one of the following actions to capture the desired part of the **Zoom** window:

Entire Zoom window Press `Ctrl+W` and select the **Zoom** window.

Client area of Zoom window	Press `Ctrl+A` and select the **Zoom** window.
Selected area of Zoom window	Press `Ctrl+R`. Move the mouse cursor to desired location to begin capture. While pressing and holding the left mouse button, drag the mouse to the screen location to end capture, and release the mouse button.

4. Optionally, you can fit the bitmap in its window, resize it, and save it.

Saving Captured Bitmaps

You can, if you want, save the bitmaps you have captured in the **Bitmap Tool**. You should adopt a naming convention that helps you distinguish between the first bitmap in the comparison, called the baseline bitmap, and the second bitmap, called the result bitmap. You can make the distinction in the file name itself, for example, by appending or prefixing a `b` or `r` to the name and using the same file extension for all bitmap files. Or you might use the same file name for both baseline and result bitmaps and add a unique file extension.

> **Example**
>
> You save baseline and result bitmaps of the **Open** dialog box as `open.bmp` and `open.rmp`. Alternatively, you might name them `openbase.bmp` and `openres.bmp`, respectively.

The following table lists the file extensions supported by the **Bitmap Tool**. We recommend that you use `.bmp` for baseline bitmaps and `.rmp` for result bitmaps.

If you are saving	And you want the file name to be	Then use this extension
Baseline bitmap	Identical to the result bitmap's	`.bmp`
Result bitmap	Identical to the baseline bitmap's	`.rmp`
Either baseline or result bitmap	Unique	`.bmp` or `.dib` (Device Independent Bitmap)

Note: Silk Test Classic uses `.rmp` for bitmaps that are captured within a test case and fail verification.

Comparing Bitmaps

The **Bitmap Tool** can create and graphically locate the differences between two bitmaps. You can use all Windows functionality to resize, save, and otherwise manipulate bitmaps, in addition to the special comparison features included in the tool.

Using the **Bitmap Tool**, you can:

- Show the areas of difference.
- Zoom in on the differences.
- Jump from one zoomed difference to the next.
- View on-line statistics about the bitmaps.
- Edit (copy and paste), print, and save bitmaps.
- Create masks.

The **Bitmap Tool** has the following major comparison commands:

Command	Description
Show	Creates a **Differences** window, which is a child window containing a black-and-white bitmap. Black represents areas with no differences and white represents areas with differences.
Zoom	Creates a special, not sizable, **Zoom** window with three panes and resizes and stacks the **Baseline**, **Differences**, and **Result** windows. • The top pane of the **Zoom** window contains a zoomed portion of the **Baseline** window. • The middle pane shows a zoomed portion of the **Differences** window. • The bottom pane shows a zoomed portion of the **Result** window. All three zoomed portions show the same part of the bitmap. When you move the mouse within any of the three windows, the **Bitmap Tool** generates a simultaneous and synchronized real-time display in all three panes of the **Zoom** window. While in scan mode, you can capture the **Zoom** window to examine a specific bitmap difference.
Scan	The tool indicates the location of the first difference it finds by placing a square in the same relative location of the **Baseline**, **Result**, and **Differences** windows. The three panes of the **Zoom** window also show the difference.
Comparison Statistics	Provides statistics about the bitmaps.

You can also compare bitmaps by creating and applying masks.

Rules for Using Comparison Commands

You should be familiar with the following rules before using the commands:

- If you are comparing two new bitmaps captured in the tool, designate one bitmap as the baseline, the other as the result bitmap.
- If you are comparing two existing, saved bitmaps, open first the bitmap that you consider the baseline. The tool automatically designates the first bitmap you open as the baseline, and the second as the result.
- The commands must be used in this order: **Show**, **Zoom**, and **Scan**.

Bitmap Functions

`CaptureBitmap`, `SYS_CompareBitmap`, `WaitBitmap`, and `VerifyBitmap` are built-in bitmap-related 4Test functions. In particular, `VerifyBitmap` is useful for comparing a screen image during the execution of a test case to a baseline bitmap created in the **Bitmap Tool**. If the comparison fails, Silk Test Classic saves the actual bitmap in a file. In the following example, the code compares the test case bitmap (the baseline) against the bitmap of `TestApp` captured by `VerifyBitmap`:

```
TestApp.VerifyBitmap ("c:\sample\testbase.bmp")
```

Baseline and Result Bitmaps

To compare two bitmaps, you must designate one bitmap in the comparison as the baseline and the second bitmap as the result. While you may have many bitmap files open in the **Bitmap Tool**, at any one time only one bitmap can be set as the baseline and one as the result. If you want to set new baseline and result bitmaps, you must first un-set the current assignments.

These designations are temporary and at any time you can set and reset a bitmap as a baseline, result, or neither.

Designating a Bitmap as a Baseline

To designate a bitmap as a baseline:

In the **Bitmap Tool**, click **Bitmap** > **Set Baseline**. The **Set Baseline** menu item is checked. The title bar of the child window changes to **Baseline Bitmap -- filename.bmp**.

Designating a Bitmap as a Results File

To designate a bitmap as a results file:

In the **Bitmap Tool**, click **Bitmap** > **Set Result**. The **Set Result** menu item is checked. The title bar of the child window changes to **Result Bitmap -- filename.rmp**.

Un-Setting a Designated Bitmap

Uncheck the menu item. For example, to un-set a baseline bitmap, uncheck **Bitmap** > **Set Baseline**. The check mark is removed.

Uncheck the menu item.

For example, to un-set a baseline bitmap, uncheck **Bitmap** > **Set Baseline**.

The check mark is removed.

Zooming the Baseline Bitmap, Result Bitmap, and Differences Window

Choose **Differences** > **Show** and then **Differences** > **Zoom**.

The tool arranges the **Baseline Bitmap** on top, the **Result Bitmap** on the bottom, and the **Differences** window in the middle. To the right of these, the tool creates a **Zoom** window with three panes, arranged like the bitmap windows

Looking at Statistics

The **Differences** > **Comparison Statistics** command displays information about the baseline and result bitmaps, with respect to width, height, colors, bits per pixel, number of pixels, and the number and percentage of differences (in pixels).

Viewing Statistics by Comparing the Baseline Bitmap and the Result Bitmap

To view statistics by comparing the baseline bitmap and the result bitmap:

Click **Differences** > **Comparison Statistics**. The **Bitmap Comparison Statistics** window opens.

Note: The number of colors is derived from the following formula: number of colors = 2 ^ (bits per pixel).

Exiting from Scan Mode

To exit from the scan mode:

Click **Differences** > **Scan**. Exiting scan leaves the tool in zoom mode.

Starting the Bitmap Tool

This section lists the locations from which you can start the **Bitmap Tool**.

Starting the Bitmap Tool from its Icon and Opening Bitmap Files

1. Click **Start** > **Programs** > **Silk** > **Silk Test** > **Tools** > **Silk Test Bitmap Tool**. The **Bitmap Tool** window displays.
2. Do one of the following:

Open an existing bitmap file	Click **File** > **Open** and specify a file in the **Open** dialog box. See *Overview of Comparing Bitmaps*.
Capture a new bitmap	See *Capturing a Bitmap in the Bitmap Tool*.

Starting the Bitmap Tool from the Results File

When the verification of a bitmap fails in a test case, Silk Test Classic saves the actual result in a bitmap file with the same name as the baseline bitmap but with the extension `.rmp`. So, if the bitmap file `testbase.bmp` fails the comparison, Silk Test Classic names the result bitmap file `testbase.rmp`. It also logs an error message in the results file.

Note: In some cases this error message does not reflect an actual error. In particular, when Silk Test Classic compares a bitmap it captured with one captured in the **Bitmap Tool**, the comparison fails because Silk Test Classic stores footer information in its bitmap. The bitmaps might in fact be identical in all ways except for this information.

To compare the actual bitmap generated by the test case against the baseline bitmap generated by the bitmap tool or one of Silk Test Classic's built-in functions, click the box icon preceding the error message.

Silk Test Classic opens the bitmap tool, opens both the baseline bitmap, which is the expected bitmap as a `.bmp` file, and the result bitmap, which is the actual bitmap as a `.rmp` file, creates a **Results/View Differences** and places it in between the baseline bitmap and the result bitmap. The right portion of the tool displays a three-paned **Zoom** window.

Starting the Bitmap Tool from the Run Dialog Box

1. Click **Start** > **Run**. The **Run** dialog box displays.
2. Type the pathname of the tool's executable file and any parameters in the **Command Line** field and click **OK**. The **Bitmap Tool** starts. Any bitmaps you specified on the command line are opened.
3. See *Overview of Comparing bitmaps*.
4. If you did not specify any files in the command line, go to the next step.

   You can now open existing bitmaps created in Silk Test Classic or in the tool, or you can capture new bitmaps.
5. Do one of the following:

Open an existing bitmap file	Click **File** > **Open** and specify a file in the **Open** dialog box. See *Overview of Comparing Bitmaps*.
Capture a new bitmap	See *Capturing a Bitmap in the Bitmap Tool*.

Using Masks

A mask is a bitmap that you apply to the baseline and result bitmaps in order to exclude any part of a bitmap from comparison by the **Bitmap Tool**. For example, if you are testing a custom object that is painted on the screen and one part of the object is variable, you might want to create a mask to filter out the variable part from the bitmap comparison.

You might consider masking any differences that you decide are insignificant or that you know will vary in an effort to avoid test case failure. For example, suppose a test case fails because one bitmap includes a flashing area of a dialog box. In the **Bitmap Tool** you can block the flashing area from the two bitmaps by creating and applying a mask to them. Once a mask is applied and the masked bitmaps are saved, the mask becomes a permanent part of the baseline bitmaps you are comparing. Masks can also be saved in separate files and used in test cases.

You can create a mask in two ways:

- By converting the **Differences** window to a mask. A mask created this way filters out all differences.
- By opening a new mask window and specifying rectangular areas to mask.

The Open Agent and Classic Agent capture bitmaps in different color depths. By default, the Open Agent captures bitmaps using the current desktop settings for color depth. The Classic Agent captures bitmaps as 24-bit images. If you create a mask file for the captured bitmap using the Bitmap tool, the mask file is saved as a 24-bit bitmap. If the bitmap mask file does not have the same color depth as the bitmap that you capture, an error occurs. To ensure that `VerifyBitmap` functions do not fail due to different color-depth settings between the captured image and the mask image, ensure that the bitmaps have the same color depth.

Prerequisites for the Masking Feature

Before using the masking feature, you must:

- Capture or open two bitmaps to compare. Set `baselinesetbaseline` and `resultsetresult` bitmaps, if currently un-set.
- Determine which sections you need to mask. Use one or more comparison `featurescomparisoncmds`, if necessary, to locate bitmap differences.

Applying a Mask

1. Open the mask bitmap file and click **Bitmap** > **Set Mask**.
2. Click **Edit** > **Apply Mask**.

The Open Agent and Classic Agent capture bitmaps in different color depths. By default, the Open Agent captures bitmaps using the current desktop settings for color depth. The Classic Agent captures bitmaps as 24-bit images. If you create a mask file for the captured bitmap using the Bitmap tool, the mask file is saved as a 24-bit bitmap. If the bitmap mask file does not have the same color depth as the bitmap that you capture, an error occurs. To ensure that `VerifyBitmap` functions do not fail due to different color-depth settings between the captured image and the mask image, ensure that the bitmaps have the same color depth.

Editing an Applied Mask

You can edit a mask after it has been applied:

- To add to the mask, place the mouse cursor in the baseline bitmap window at the position where you want to begin adding to the mask. Click and drag the mouse cursor to outline a rectangle. Then release the left mouse button.
- To delete part of the mask, place the mouse cursor in the baseline bitmap window at the position where you want to begin deleting part of the mask. While pressing and holding the **Shift** key, drag the mouse cursor over the area of the existing map that you want to delete, and then release the **Shift** key and the left mouse button.

Creating and Applying a Mask that Excludes Some Differences or Just Selected Areas

1. Click **Edit** > **New Mask**. The bitmap tool creates an empty **Mask Bitmap child** window that is the same size as the baseline bitmap.

2. Using the **Differences** window to help you locate differences, place the mouse cursor in the baseline bitmap window at the position where you want to begin creating the mask. As you press and hold the left mouse button, drag the mouse cursor to outline a rectangle. Then release the left mouse button. The rectangular outline in the baseline map changes to a filled-in rectangle. The mask bitmap window also contains a like-sized rectangle in the same relative location.
3. Repeat step the previous step until you have completed the mask.
4. If you want to delete a portion of the mask, place the mouse cursor in the baseline bitmap window at the position where you want to begin editing. While pressing the Shift key and then the left mouse button, drag the mouse cursor over the area of the existing map that you want to delete, and then release the Shift key and the left mouse button.

   The area of the mask overlapped by the rectangle outline disappears in both the baseline and mask bitmap window.
5. Choose **Edit** > **Apply Mask**. The bitmap tool applies the mask to the result bitmap and closes the **Differences** window.
6. Choose one of the following actions:

Keep the baseline and result bitmaps with the mask applied	Save the bitmap files. The mask is now a permanent part of the bitmap files.
Unapply the mask	Close the mask bitmap window. Saving is optional.
Keep the mask as it is	Save the mask file.
Edit the mask	Choose **File** > **Save** and close the mask bitmap window. This un-applies the mask.

The Open Agent and Classic Agent capture bitmaps in different color depths. By default, the Open Agent captures bitmaps using the current desktop settings for color depth. The Classic Agent captures bitmaps as 24-bit images. If you create a mask file for the captured bitmap using the Bitmap tool, the mask file is saved as a 24-bit bitmap. If the bitmap mask file does not have the same color depth as the bitmap that you capture, an error occurs. To ensure that `VerifyBitmap` functions do not fail due to different color-depth settings between the captured image and the mask image, ensure that the bitmaps have the same color depth.

Creating and Applying a Mask that Excludes All Differences

1. Click **Differences** > **Show** to open a **Differences** window, if one is not already open.
2. Click **Differences** > **Convert to Mask**. A message is displayed: `Bitmaps are now identical on screen.`
3. Click **OK**.

   The bitmap tool creates an untitled mask bitmap from the **Differences** window, swapping black and white, and applies the mask to the baseline and result bitmaps.
4. Choose one of the following actions:

Keep the baseline and result bitmaps with the mask applied	Save the bitmap files. The mask is now a permanent part of the bitmap files.
Unapply the mask	Close the mask bitmap window. Saving is optional.
Keep the mask as it is	Save the mask file.
Edit the mask	Choose **File** > **Save** and close the mask bitmap window. This un-applies the mask.

The Open Agent and Classic Agent capture bitmaps in different color depths. By default, the Open Agent captures bitmaps using the current desktop settings for color depth. The Classic Agent captures bitmaps as 24-bit images. If you create a mask file for the captured bitmap using the Bitmap tool, the mask file is saved as a 24-bit bitmap. If the bitmap mask file does not have the same color depth as the bitmap that

you capture, an error occurs. To ensure that `VerifyBitmap` functions do not fail due to different color-depth settings between the captured image and the mask image, ensure that the bitmaps have the same color depth.

Saving a Mask

Masks can be saved in a file, applied to the baseline and result bitmaps for you to examine on screen only, or applied to and saved in the baseline and result bitmap files. Once masks are applied and saved, they become a permanent part of the baseline and result bitmaps. The advantage of saving the mask alone is that later you can read in the mask file and apply it to the bitmap on screen, thus allowing you to keep the bitmap in its original state.

You can supply the name of a mask bitmap file (as well as its associated baseline bitmap file) as an argument to bitmap functions.

The **Bitmap Tool** supports the `.msk` file extension for mask files. Alternatively, you can designate a mask in the file name and use the generic `.bmp` extension. We recommend, however, that you use the `.msk` extension.

The following bitmap-related functions accept mask files as arguments:

- `GetBitmapCRC`
- `SYS_CompareBitmap`
- `VerifyBitmap`
- `WaitBitmap`

The Open Agent and Classic Agent capture bitmaps in different color depths. By default, the Open Agent captures bitmaps using the current desktop settings for color depth. The Classic Agent captures bitmaps as 24-bit images. If you create a mask file for the captured bitmap using the Bitmap tool, the mask file is saved as a 24-bit bitmap. If the bitmap mask file does not have the same color depth as the bitmap that you capture, an error occurs. To ensure that `VerifyBitmap` functions do not fail due to different color-depth settings between the captured image and the mask image, ensure that the bitmaps have the same color depth.

Analyzing Bitmaps for Differences

This section describes how you can analyze bitmaps for differences.

Scanning Bitmap Differences

To scan the differences between the baseline and result bitmaps:

Click **Differences** > **Scan** or **Differences** > **Next**. The tool indicates the location of the first difference it finds by placing a square in the same relative location of the **Baseline**, **Result**, and **Differences** windows. The three panes of the **Zoom** window also show the difference.

Showing Areas of Difference

The **Show** command creates a **Differences** window which is a child window containing a black-and-white bitmap. Black represents areas with no differences and white represents areas with differences.

Graphically Show Areas of Difference Between a Baseline and a Result Bitmap

To graphically show the differences between a baseline and a result bitmap:

Click **Differences** > **Show**. The **Bitmap Tool** displays a **Differences** window along with the source baseline and result bitmaps from which it was derived.

Moving to the Next or Previous Difference

You must first create a **Differences** window and a **Zoom** window using **Differences** > **Show** and **Differences** > **Zoom**.

The **Scan** command on the **Differences** menu automates zoom mode and causes the bitmap tool to scan for differences from left to right and top to bottom. When the first difference is found, a small square, 32 x 32 pixels, is shown in the **Baseline Bitmap**, **Result Bitmap**, and **Differences Bitmap** windows in the same relative location. In addition, that location is shown in all three panes in the **Zoom** window.

Click **Differences** > **Next** or **Differences** > **Previous**.

Zooming in on the Differences

The **Zoom** command creates a special, not sizable, **Zoom** window with three panes and resizes and stacks the **Baseline**, **Differences**, and **Result** windows.

- The top pane of the **Zoom** window contains a zoomed portion of the **Baseline Bitmap** window.
- The middle pane shows a zoomed portion of the **Differences** window.
- The bottom pane shows a zoomed portion of the **Result Bitmap** window.

All three zoomed portions show the same part of the bitmap. When you move the mouse within any of the three windows, the bitmap tool generates a simultaneous and synchronized real-time display in all three panes of the **Zoom** window.

While in scan mode, you can capture the **Zoom** window to examine a specific bitmap difference.

Working with Result Files

This section describes how you can use result files to interpret the results of your tests.

Attaching a comment to a result set

You can attach comments to individual results sets to record useful information about the test run:

1. Open the results file.
2. Click **Results** > **Select** to display the **Select Results** dialog.
3. Select the results set to which you want to attach a comment.
4. Type the comment in the **Comment** text field at the bottom of the dialog. The comment appears in the Comment column in the **Select Results** dialog.
5. Click **OK**.

Silk Test Classic displays the comments in the various dialogs that list results sets, such as the **Extract Results** and **Delete Results** dialogs.

Comparing Result Files

The **Compare Two Results** command allows you to quickly note only the results that have changed from a prior run without having to look at the same errors over again. The command identifies differences based on the following criteria:

- A test passes in one test plan run and fails in the other.
- A test fails in both runs but the error is different.
- A test is executed in one test plan run but not in the other.

Silk Test Classic uses the test descriptions as well as the test statements to identify and locate the various cases in the test plan. Therefore, if you change the descriptions or statements between runs, Silk Test Classic will not be able to find the test when you run **Compare Two Results**.

1. Open two results files.
2. Make the results set you want to compare to another results set the active window.
3. Choose **Results** > **Compare Two Results**.
4. On the **Compare Two Results** dialog, select a results set from the list box and click **OK**.
5. When the results set is displayed again, a colored arrow is positioned in the left margin for every test that is different.

   A red arrow indicates that the difference is due to the pass/fail state of the test changing.

   A magenta arrow indicates that the difference is due to the addition or removal of the test in the compared test run.
6. Click **Results** > **Next Result Difference** to search for the next difference or choose **Results** > **Next Error Difference** to search for the next difference that is due to the change in a pass/fail state of a test.

Silk Test Classic uses the test descriptions as well as the script, testcase, and testdata statements to identify and locate the various cases in the test plan and in the results set. When test results overlap in the two results set that were merged, the more recent run is used. If you change a test description between runs or modify the statements, Silk Test Classic might be unable to find the test when you try to merge results. Silk Test Classic places these orphaned tests at the top of the results set.

Customizing results

You can modify the way that results appear in the results file as follows:

- Change the colors of elements in the results file
- Change the default number of results sets
- Display a different set of results
- Remove the unused space in a results file

You can also view an individual summary.

Deleting a results set

1. Click **Results** > **Delete**. Silk Test Classic displays the **Delete Results** dialog with the most current results set displayed first.
2. Select the set of results you want to delete and click **OK**.

Change the default number of results sets

1. Click **Options** > **Runtime**. The **Runtime Options** dialog box displays.
2. In the **History Size** field, change the number to the number of results files you want.

   Note: By default, five result sets are kept.

Changing the Colors of Elements In the Results File

1. In Silk Test Classic, click **Options** > **Editor Colors** to display the **Editor Colors** dialog.
2. Select an element from the **Editor Item** list box.

3. Select one of the 16 colors from the palette or modify the RGB values of the selected color. To modify RGB value, select the color. Slide the bar to the left or right, click the spin buttons, or type specific RGB values until you get the color you want.
4. When you are satisfied with the color, click **OK**.

To revert to the default colors, click **Reset**. By default, these results file elements are displayed in the following colors:

Results file element	Default color/icon
Error messages and warnings	Red plus sign (bold on black-and-white monitor)
Warnings only	Purple plus sign
Test descriptions of executed tests	Dark blue
Test descriptions of unexecuted tests	Grayed out
Other descriptive lines	Black

Fix incorrect values in a script

1. Make the results file active.
2. Click **Results** > **Update Expected Value**.
3. Optionally, select **Run** > **Testcase** in order to run the test and confirm that it now passes. The expected values in the script are replaced with the actual values found at runtime.

Marking Failed Testcases

When a testplan results file shows testcase failures, you might choose to fix and then rerun them one at a time. You might also choose to rerun the failed testcases at a slower pace without debugging them to watch their execution more carefully.

Make the results file active and click **Results** > **Mark Failures in Plan**.

All failed testcases are marked and the testplan is made the active file.

Merging results

You can merge results in two different ways:

- Merging two results sets in a results file.
- Merging results of manual tests.

Navigating to errors

To find and expand the next error or warning message in the results file, choose **Edit** > **Find Error**. To skip warning messages and find error messages only, in the **Runtime Options** dialog, uncheck the check box labeled **Find Error stops at warnings**.

You can also use the **Find**, **Find Next**, and **Go to Line** commands on the **Edit** menu to navigate through a results file.

To expand an error message to reveal the cause of an error, click the red plus sign preceding the message. In addition to the cause, you can see the call stack which is the list of 4Test functions executing at the time the error occurred.

There are several ways to move from the results file to the actual error in the script:

- Double-click the margin next to an error line to go to the script file that contains the 4Test statement that failed.
- Click an error message and choose **Results** > **Goto Source**.
- Click an error message and press **Enter**.

To navigate from a testplan test description in a results file to the actual test in the testplan, click the test description and click **Results** > **Goto Source**.

Viewing an individual summary

1. Click a testcase line in a suite or script results file, or click a test description in a testplan results file.
2. Click **Results** > **Show Summary**.

Storing and Exporting Results

You can store and export results in a variety of ways:

- Store results in an unstructured ASCII format.
- Exporting results to a structured file for further manipulation.
- Sending the results directly to Issue Manager.

Storing results

Silk Test Classic allows you to extract the information you want in an unstructured ASCII text format and send it to a printer, store it in a file, or look at it in an editor window.

To store results in an unstructured ASCII format

1. Click **Results** > **Extract**.
2. In the **Extract To group** box on the **Extract Results** dialog, select the radio button for the destination of the extracted output: **Window (default)**, **File**, or **Printer**.
3. In the Include group box, check one or more check boxes indicating which optional text, if any, to extract. (This optional text is in addition to the output selected in the **Expand group** box.) The choices are:
4. Select a radio button in the **Expand** group box indicating which units to extract information about. Select **Scripts**, **Scripts and Testcases (default)**, or **Anything with Errors**.
5. Select one or more results sets from the **Results to Extract** group box.
6. Click **OK**.

Exporting Results to a Structured File for Further Manipulation

1. Click **Results** > **Export**. The **Export Results** dialog displays.
2. Specify the file name. By default, the name `results file.rex` is suggested (for results export).
3. Specify which fields you want to export to the file.
4. Specify how you want the fields delimited in the file. The default is to comma delimit the fields and put quotations marks around strings.

   You can pick another built-in delimited style listed in the **Export format** list box or select **Custom** and specify your own delimiters.
5. To include header information in the file, check the **Write header** check box. Header information contains the name of the results file, which fields were exported, and how the fields were delimited.

6. To include the directory and file that stores the results file in the file, check the **Write paths relative to the results file** check box.
7. Specify which results sets you want to export. The default is the results set that is currently displayed in the results window.
8. Click **OK**. The information is saved in a delimited text file. You can import that file into an application that can process delimited files, such as a spreadsheet.

Removing the unused space in a results file

1. Open a results file.
2. Click **Results** > **Compact**. The file size is reduced.

Sending Results Directly to Issue Manager

Silk Central Issue Manager is the defect-tracking product that you can use to create and manage bug reports, enhancement requests, and documentation issues for your application. Issue Manager is integrated with Silk Test Classic. You can associate individual Silk Test Classic tests with defects stored in Silk Central Issue Manager and have Silk Central Issue Manager process the defects based on the results of the tests.

You can pass your test results to Silk Central Issue Manager in two ways:

Sending results directly to Silk Central Issue Manager	This is the easiest way to pass the results if you are running both Silk Central Issue Manager and Silk Test Classic.
Exporting the results to a `.rex` file for importing later in Silk Central Issue Manager.	`.rex` files can be read correctly by Silk Central Issue Manager 3.2 (and above). While you can export a `.rex` file to previous versions of SilkRadar, syntax/data errors occur.

Logging Elapsed Time Thread and Machine Information

Using the **Runtime Options** dialog, you can specify that you want to log elapsed time, thread number, and current machine information. This information is then written to the results file where you can display and sort it. For example, if you encounter nested testcases in the results files because you use multi-threading, check this check box to record thread number information in your results file. Then, you can sort the lines in your results file by thread number to better navigate within the nested testcases.

1. Click **Options** > **Runtime** to open the **Runtime Options** dialog.
2. In the **Results** area, check the **Log elapsed time**, **thread**, and **machine for each output line** check box.
3. Click **OK**.

Presenting Results

This section describes how you can use charts and reports to present the results of your tests.

Fully customize a chart

1. Generate the **Pass** > **Fail** report and click the **Chart** tab.
2. Click the area of the chart that you want to customize, for example, the text that appears for the title and footnote.

3. Double-click the selected area. A dialog displays showing the properties for the selected area. (You can also right-click anywhere on a chart and select the area you want to modify from a popup menu.)
4. Make your changes.
5. Click **OK**.

Generate a Pass/Fail Report on the Active Test Plan Results File

Note: You can only generate pass/fail reports for the results of test plans, not for the results of individual tests.

1. Make sure the test plan results file you want to report on is active, and then click **Results** > **Pass/Fail Report**.
2. On the **Results Pass/Fail Report** dialog box, select an attribute to report on from the **Subtotal by Attribute** list.
3. Click **Generate**.
4. Take one of the following actions:

Subtotal the report by a different attribute	Select a different attribute in the **Subtotal By Attribute** list, then click **Generate**.
Print the report	Click **Print**. You can set the margins, headers and footers, print quality, and fonts for the report. To change the font, click **Font**. To change the printer setup, click **Setup**. When you have finished setting these options, click **OK** to print the report.
Chart the report	Display the **Chart** tab.
Write the report to a comma-delimited ASCII file	Click **Export**, specify the full path of the file and click **OK**. You can open the file in a spreadsheet application that accepts comma-delimited data.

Producing a Pass/Fail Chart

You can create a chart out of a generated **Pass/Fail** report, or you can directly create a graph of the test plan results information without a preexisting report.

Note: You can only generate pass/fail reports for the results of test plans, not for the results of individual tests.

1. Open the result file of a test plan execution in Silk Test Classic.
2. In the Silk Test Classic menu, click **Results** > **Pass/Fail Report**. The **Pass/Fail Report** dialog box opens.
3. Click the **Chart** tab.

   If you have already generated a report, Silk Test Classic displays a chart of the generated report. You might need to resize the window so there is enough room to display the chart well. If you have not generated a report, Silk Test Classic displays a default chart, which allows you to modify chart parameters before you actually generate the chart.
4. Perform one of the following actions:

Change basic charting properties	1. Click **Setup**. The **Chart Settings** dialog is displayed. 2. To change the chart type, select an option from the **Chart Type** list. Silk Test Classic provides bar charts, line charts, and area charts.

3. Click **Apply** to update the chart and leave the **Chart Settings** dialog open. You can also choose whether the chart is three- dimensional, is stacked (for bar charts), and displays a legend, which describes the data being charted. Silk Test Classic displays a model that represents how the chart will look based on current settings.

Add the results from another execution of the test plan to the chart

1. Click **Select**. The **Select Results** dialog is displayed, listing recent runs of the current test plan. Silk Test Classic keeps a history of results for each test plan. The number of results it keeps is determined by the value for **History Size** in the **Runtime Options** dialog.
2. Select the results you want to add to the chart. The results from the selected execution of the test plan will be added to the results currently charted. You can use this feature to compare two different runs of the same tests to spot problem areas. You can chart today's results, then click **Setup** and select yesterday's results to have both appear on one chart.

Move a part of the chart

1. Click the part you want to move, such as the title, legend, or footnote (the text that displays below the chart). The area is selected.
2. Drag it with the mouse.

Print the chart

1. Click **Print**. The **Print Pass/Fail Chart** dialog displays. You can specify a header or footer.
2. Click **OK** to print the chart.

Copy the chart to the clipboard

1. Right-click anywhere on the displayed chart, and then click **Copy**.
2. The chart is placed on the clipboard. You can paste it into another application.

Change advanced charting properties

Usually you can get the chart you want using the default and basic charting properties. But if you want more customization, you can modify just about any property in the chart, including:

- text that appears for the title and footnote
- font used for any text in the chart
- location for the title, legend, and footnote
- colors used for the data
- size and spacing of the bars in bar charts
- borders and shading to the background (backdrop) of any area
- See Customizing a chart.

Generate the chart

Once you are satisfied with the chart parameters, click **Generate**. The **Pass/Fail** chart is displayed.

Displaying a different set of results

1. Click **Results** > **Select**. Silk Test Classic displays the **Select Results** dialog with the most current results set displayed first.
2. Select the set of results you want to see and click **OK**.

Debugging Test Scripts

This section describes how you can debug your test scripts with Silk Test Classic.

Designing and testing with debugging in mind

Here are some suggestions for designing and testing a script that will facilitate debugging it later:

- Plan for debugging (and robustness) when you're designing the script, by having your functions check for valid input and output, and perform some operation that informs you if problems occur.
- Test each function as you write it, by building it into a small script that calls the function with test arguments and performs some operation that lets you know it works. Or use the debugger to step through the execution of each function individually after you have coded all (or part) of the script.
- Test each routine with the full range of valid data values, including the highest and lowest valid values. This is a good way to find errors in control loops.
- Test each routine with invalid values; it should reject them without crashing.
- Test each routine with null (empty) values. Depending on the purpose of the script, it might be useful if a reasonable default value were provided when input is incomplete.

Overview of the Debugger

You will find out about many of the errors or inconsistencies in your scripts when Silk Test automatically raises an exception in response to them. Some problems, however, cause a script to work in unexpected ways, but do not generate exceptions. You can use the debugger to solve these kinds of problems.

Using the debugger, you can step through a script a line at a time and stop at specified breakpoints, as well as examine local and global variables and enter expressions to evaluate.

But the debugger is more than just a tool for fixing scripts. You can also use it to help find problems in your application using the debugging facilities to step through the application slowly so you can determine just where a problem occurs.

The debugger allows you to view the results of your testing in the following ways:

- View the debugging transcript when you debug a script. See *Viewing the debugging transcript*. Silk Test records error information and output from the print statements in a transcript, not in a results file.
- Examine the debugging variables while you are debugging a test script. See *View variables*.
- View the call stack. The call stack is a description of all the function calls that are currently active in the script you are debugging. By viewing the call stack, you can trace the flow of execution, possibly uncovering errors that result when a script's flow of control is not what you intended. To view the current call stack, choose **View** > **Call Stack**. Silk Test Classic displays the call stack in a new window. To return to the script being debugged, press F6 or choose **View** > **Module** and select the script from the list.

You cannot use the debugger from plan (`*.pln`) files, however, you could call test cases from a `main()` function and debug it from there.

You may not modify files when you are using the debugger. If you want to fix a problem in a file, you must first stop the debugger, and then make the fix.

Executing a script in the debugger

Once you have set one or more breakpoints, you can start executing your script.

1. Click **Debug** > **Run**. Silk Test Classic runs the script until it hits the first breakpoint, an error occurs, or the script ends. Silk Test Classic displays a blue triangle next to the line where it stopped running the script.
2. Click **Debug** > **Continue**. Silk Test Classic runs the script until it hits the next breakpoint, an error occurs, or the script ends.
3. Click **Debug** > **Step Into**, **Debug** > **Step Over**, or **Debug** > **Finish Function** to run a smaller chunk of your script.

Starting the debugger

There are several ways to enter the debugger:

Script in active window	Click **Run** > **Debug**. Silk Test Classic enters the debugger and pauses. It does not set a breakpoint.
Another script	Click **File** > **Debug** and select the script file from the Debug dialog. Silk Test Classic enters the debugger and pauses. It does not set a breakpoint.
A testcase	With a script active, click **Run** > **Testcase**, select a testcase from the **Run Testcase** dialog, and click **Debug**. Silk Test Classic enters the debugger and sets a breakpoint at the first line of the testcase.
An application state	Click **Run** > **Apllication State**, select an application state from the **Run Application State** dialog, and click **Debug**. Silk Test Classic enters the debugger and sets a breakpoint at the first line of the application state definition.
A plan file	You cannot use the debugger from plan files (`*.pln`) , however, you can call testcases from a `main()` function and debug it from there.

When you enter the debugger, you can execute the script under your control.

You cannot edit a script when you are in the debugger.

Debugger menus

In debugging mode, the menu bar includes three additional menus:

- **Debug** menu commands allow you to control the script's flow.
- **Breakpoint** menu commands add or remove a breakpoint.
- **View** menu commands display different elements of the running script (for example, local and global variables, the call stack, and breakpoints) and evaluate expressions.

Stepping into and over functions

Sometimes the key to locating a bug in your code is to divide the script up into discrete functions, and debug each function separately. One good way to do this is with the **Step Into**, **Step Over**, and **Finish Function** commands on the **Debug** menu. These commands let you run and test functions individually:

Step Into	Step through the function one line at a time, executing each line in turn as you go.
Step Over	Speed up debugging if you know a particular function is bug-free.

Finish Function	Execute the script until the current function returns. Silk Test Classic sets the focus at the line where the function returns. Try using `Finish Function` in combination with **Step Into** to step into a function and then run it.

Working with scripts

To run the script you are debugging	Click **Debug** > **Run**. The script runs until a breakpoint is hit, an error occurs, or it terminates.
To reset a script	Click **Debug** > **Reset**. This frees memory, frees all variables, and clears the call stack. The focus will be at the first line of the script.
To stop execution of a running script	Press `Shift+Shift` when running a script on the same machine or choose **Debug** > **Abort** when running a script on a different machine.

Exiting the debugger

You can leave the debugger whenever execution is stopped.

To exit the debugger, click **Debug** > **Exit**.

Breakpoints

A breakpoint is a line in the script where execution stops so that you can check the script's status. The debugger lets you stop execution on any line by setting a breakpoint. A breakpoint is denoted as a large red bullet.

One useful way to debug a script is to pause it with breakpoints, observe its behavior and check its state, then restart it. This is useful when you are not sure what lines of code are causing a problem.

During debugging, you can:

- Set breakpoints on any executable line where you want to check the call stack.
- Examine the values in one or more variables.
- See what a script has done so far.

You cannot set breakpoints on blank lines or comment lines.

Setting Breakpoints

You can set breakpoints on most lines in the script except for blank lines or comment lines.

The First Line of a Function (or testcase)

1. Click **Breakpoint** > **Add**.
2. Double-click a module name to have the functions declared in that module listed in the **Function** list box.
3. Double-click a function name to set a breakpoint on the first line of that function.

Any Line in a Function (or testcase)

Place the cursor on the line where you want to set a breakpoint and choose **Breakpoint** > **Toggle**.

or

Double-click in the left margin of the line.

A Specific Line in a Script

1. Click **Breakpoint** > **Add**.
2. In the **Breakpoint** field, type the number of the line on which you want to set a breakpoint. (For example, entering 8 sets a breakpoint on the eighth line of the script.)
3. Click **OK**.

Temporary Breakpoints

Click **Debug** > **Run To Cursor** to set a temporary breakpoint (indicated by a hollow red circle in the margin) on the line containing the cursor. The script runs immediately stopping at the current line. The breakpoint is cleared after it is hit.

Viewing Breakpoints

To view a list of all the breakpoints in a script, click **View** > **Breakpoints**.

Deleting Breakpoints

You can delete breakpoints in any of the following ways:

All breakpoints

1. Click **Breakpoint** > **Delete All**.
2. Click **Yes**.

An individual breakpoint

Place the cursor on the line where the breakpoint is set and click **Breakpoint** > **Toggle** .

or

Double-click in the left margin of the line

One or more breakpoints

1. Click **Breakpoint** > **Delete**.
2. Select one or more breakpoints from the list box and click **OK**.

Variables

This section describes how you can use variables.

Viewing variables

To view a list of all the local variables that are in scope (accessible) from the current line, including their values, choose **View** > **Local Variables**.

To view a list of global variables, choose **View** > **Global Variables**. The variables and their values are listed in a new window.

If a variable is uninitialized, it is labelled `<unset>`.

If a variable has a complex value, like an array, Silk Test Classic might need to display its result in collapsed form. Use **View** > **Expand Data** and **View** > **Collapse Data** (or double-click the plus icon) to manipulate the display.

To return to the script being debugged, press F6 or choose View/Module and select the script from the displayed list.

Changing the value of variables

To change the value of an active variable, select the variable and type its new value in the **Set Value** field.

While viewing variables, you can also change their values to test various scenarios.

When you resume execution, Silk Test Classic uses the new values.

Expressions

This section describes how you can use expressions.

Overview of Expressions

If you type an identifier name, the result is the value that variable currently has in the running script. If you type a function name, the result is the value the function returns. Any function you specify must return a value, and must be in scope at the current line.

Properties and methods for a class are valid in expressions, as long as the declaration for the class they belong to is included in one of the modules used by the script being debugged.

If an expression evaluates to a complex value, like an array, Silk Test Classic may display its result in collapsed form. Use **View** > **Expand Data** or **View** > **Collapse Data** (or double-click on the plus icon) to manipulate the display.

When a script reaches a breakpoint, you can evaluate expressions.

Evaluate expressions

1. Click **View** > **Expression**.
2. Type an expression into the input area and press **Enter** to view the result.

If you type an identifier name, the result is the value that variable currently has in the running script. If you type a function name, the result is the value the function returns. Any function you specify must return a value, and must be in scope at the current line.

Properties and methods for a class are valid in expressions, as long as the declaration for the class they belong to is included in one of the modules used by the script being debugged.

If an expression evaluates to a complex value, like an array, Silk Test Classic may display its result in collapsed form. Use **View** > **Expand Data** or **View** > **Collapse Data** (or double-click the plus icon) to manipulate the display.

Enabling View Trace Listing

When you run a script, Silk Test Classic can record all the methods that the script invoked in a transcript. Each entry in the transcript includes the method name and the arguments passed into the method. You can use this information to debug the script, because you can see exactly which functions were actually called by the running script.

1. Click **Options** > **Runtime** to display the **Runtime Options** dialog box.
2. Check the **Print Agent Calls** and the **Print Tags with Agent Calls** check boxes.

3. Run the script.

   The transcript contains error information and the output from print statements, and additionally lists all methods that are called by the script.
4. To check the agent trace during debugging, when execution pauses, click **View** > **Transcript**.

Viewing a list of modules

1. Click **View** > **Module**. Silk Test Classic displays a list of modules in the **View Module** dialog. The list includes all the modules loaded at startup (that is, the modules loaded by `startup.inc`, including `winclass.inc`), so you can set breakpoints in functions, window class declarations, and so forth.
2. Double-click a module's name to view it in a debug window.

View the debugging transcripts

Choose **View** > **Transcript** when execution is stopped.

Silk Test Classic displays the transcript in a new window. To save its contents to a text file, choose **File** > **Save**.

The **Transcript** window has an **Execute** field that you can use to send commands to the application you are testing. You can type in any command that would be valid in a script and click **Execute**. For example, you might want to print the value of a variable or the contents of a window.

Debugging Tips

This section provides tips that might help you in debugging your tests.

Checking the precedence of operators

The order in which 4Test applies operators when it evaluates an expression may not be what you expect. Use parentheses, or break an expression down into intermediate steps, to make sure it works as expected. You can use **View** > **Expression** to evaluate an expression and check the result.

Code that never executes

To check for code that never executes, step through the script with **Debug** > **Step Into**. See the **Debug** menu for more information.

Global and local variables with the same name

It is usually not good programming practice to give different variables the same names. If a global and local variable with the same name are in scope (accessible) at the same time, your code can access only the local variable.

To check for repeated names, use **View** > **Local Variables** and **View** > **Global Variables** to see if two variables with the same name are in scope simultaneously.

Global variables with unexpected values

When you write a function that uses global variables, make sure that each variable has an appropriate value when the function exits. If another function uses the same variable later, and it has an unexpected value on entry to the function, an error could occur.

To check that a variable has a reasonable value on entry to a function, set a breakpoint on the line that calls the function and use the command **View** > **Global Variables** to check the variable's value.

Incorrect use of break statements

A `break` statement transfers control of the script out of the innermost nested `for`, `for each`, `while`, `switch`, or `select` statement only. `Break` exits from a single loop level, not from multiple levels. Use **Debug** > **Step Into** to step through the script one line at a time and ensure that the flow of control works as you expect. See **Debug** menu for more details.

Incorrect values for loop variables

When you write a `for` loop or a `while` loop, be sure that the initial, final, and step values for the variable that controls the loop are correct. Incrementing a loop variable one time more or less than you really want is a common source of errors.

To make sure a control loop works as you expect, use **Debug** > **Step** Into to step through the execution of the loop one statement at a time, and watch how the value of the loop variable changes using **View** > **Local Variables**. See **Debug** menu for more details.

Infinite loops

To check for infinite loops, step through the script with **Debug** > **Step Into**. See **Debug** menu for more details.

Typographical errors

It is easy to make typographical errors that the 4Test compiler cannot catch. If a line of code does nothing, this might be the problem.

Uninitialized variables

Silk Test Classic does not initialize variables for you. So if you have not initialized a variable on entry to a function, it will have the value `<unset>`. It is better to explicitly give a value to a variable than to trust that another function has already initialized it for you. Also, remember that 4Test does not keep local variables around after a function exits. The next time the function is called, its local variables could be uninitialized.

If you are in doubt about whether a variable has a reasonable value at a particular point, set a breakpoint there and use **View** > **Global Variables** or **ViewLocal Variables** to check the variable's value.

Troubleshooting the Classic Agent

This section provides information and workarounds for working with the Classic Agent.

ActiveX and Visual Basic Applications

This functionality is supported only if you are using the Classic Agent.

This section provides help and troubleshooting information for working with ActiveX and Visual Basic applications.

What Happens When You Enable ActiveX/Visual Basic?

This functionality is supported only if you are using the Classic Agent.

When you enable ActiveX/Visual Basic, Silk Test Classic updates to use the appropriate Visual Basic/ ActiveX include file and to merge the Visual Basic/ActiveX property sets and the Help text for the **Library Browser**.

Silk Test Classic Does Not Display the Appropriate Visual Basic Properties

This functionality is supported only if you are using the Classic Agent.

Check your property set initialization file if Silk Test Classic does not display the appropriate Visual Basic properties, when you try to verify these properties. The property set initialization file is called `vbprpset.ini` and is located in the Silk Test Classic installation directory. When you open this file in an editor, you should see familiar Visual Basic properties. If you do not see the Visual Basic properties, reinstall Silk Test Classic with enhanced Visual Basic and ActiveX support.

Silk Test Classic Does Not Recognize ActiveX Controls in a Web Application

This functionality is supported only if you are using the Classic Agent.

If you notice that Silk Test Classic does not recognize the ActiveX controls in your Web application, when you are recording or playing back tests, make sure that the ActiveX/Visual Basic support is enabled for your browser.

Silk Test Classic Displays an Error When Playing Back a Click on a Sheridan Command Button

This functionality is supported only if you are using the Classic Agent.

If you record a click against a Sheridan command button, which is handled by the `OLESSCommand` class in 4Test, Silk Test Classic does not play back the click and records an error in the results file for the test.

The cause of this behavior is that the `OLESSCommand` should be a windowless control.

To record the click, edit the class declaration for the `OLESSCommand` class in the `VBclass.inc` file, to have the class inherit from the `WindowlessControl` class instead of the `Pushbutton` class.

Silk Test Classic Displays Native Visual Basic Objects as Custom Windows

This functionality is supported only if you are using the Classic Agent.

The following two reasons might prevent Silk Test Classic from recording declarations for native Visual Basic objects, and instead recording them as custom windows (`CustomWin`):

- You did not follow the procedure for recording classes.
- The extension is not loaded properly into the application.

Record Class Finds no Properties or Methods for a Visual Basic Object

This functionality is supported only if you are using the Classic Agent.

If record class finds no properties or methods for an object, perform the following tasks:

- Make sure the extension is loaded and enabled properly.
- Verify that the object, whose class you are recording, is an ActiveX control or a native Visual Basic control.
- If the ActiveX control was created in Visual Basic 5, it must expose its properties and methods. For additional information on exposing properties and methods, refer to the Visual Basic 5 documentation.

Inconsistent Recognition of ActiveX Controls

This functionality is supported only if you are using the Classic Agent.

Silk Test Classic might recognize ActiveX controls, which often have a native class beginning with OLE, as having the native class `ATL:<variable hex #>`, or Silk Test Classic might recognize the ActiveX controls as children of an `ATL:<variable hex #>` control. In other cases, the parent control of an ActiveX control might seem to disappear on occasion. In such a case, the following settings in the `extend.ini` and `axext.ini` files might help.

- If the inconsistent recognition problem involves ATL controls, then first try setting `DontIgnore=ATL` in the `[VBOptions]` section.
- If the recognition problem occurs when Silk Test Classic invokes the application under test (AUT) especially if the AUT seems slow to render completely, then try to kill the agent while leaving the application running, and then restart the agent. If Silk Test Classic recognizes the control properly, then you should be able to correct the problem without having to kill the agent by setting `AxextDelay=<n>` in the `[VBOptions]` section, where <n> is the number of seconds that the application should require to start up completely.

  Note: The amount of time that Silk Test Classic requires to recognize the application when it is invoked is extended by `AxextDelay` seconds, so you should not make the number too large. In addition, if the application contains frame classes such as `AfxFrameOrView42` or `AfxMDIFrame42`, then you should try class-mapping them to `Ignore`. Ignoring those windows may eliminate some unnecessary layers and also make the Silk Test Classic recognition of the window hierarchy more consistent.

Test Failures During Visual Basic Application Configuration

This functionality is supported only if you are using the Classic Agent.

If the following suggestions do not address the problem you are having, you can enable your extension manually. The configuration of your Visual Basic application might fail for one of the following reasons:

Reason	Solution
The application might not be ready to test	To enable your application for testing: **1.** Click **Enable Extensions** on the **Basic Workflow** bar. **2.** On the **Enable Extensions** dialog box, select the application for which you want to enable extensions. **3.** Close and restart your application. Make sure the application has finished loading, and then click **Test**.
You might have another configured Visual Basic application open	You must close all configured Visual Basic applications before you can configure another Visual Basic application.

Application Environment

This section provides help and troubleshooting info for your application environment.

Dr. Watson when Running from Batch File

This functionality is supported only if you are using the Classic Agent.

If you get a Dr. Watson error when trying to record window declarations for a Java application launched through a batch file, the *classpath* set in the batch file is overriding the global *classpath*. Make sure the classpath in the batch file points to the appropriate Silk Test Classic .jar file.

I Cannot Get Silk Test Classic to Work With JBuilder or Oracle JDeveloper

This functionality is supported only if you are using the Classic Agent.

Make sure that you have configured Silk Test Classic properly for these environments.

Silk Test Classic does not Launch my Java Web Start Application

This functionality is supported only if you are using the Classic Agent.

Sometimes the default base state does not launch your Java Web Start application. If your application requires Java Web Start to launch, then you must manually edit the constant *sCmdLine* in your declaration for the main application window, which is designated as the *wMainWindow*.

Create New Test Frame detects the Windows command line that directly invoked the main window, so it will set *sCmdLine* to the JRE command line launched by Java Web Start. However, since you want to start Java Web Start instead of directly starting the JRE, you must edit *sCmdLine* to launch the Java Web Start executable, `javaws.exe`, with the .jnlp file for your application.

Example

```
const sCmdLine = "C:\Program Files\Java Web Start\javaws.exe C:
\MyAppDir\MyJWSApp.jnlp"
```

Which JAR File do I Use?

This functionality is supported only if you are using the Classic Agent.

For JDK/JRE 1.3, 1.4, and higher, use **SilkTest_Java3.jar**.

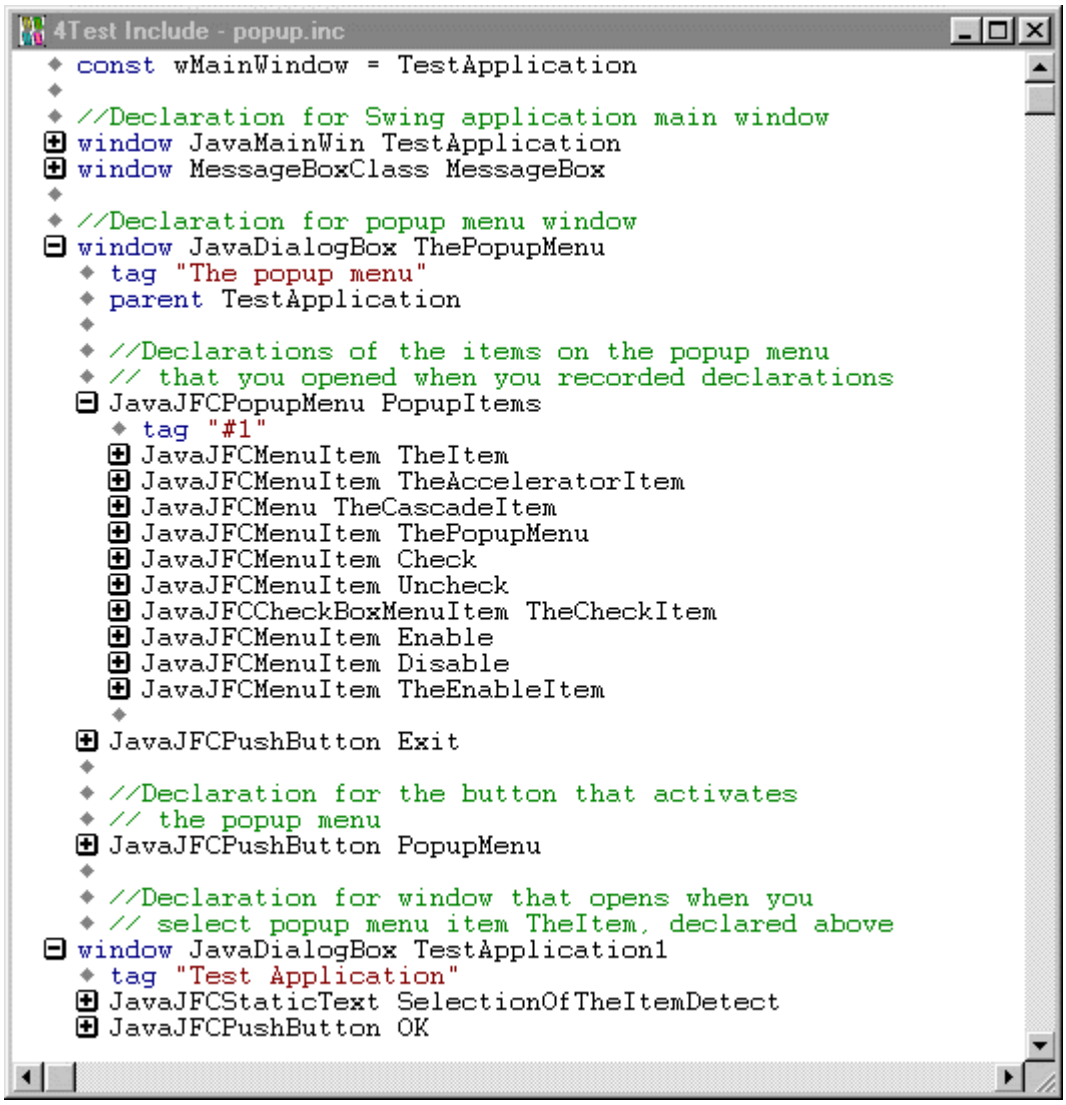
For information about new features, supported platforms and versions, known issues, and work-arounds, refer to the *Release Notes*.

Sample Declarations and Script for Testing JFC Popup Menus

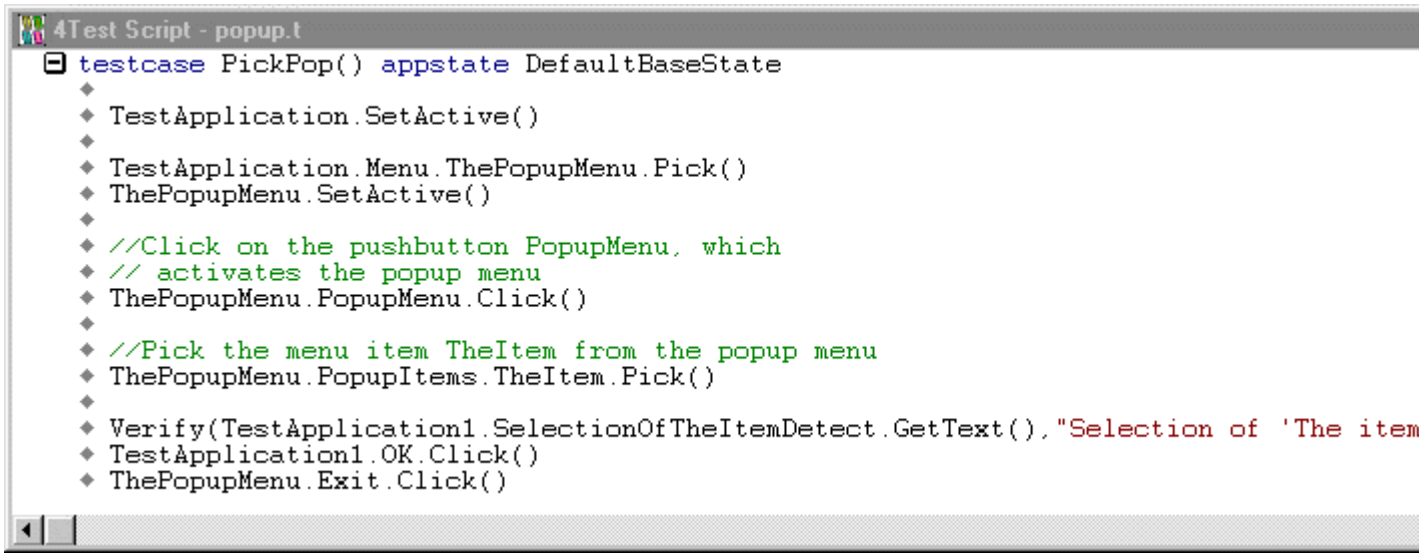
This functionality is supported only if you are using the Classic Agent.

When you record `Pick`() for a Java popup menu item, make sure that the recorder window shows the proper menu item. If the recorder seems to stick on the wrong menu item, move the mouse off of the popup menu and then back on to it. That should force the recorder to update.

Sample test frame:



Sample test script:



Java Extension Loses Injection when Using Virtual Network Computing (VNC)

This functionality is supported only if you are using the Classic Agent.

The Java extension can periodically stop recognizing Java objects on JFC (Swing) applications if the Agent machine, where the Java application is running, is being "viewed" remotely through Virtual Network Computing (VNC), a popular tool that lets you view and control a remote machine.

Symptoms of this problem include the following:

- The main window is viewed as a `JavaDialogBox`.
- An `*** Error: Window '[JavaMainWin]<window name>' was not found` message box displays.
- `JavaJFCMenu` items are not seen, resulting in an error such as `*** Error: Window JavaJFCMenu <menu name>' was not found`.

Avoid using VNC to view the automation which is running.

Troubleshooting Basic Workflow Issues

The following troubleshooting tips may help you with the basic workflow:

I restarted my application, but the Test button is not enabled

In order to enable the **Test** button on the **Test Extensions** dialog box, you must restart your application. Do not restart Silk Test Classic; restart the application that you selected on the **Enable Extensions** dialog box.

You must restart the application in the same manner. For example, if you are testing:

- A standalone Java application that you opened through a Command Prompt, make sure that you close and restart both the Java application and the Command Prompt window .
- A browser application or applet, make sure you return to the page that you selected on the **Enable Extensions** dialog box.

- An AOL browser application, make sure that you do not change the state of the application, for example resizing, or you may have issues with playback.

You can configure only one Visual Basic application at a time.

The test of my enabled Extension failed – what should I do?

If the test of your application fails, see *Troubleshooting Configuration Test Failures* for general information.

Browsers

This section provides help and troubleshooting information for working with browsers.

I Am not Testing Applets but Browser is Launched During Playback

This functionality is supported only if you are using the Classic Agent.

When you enable a browser extension in Silk Test Classic, the recovery system automatically launches the enabled browser when returning to DefaultBaseState. During sessions when you are testing only standalone Java applications, we recommend that you set **Default browser** to `none` in the **Runtime Options** dialog box.

Playback is Slow when I Test Applications Launched from a Browser

This functionality is supported only if you are using the Classic Agent.

When you test applications, not applets, launched from a browser, the recovery system performs several functions that might impact performance, including the following:

- Closes the Java application when exiting each test case.
- Loads the URL of the launch site and relaunches the Java application when entering each test case.

To avoid opening and closing the Java application for each test case, you can add the following code to your test frame:

1. Add an `Invoke`() method inside the `JavaMainWin` definition that will launch the Java application.
2. Inside the *wMainWindow* definition of your test frame, assign the constant *lwLeaveOpen* to the main window of the Java application.
3. Add `TestCaseEnter`() and `TestCaseExit`() methods at the top of the test frame.

Library Browser does Not Display Web Browser Classes

This functionality is supported only if you are using the Classic Agent.

Problem

The Library Browser does not display any of the classes for Web browsers.

This problem can be caused by either of two conditions:

- The extensions for the Web browser you are testing have not been enabled.

- Silk Test Classic was not able to successfully compile the files that populate the Library Browser with the browser classes.
- Successful compilation incorporates the information in the Silk Test Classic browser include file, for example the `extend\firefox.inc` file. The browser include file pulls in the appropriate help file (`.ht`) to populate the Library Browser with the additional classes (`extend\firefox.ht`).

Solution

1. Make sure that the browser extensions are enabled on the target and host computers.
   - On the target computer, from the Silk Test Classic program group, choose **Extension Enabler**. Check the **Primary Extension** for the browser to make sure it says `Enabled`.
   - On the host computer, from Silk Test Classic, choose **Options** > **Extensions**. Check the **Primary Extension** for the browser to make sure it says `Enabled`.
2. Check the Silk Test Classic **Runtime Options** dialog box. In Silk Test Classic, click **Options** > **Runtime**. Make sure the **Use Files** field contains the browser include file, for example `firefox.inc`.
3. Exit Silk Test Classic and restart it.
4. This action causes Silk Test Classic to recompile the include files.

Error Messages

This section provides help and troubleshooting information for error messages.

Agent not responding

Problem

You get the following error message:

```
Error: Agent not responding
```

This error can occur for a number of reasons.

Solution

Try any or all of the following:

- Restart the application that you are testing.
- Restart Silk Test Classic.
- Restart the Host machine.

If you are recording declarations on a very large page and get this error, consider increasing the `AgentTimeout`.

BrowserChild MainWindow Not Found When Using Internet Explorer 7.x

This functionality is supported only if you are using the Classic Agent.

Problem

My scripts are failing in Internet Explorer 7.x with the following error `[BrowserChild] MainWindow not found`.

Solution

When recording a new frame file using **Set Recovery System**, by default Silk Test does not explicitly state that the parent of the window is a browser. To resolve this issue, the "parent Browser" line must be added to the frame file. For example:

```
[-] window BrowserChild Google
[ ] tag "Google"
[ ] parent Browser
```

Cannot find file agent.exe

This functionality is supported only if you are using the Classic Agent.

Problem

Running a script gives the following error:

```
Cannot find file agent.exe [or one of its components]. Check to ensure the
path and filename are correct and that all required libraries are available.
```

Solution

1. Exit Silk Test Classic (this automatically shuts down the agent).
2. Explicitly start the agent by itself.
3. Restart Silk Test Classic.

Control is not responding

Problem

You run a script and get the following error: `Error: Control is not responding`

This is a catch-all error message. It usually occurs in a `Select( )` statement when Silk Test Classic is trying to select an item from a ListBox, TreeView, ListView, or similar control.

The error can occur after the actual selection has occurred, or it can occur without the selection being completed. In general the error means that the object is not responding to the messages Silk Test Classic is sending in the manner in which it expects.

Solution

Try these things to eliminate the error message:

- If the line of code is inside a `Recording` block, remove the `Recording` keyword.
- Set the following option just before the line causing the error: `Agent.SetOption(OPT_VERIFY_RESPONDING, FALSE).`
- If the selection is successful, but you still get the error, try using the `Do . . . except` feature.

Functionality Not Supported on the Open Agent

If you use Classic Agent functionality in an Open Agent script, an error message displays, stating that the functionality is not supported on the Open Agent.

Example

For example, if you try to call the `ClearTrap` function of the Classic Agent on a `MainWin` object in an Open Agent script, the following error message displays:

```
The Open Agent does not support the function
'MainWin::ClearTrap'#
```

Unable to Connect to Agent

Problem

You get the following error message: `Error: Unable to connect to agent`

This error can occur for a number of reasons.

Solution

Connect to the default agent	Click **Tools** > **Connect to Default Agent**. The command starts the Classic Agent or the Open Agent on the local machine depending on which agent is specified as the default in the **Runtime Options** dialog. If the Agent does not start within 30 seconds, a message is displayed. If the default Agent is configured to run on a remote machine, you must connect to it manually.
Restart the agent that you require for testing	Click **Start** > **Programs** > **Silk** > **Silk Test** > **Tools** > **Silk Test Open Agent** or **Start** > **Programs** > **Silk** > **Silk Test** > **Tools** > **Silk Test Classic Agent** .

Unable to Delete File

This functionality is supported only if you are using the Classic Agent.

While you are using the Basic Workflow to configure a Java application, Silk Test Classic is unable to delete the JVM files

You may have another application running that uses the same JVM as the application being configured or an application may be slow to release the JVM.

Close all applications that use the JVM (if any) and click **Retry**. That should give the application enough time to release the JVM.

If the suggestion above does not solve the problem, you can enable your extension manually.

Unable to Start Internet Explorer

This functionality is supported only if you are using the Classic Agent.

Problem

While trying to record or run a testcase you get the following message: `Error: Unable to start Internet Explorer #`

This error occurs because the extensions enabled in the **Extensions Enabler** and **Options** > **Extensions** do not match the default browser.

Solution

Review the settings for your default browser and make them consistent with the settings for the host machine in **Options** > **Extensions** and for the target machine in the **Extension Enabler**.

Variable Browser not defined

This functionality is supported only if you are using the Classic Agent.

Problem

If you are encountering the following message: `Error: Variable Browser is not defined`.

This error occurs because no browser extensions have been enabled.

Solution

Enable at least one browser extension.

Window Browser does not define a tag

This functionality is supported only if you are using the Classic Agent.

Problem

While trying to run or record testcases you get the following message: `Error: Window Browser does not define a tag for <Operating System>`.

This error occurs because no default browser has been specified.

Solution

You need to set a default browser. Whenever you record and run testcases, you need to have a default browser set so Silk Test Classic knows which browser to use.

Window is not active

Problem

You run a script and get the following error: `Error: Window 'name' is not active`.

This error means that the object Silk Test Classic is trying to act on is not active. This message applies to top-level windows (`MainWin`, `DialogBox`, `ChildWin`).

Solution

You can correct the error by doing one of the following:

1. Edit the script and add an explicit `SetActive()` statement to the window you are trying to act on just above the line where the error is occurring. An easy way to do this is to double-click the error in the results file. You will be brought to the line in the script. Insert a new line above it and add a line ending with the `SetActive()` method.
2. Tell Silk Test Classic not to verify that windows are active. There are two ways to do this:

   To turn off the verification globally, uncheck the **Verify that windows are active** option on the **Verification** tab in the **Agent Options** dialog (**Options** > **Agent**).

   To turn off the option in your script on a case by case basis, add the following statement to the script, just before the line causing the error: `Agent.SetOption(OPT_VERIFY_EXPOSED, FALSE).`

3. Then add the following line just after the line causing the error:
   `Agent.SetOption(OPT_VERIFY_EXPOSED, TRUE)`.

   This means Silk Test Classic will execute the action regardless of whether the window is active.
4. Extend the window time out to be greater than 10 by inserting the **Agent - Window Timeout** to `>= 10` into your `partner.ini`.

Window is not enabled

Problem

You run a script and get the following error: `Error: Window 'name' is not enabled`.

This error means that the object that Silk Test Classic is trying to act on is not enabled. This message applies to controls inside top-level windows (such as PushButton and CheckBox).

Solution

You can correct this problem in one of two ways.

- If the object is indeed disabled, edit the script and add the actions that will enable the object.
- If the object is in fact enabled and you want the script to perform the action, tell Silk Test Classic not to verify that a window is enabled:

To turn off the verification globally, uncheck the **Verify that windows are enabled** option on the **Verification** tab in the **Agent Options** dialog box (**Options** > **Agent**).

To turn off the option in your script on a case-by-case basis, add the following statement to the script, just before the line causing the error: `Agent.SetOption(OPT_VERIFY_ENABLED, FALSE)`

Then add the following line just after the line causing the error:
`Agent.SetOption(OPT_VERIFY_ENABLED, TRUE)`.

This means Silk Test Classic will execute the action regardless of whether the window is enabled.

Window is not exposed

Problem

You run a script and get the following error: `Error: Window 'name' is not exposed`.

Sometimes, applications are written such that windows are hidden to the operating system, even though they are fully exposed to the user. A running script might generate an error such as `Window not exposed`, even though you can see the window as the script runs.

Solution

While it might be tempting to simply turn off the checks for these verifications from the **Agent Options** > **Verification** dialog box, the best course of action is to take such errors on a case by case basis, and only turn off the verification in cases where the window is genuinely viewable, but Silk Test Classic is getting information from the operating system saying the object is not visible.

1. Add the following statement to the script, just before the line causing the error:
   `Agent.SetOption(OPT_VERIFY_EXPOSED, FALSE).`
2. Then add the following line just after the line causing the error:
   `Agent.SetOption(OPT_VERIFY_EXPOSED, TRUE).`

This means Silk Test Classic will execute the action regardless of whether it thinks the window is exposed.

Window not found

Problem

You run a script and get the following error: `Error: Window 'name' was not found`.

Resolution

This error occurs in the following situations:

When the window that Silk Test Classic is trying to perform the action on is not on the desktop.	If you are watching the script run, and at the time the error occurs you can see the window on the screen, it usually means the tag that was generated is not a correct tag. This could happen if the application changed from the time the script or include file was originally created. To resolve this issue, enable view trace listing in your script.
The window is taking more than the number of seconds specified for the window timeout to open.	To resolve this issue, set the **Window Timeout** value to prevent `Window Not Found` exceptions
Only if you are using the Classic Agent, in the TrueLog Options - Classic Agent dialog box, if all of the following options are set	• The action `PressKeys` is enabled. • Bitmaps are captured after or before and after the `PressKeys` action. • `PressKeys` actions are logged. The preceding settings are set by default if you select `Full` as the **TrueLog** preset. To resolve this issue, modify your test case.

Functions and Methods

This section provides help and troubleshooting information for functions and methods.

Class not Loaded Error

This functionality is supported only if you are using the Classic Agent.

You should not simply add the .jar file containing the Java class referenced in the `InvokeJava`() call to the directory containing the .jar files used by your application. `InvokeJava`() will be able to load your Java class only if it can be loaded by one of the following class loaders. If you receive a `Class Not Loaded` error when calling `InvokeJava`():

1. By default, `InvokeJava()` uses the ext class loader. To use this loader, the .jar file containing the Java class referenced in the `InvokeJava`() call should reside in the JVM's `lib\ext` directory with SilkTest's .jar file, which is `SilkTest_Java3.jar`.

   The class should be in the root directory of the .jar file, so that there is no path information in the .jar file.
2. Alternatively, `InvokeJava`() can use the application class loader. To use this loader, the .jar file containing the Java class referenced in the `InvokeJava`() call should be part of the application's classpath.

Exists Method Returns False when Object Exists

This functionality is supported only if you are using the Classic Agent.

There is a timing issue in Java that causes a delay when Java applications or applets render objects. To ensure that the `Exists` method detects Java objects, call it with a timeout parameter. You might need to experiment with different timeout values to find the one that works best for your system configuration, and application or applet.

For example `Verify (JavaAwtPushButton.Exists(1), TRUE)`.

How can I Determine the Exact Class of a java.lang.Object Returned by a Method

This functionality is available only for projects or scripts that use the Classic Agent.

Many Java methods return values of type `java.lang.Object`. In order to call such methods in Silk Test Classic, you must use `invokeMethods`() to call a method on the return object. Eventually, you must call a method that returns a 4Test-compatible value. However, you need to know the exact class of the `java.lang.Object` in order to know which methods are available for that object. Otherwise, you can only call `java.lang.Object` methods, which is a fairly limited list.

If your method returns a `java.lang.Object` value, you can use the following `invokeMethods`() call to return the name of the class of the return object:

```
STRING sClass = wObj.invokeMethods({"<method that returns java.lang.Object>",
  "getClass", "getName"}, {{<parameter list for the method of interest>}, {},
{}})
```

The above statement will call the following 3 Java methods:

<method that returns java.lang.Object> (<parameter list for the method of interest>)	This is the method of interest. It returns a value of type `java.lang.Object`, which is not 4Test-compatible and therefore must be used to call a new method. **Record Class** only lists this method if you check **Show all methods**. The method displays in the commented list below the declaration of `wObj.invokeMethods`().
getClass()	This `java.lang.Object` method returns a value of type `java.lang.Class`, which is not 4Test-compatible and therefore must be used to call a new method.
getName()	This `java.lang.Object` method returns a value of type `java.lang.String`, which is 4Test-compatible.

Once you know the name of the class, you can call methods specific to that class. Preferably those methods will return a 4Test-compatible type. Otherwise, you will need to chain additional methods in the `wObj.invokeMethods`() call.

Expanding on the previous example:

```
STRING sClass = wObj.invokeMethods({"<method that returns java.lang.Object>",
  "getClass", "getName"}, {{<parameter list for the method of interest>}, {},
{}})
  ANYTYPE aProp1
  ANYTYPE aProp2
  switch sClass
    case "ClassA"
      aProp1 = wObj.invokeMethods({"<method that returns java.lang.Object>",
        "getClassAProperty1"}, {{<parameter list for the method of
interest>}, {}})
      aProp2 = wObj.invokeMethods({"<method that returns java.lang.Object>",
```

```
        "getClassAProperty2"}, {{<parameter list for the method of
interest>}, {}})
    case "ClassB"
      aProp1 = wObj.invokeMethods({"<method that returns java.lang.Object>",
        "getClassBProperty1"}, {{<parameter list for the method of
interest>}, {}})
      aProp2=wObj.invokeMethods({"<method that returns java.lang.Object>",
        "getClassBProperty2"}, {{<parameter list for the method of
interest>}, {}})
    default
      RaiseError (E_UNSUPPORTED, "java.lang.Object is of an unknown class:
{sClass}")
```

`toString`() is a useful general method. It is a `java.lang.Object` method that returns a value of type `java.lang.String` and which translates to 4Test type STRING. You may be able to use `toString`() to return a value when you do not really understand what the previous method does. However, `toString`() may return a blank string, or the value may not make sense.

How to Define lwLeaveOpen

This functionality is supported only if you are using the Classic Agent.

Add the constant *lwLeaveOpen* inside the `wMainWindow` declaration to instruct the recovery system to leave the Java application open when returning to base state. You must assign this constant to the identifier of the main window of the Java application.

Example

The declaration for the main window of the Java application looks like the following:

```
window JavaMainWin TestApp
```

Following is the constant *lwLeaveOpen* inserted in the header section of the `wMainWindow` declaration.

Note: The constant *lwLeaveOpen* is assigned to *TestApp*, the identifier of the main window of the Java application.

```
window BrowserChild JavaAWTTestApplet
  tag "Java 1.1 AWT TestApplet"

  // The URL of this page
  const sLocation = "…"

  // The login user name
  // const sUserName = ?

  // The login password
  // const sPassword = ?

  // The size of the browser window
  // const POINT BrowserSize = {600, 400}

  // Sets the browser font settings to the default
  // const bDefaultFont = TRUE

  const lwLeaveOpen = {TestApp}
```

Defining TestCaseEnter and TestCaseExit Methods

This functionality is supported only if you are using the Classic Agent.

Add one `TestCaseEnter` and one `TestCaseExit` method at the top of your test frame to ensure that the Java application is not opened and closed unnecessarily. `TestCaseEnter` calls the Invoke method you defined earlier to launch the Java application only if it is not already running.

Note: `DefaultTestCaseEnter` and `DefaultTestCaseExit` are also called to retain the benefits of the recovery system in resetting the application to a base state.

Example

The following code sample shows how the `TestCaseEnter` and `TestCaseExit` methods are inserted at the top of a test frame:

```
TestCaseEnter()
DefaultTestCaseEnter()
if ( ! TestApp.Exists() )
  TestApp.Invoke()
TestApp.SetActive()
TestCaseExit(BOOLEAN bTRUE)
  Browser.SetActive()
DefaultTestCaseExit(bTRUE)

const wMainWindow = JavaAWTTestApplet
  window BrowserChild JavaAWTTestApplet
  window JavaMainWin TestApp
…
```

How to Write the Invoke Method

This functionality is supported only if you are using the Classic Agent.

Add an invoke method inside the `JavaMainWin` definition. This method should interact with the appropriate controls on the HTML page to launch the Java application from the browser. You can code this method by hand or click **Record** > **Method** to use the dialog box.

Example

An HTML page uses a pushbutton inside an applet to launch a standalone Java application from the browser.

The declaration for the `wMainWindow` looks like the following:

```
window BrowserChild JavaAWTTestApplet
```

The declaration for the pushbutton inside the applet looks like the following:

```
JavaApplet StartTheTestApplicationIn2
  tag "Start the Test Application in a"
JavaAwtPushButton StartTestApplication
tag
"Start Test Application"
```

The following is a sample `Invoke` method highlighted in blue and declared inside the declaration for the main window of the Java application.

Note: The method clicks on the pushbutton that launches the Java application.

```
window JavaMainWin TestApp
  tag "Test Application"
    Menu File
    Menu Control
    Menu Menu
    Menu DisabledMenu
```

```
    void OpenWindows (STRING sMenuItem)
    void Invoke()

JavaAWTTestApplet.StartTheTestApplicationIn2.StartTestApplicatio
n.Click()
```

I cannot Verify $Name Property during Playback

This functionality is supported only if you are using the Classic Agent.

If you do not explicitly set the native `$Name` property of a Java control, Java (AWT) assigns a different default name to each instance of the control. As a result, you will not be able to verify the `$Name` property of the control because it will have a different name each time you run your test, and each time you close and reopen it during one test run.

When you want to verify the `$Name` property of a Java control, explicitly name the control by adding a call to the native method `setName` in your script before each call to `VerifyProperties`.

If you do not want to verify the name of a Java control, make sure you uncheck the `$Name` property in the **Verify Window** dialog box.

Example

VerifyProperties will fail in the following script because the `JavaAwtCheckbox` *TheCheckBox* is assigned a different `$Name` during playback than when the test case was recorded:

```
testcase Test1 () appstate none
  recording
    TestApplication.SetActive ()
    TestApplication.Control.CheckBox.Pick ()
    xCheckBox.TheCheckBox.VerifyProperties ({...})
      ""
      {...}
        {"$Name", "checkbox0"}
    xCheckBox.SetActive ()
    xCheckBox.Exit.Click ()
```

We can make the test run successfully by explicitly setting the name of *TheCheckBox* before verifying properties, as shown in the following script:

```
testcase Test1 () appstate none
  recording
    TestApplication.SetActive ()
    TestApplication.Control.CheckBox.Pick ()
    xCheckBox.TheCheckBox.setName("checkbox0")
    xCheckBox.TheCheckBox.VerifyProperties ({...})
      ""
      {...}
        {"$Name", "checkbox0"}
    xCheckBox.SetActive ()
    xCheckBox.Exit.Click ()
```

Errors when calling nested methods

This functionality is supported only if you are using the Classic Agent.

You won't be able to call nested methods when any of the intermediate return objects are not 4Test-compatible. To work around this problem, add the method `invokeMethods` by hand to your test script. This method allows you to call nested methods inside Java.

Methods Return Incorrect Indexed Values in My Scripts

This functionality is supported only if you are using the Classic Agent.

There is an incompatibility in indexing between 4Test methods and native Java methods for classes such as `ListBox`, `PopupList`, and `ScrollBar`. 4Test methods are 1-based; Java native methods are 0-based. If you mix 4Test methods and native methods in a test script where you retrieve indexed values, you must compensate for the difference in indexing schemes to maintain the integrity of your test results.

We recommend that you do not mix methods in test scripts if at all possible. A good rule is that when both types of methods are available for all controls in your applications, use the 4Test methods only.

In situations where 4Test methods are not available for some of your classes and you must mix in native methods, use the following precautions when writing code to get indexed values:

- Do not pass an index of 0 to a 4Test method.
- Adjust indexes accordingly.

For `ListBox` *Lb*, the native Java AWT list method call *Lb*.`getItem`(*n*) retrieves the same value as the 4Test list method call *Lb*.`GetItemText`(*n*+1), but not the same value as *Lb*.`GetItemText`(*n*).

Handling Exceptions

This section provides help and troubleshooting information for handling exceptions.

Default Error Handling

If a test case fails, for example if the expected value doesn't match the actual value in a verification statement, by default Silk Test Classic calls its built-in recovery system, which:

- Terminates the test case.
- Logs the error in the results file.
- Restores your application to its default base state in preparation for the next test case.

These runtime errors are called exceptions. They indicate that something did not go as expected in a script. They can be generated automatically by Silk Test Classic, such as when a verification fails, when there is a division by zero in a script, or when an invalid function is called.

You can also generate exceptions explicitly in a script.

However, if you do not want Silk Test Classic to transfer control to the recovery system when an exception is generated, but instead want to trap the exception and handle it yourself, use the 4Test `do...except` statement.

Custom Error Handling

You can also use `do ... except` to perform some custom error handling, then use the re-raise statement to pass control to the recovery system as usual.

Example: do ... except

The Text Editor application displays a message box if a user searches for text that does not exist in the document. You can create a data-driven test case that verifies that the message box appears and that it displays the correct message. Suppose you want to determine if the Text Editor application is finding false matches, that is, if it is selecting

text in the document before displaying the message box. That means that you want to do some testing after the exception is raised, instead of immediately passing control to the recovery system. The following code sample shows how you can use `do ... except` to keep the control inside the test case:

```
testcase Negative (SEARCHINFO Data)
  STRING sMatch
  TextEditor.File.New.Pick ()
  DocumentWindow.Document.TypeKeys (Data.sText + Data.sPos)
  TextEditor.Search.Find.Pick ()
  Find.FindWhat.SetText (Data.sPattern)
  Find.CaseSensitive.SetState (Data.bCase)
  Find.Direction.Select (Data.sDirection)
  Find.FindNext.Click ()

  do
    MessageBox.Message.VerifyValue (Data.sMessage)
  except
    sMatch = DocumentWindow.Document.GetSelText ()

    if (sMatch != "")
      Print ("Found " + sMatch + " not " + Data.sPattern)
    reraise
  MessageBox.OK.Click ()

  Find.Cancel.Click ()
  TextEditor.File.Close.Pick ()
  MessageBox.No.Click ()
```

This following tasks are performed in the example:

- A test is performed after an exception is raised.
- A statement is printed to the results file if text was selected.
- The recovery system is called.
- The recovery system terminates the test case, logs the error, and restores the test application to its default base state.

As the example shows, following the do keyword is the verification statement, and following the except keyword are the 4Test statements that handle the exception. The exception-handling statements in this example perform the following tasks:

- Call the `GetSelText` method to determine what text, if any, is currently selected in the document.
- If the return value from the `GetSelText` method is not an empty string, it means that the application found a false match.
- If the application found a false match, print the false match and the search string to the results file.
- Re-raise the exception to transfer control to the recovery system.
- Terminate the test case.

The reraise statement raises the most recent exception again and passes control to the next exception handler. In the preceding example, the reraise statement passes control to the built-in recovery system. The reraise statement is used in the example because if the exception-handling code does not explicitly re-raise the exception, the flow of control passes to the next statement in the test case.

Trapping the exception number

Each built-in exception has a name and a number (they are defined as an enumerated data type, EXCEPTION). For example, the exception generated when a verify fails is `E_VERIFY ( 13700)`, and the exception generated when there is a division by zero is `E_DIVIDE_BY_ZERO ( 11500)`.

All exceptions are defined in `4test.inc`, in the directory where you installed Silk Test Classic.

You can use the `ExceptNum` function to test for which exception has been generated and, perhaps, take different actions based on the exception. You would capture the exception in a `do...except` statement then check for the exception using `ExceptNum`.

For example, if you want to ignore the exception `E_WINDOW_SIZE_ INVALID`, which is generated when a window is too big for the screen, you could do something like this:

```
do
Open.Invoke ()
except
if (ExceptNum () != E_WINDOW_SIZE_INVALID)
 reraise
```

If the exception is not `E_WINDOW_SIZE_INVALID`, the exception is reraised (and passed to the recovery system for processing). If the exception is `E_ WINDOW_SIZE_INVALID`, it is ignored.

Defining your own exceptions

In addition to using built-in exceptions, you can define your own exceptions and generate them using the `raise` statement.

Consider the following testcase:

```
testcase raiseExample ()
   STRING sTestValue = "xxx"
   STRING sExpected = "yyy"
   TestVerification (sExpected, sTestValue)

TestVerification (STRING sExpected, STRING sTestValue)
  if (sExpected == sTestValue)
    Print ("Success!")
  else
    do
      raise 1, "{sExpected} is different than {sTestValue}"
    except
print ("Exception number is {ExceptNum()}")
      reraise
```

The `TestVerification` function tests two strings. If they are not the same, they raise a user-defined exception using the raise statement.

Raise Statement

The `raise` statement takes one required argument, which is the exception number. All built-in exceptions have negative numbers, so you should use positive numbers for your user-defined exceptions. raise can also take an optional second argument, which provides information about the exception; that information is logged in the results file by the built-in recovery system or if you call `ExceptLog`.

In the preceding testcase, `raise` is in a `do...except` statement, so control passes to the except clause, where the exception number is printed, then the exception is reraised and passed to the recovery system, which handles it the same way it handles built-in exceptions.

Here is the result of the testcase:

```
Testcase raiseExample - 1 error
Exception number is 1
yyy is different than xxx
Occurred in TestVerification at except.t(31)
Called from raiseExample at except.t(25)
```

Note that since the error was re-raised, the testcase failed.

Using do...except statements to trap and handle exceptions

Using do...except you can handle exceptions locally, instead of passing control to Silk Test Classic's built-in error handler (which is part of the recovery system). The statement has the following syntax:

```
do
<statements>
except
<statements>
```

If an exception is raised in the `do` clause of the statement, control is immediately passed to the except clause, instead of to the recovery system.

If no exception is raised in the do clause of the statement, control is passed to the line after the except clause. The statements in the except clause are not executed.

Consider this simple testcase:

```
testcase except1 (STRING sExpectedVal, STRING sActualVal)

do
 Verify (sExpectedVal, sActualVal)
 Print ("Verification succeeded")
except
 Print ("Verification failed")
```

This testcase uses the built-in function `Verify`, which generates an exception if its two arguments are not equivalent. In this testcase, if `sExpectedVal` equals `sActualVal`, no exception is raised, `Verification succeeded` is printed, and the testcase terminates. If the two values are not equal, Verify raises an exception, control immediately passes to the `except` clause (the first Print statement is not executed), and `Verification failed` is printed.

Here is the result if the two values "one" and "two" are passed to the testcase:

```
Testcase except1 ("one", "two") - Passed
Verification failed
```

The testcase passes and the recovery system is not called because you handled the error yourself.

You handle the error in the except clause. You can include any 4Test statements, so you could, for example, choose to ignore the error, write information to a separate log file, and log the error in the results file.

Programmatically Logging an Error

Test cases can pass, even though an error has occurred, because they used their own error handler and did not specify to log the error. If you want to handle errors locally and generate an error (that is, log an error in the results file), you can do any of the following:

- After you have handled the error, re-raise it using the `reraise` statement and let the default recovery system handle it.
- Call any of the following functions in your script:

LogError (string, [cmd-line])	Writes string to the results file as an error (displays in red or italics, depending on platform) and increments the error counter. This function is called automatically if you don't handle the error yourself. `cmd-line` is an optional string expression that contains a command line.
LogWarning (string)	Same as `LogError`, except it logs a warning, not an error.
ExceptLog ( )	Calls `LogError` with the data from the most recent exception.

Performing More than One Verification in a Test Case

If the verification fails in a test case with only one verification statement, usually an exception is raised and the test case is terminated. However, if you want to perform more than one verification in a test case, before the test case terminates, this approach would not work.

Classic Agent Example

For example, see the following sample test case:

```
testcase MultiVerify ()
  TextEditor.Search.Find.Pick ()
  Find.VerifyCaption ("Find")
  Find.VerifyFocus (Find.FindWhat)
  Find.VerifyEnabled (TRUE)
  Find.Cancel.Click ()
```

The test case contains three verification statements. However, if the first verification, `VerifyCaption`, fails, an exception is raised and the test case terminates. The second and the third verification are not executed.

To perform more than one verification in a test case, you can trap all verifications except the last one in a `do...except` statement, like the following sample for the Classic Agent shows:

```
testcase MultiVerify2 ()
   TextEditor.Search.Find.Pick ()
   do
      Find.VerifyCaption ("Find")
   except
      ExceptLog ()
   do
      Find.VerifyFocus (Find.FindWhat)
   except
      ExceptLog ()
   Find.VerifyEnabled (TRUE)
   Find.Cancel.Click ()
```

All the verifications in this example are executed each time that the test case is run. If one of the first two verifications fails, the 4Test function `ExceptLog` is called. The `ExceptLog` function logs the error information in the results file, then continues the execution of the script.

Open Agent Example

For example, you might want to print the text associated with the exception as well as the function calls that generated the exception. The following test case illustrates this:

```
testcase VerifyTest ()
   STRING sTestValue = "xxx"
```

```
   STRING sExpectedValue = "yyy"
   CompValues (sExpectedValue, sTestValue)

CompValues (STRING sExpectedValue, STRING sTestValue)
   do
      Verify (sExpectedValue, sTestValue)
   except
      ErrorHandler ()

ErrorHandler ()
   CALL Call
   LIST OF CALL lCall
   lCall = ExceptCalls ()
   Print (ExceptData ())
   for each Call in lCall
      Print("Module: {Call.sModule}",
      "Function: {Call.sFunction}",
      "Line: {Call.iLine}")
```

- The test case calls the user-defined function `CompValues`, passing two arguments.
- `CompValues` uses `Verify` to compare its arguments. If they are not equal, an exception is automatically raised.
- If an exception is raised, `CompValues` calls a user-defined function, `ErrorHandler`, which handles the error. This is a general function that can be used throughout your scripts to process errors the way you want.
- `ErrorHandler` uses two built-in exception functions, `ExceptData` and `ExceptCalls`.

  `Except Data` — All built-in exceptions have message text associated with them. `ExceptData` returns that text.

  `ExceptCalls` — Returns a list of the function calls that generated the exception. You can see from `ErrorHandler` above, that `ExceptCalls` returns a `LIST OF CALL`. `CALL` is a built-in data type that is a record with three elements:

  - `sFunction`
  - `sModule`
  - `iLine`

  `ErrorHandler` processes each of the calls and prints them in the results file.

- Silk Test Classic also provides the function `ExceptPrint`, which combines the features of `ExceptCalls`, `ExceptData`, and `ExceptNum`.

```
Testcase VerifyTest - Passed
*** Error: Verify value failed - got "yyy", expected "xxx"
Module: Function: Verify Line: 0
Module: except.t Function: CompValues Line: 121
Module: except.t Function: VerifyTest Line: 112
```

The second line is the result of printing the information from `ExceptData`. The rest of the lines show the processing of the information from `ExceptCalls`.

This test case passes because the error was handled locally and not re-raised.

Writing an Error-Handling Function

If you want to customize your error processing, you will probably want to write your own error-handling function, which you can reuse in many scripts.

Open Agent Example

For example, you might want to print the text associated with the exception as well as the function calls that generated the exception. The following test case illustrates this:

```
testcase VerifyTest ()
   STRING sTestValue = "xxx"
   STRING sExpectedValue = "yyy"
   CompValues (sExpectedValue, sTestValue)

CompValues (STRING sExpectedValue, STRING sTestValue)
   do
      Verify (sExpectedValue, sTestValue)
   except
      ErrorHandler ()

ErrorHandler ()
   CALL Call
   LIST OF CALL lCall
   lCall = ExceptCalls ()
   Print (ExceptData ())
   for each Call in lCall
      Print("Module: {Call.sModule}",
      "Function: {Call.sFunction}",
      "Line: {Call.iLine}")
```

- The test case calls the user-defined function `CompValues`, passing two arguments.
- `CompValues` uses `Verify` to compare its arguments. If they are not equal, an exception is automatically raised.
- If an exception is raised, `CompValues` calls a user-defined function, `ErrorHandler`, which handles the error. This is a general function that can be used throughout your scripts to process errors the way you want.
- `ErrorHandler` uses two built-in exception functions, `ExceptData` and `ExceptCalls`.

`Except Data`	All built-in exceptions have message text associated with them. `ExceptData` returns that text.
`ExceptCalls`	Returns a list of the function calls that generated the exception. You can see from `ErrorHandler` above, that `ExceptCalls` returns a `LIST OF CALL`. `CALL` is a built-in data type that is a record with three elements: • `sFunction` • `sModule` • `iLine` `ErrorHandler` processes each of the calls and prints them in the results file.

- Silk Test Classic also provides the function `ExceptPrint`, which combines the features of `ExceptCalls`, `ExceptData`, and `ExceptNum`.

```
Testcase VerifyTest - Passed
*** Error: Verify value failed - got "yyy", expected "xxx"
Module: Function: Verify Line: 0
Module: except.t Function: CompValues Line: 121
Module: except.t Function: VerifyTest Line: 112
```

  The second line is the result of printing the information from `ExceptData`. The rest of the lines show the processing of the information from `ExceptCalls`.

  This test case passes because the error was handled locally and not re-raised.

Exception Values

This section describes the exceptions that are generated by Silk Test Classic under specific error conditions.

Exception value	Description
E_ABORT	Script aborted by user.
E_APP_NOT_READY	The application is not ready.
E_APP_NOT_RESPONDING	The application is not responding to input.
E_APPID_INVALID	The specified application ID is not a valid application.
E_BITMAP_NOT_STABLE	The bitmap timeout period set with OPT_BITMAP_MATCH_TIMEOUT was reached before the image stabilized.
E_BITMAP_REGION_INVALID	The specified region was off the screen.
E_BITMAPS_DIFFERENT	The comparison failed when comparing two bitmaps.
E_CANT_CLEAR_SELECTION	The selection cannot be cleared.
E_CANT_CLOSE_WINDOW	The window cannot be closed (often resulting when a confirmation dialog box pops up).
E_CANT_COMPARE_BITMAP	Silk Test Classic ran out of a system resource (such as memory) needed to compare the bitmaps.
E_CANT_CONVERT_RESOURCE	The specified resource cannot be handled by GetResource, although it is a valid resource for the widget.
E_CANT_EXIT_APP	Silk Test Classic was unable to close the application.
E_CANT_EXTEND_SELECTION	The list box selection can not be extended because nothing is selected.
E_CANT_MAXIMIZE_WINDOW	The window can not be maximized.
E_CANT_MINIMIZE_WINDOW	The window can not be minimized.
E_CANT_MOVE_WINDOW	The window can not be moved.
E_CANT_RESTORE_WINDOW	The window size can not be restored.
E_CANT_SET_ACTIVE	The window can not be set active.
E_CANT_SET_FOCUS	The window can not be given the input focus.
E_CANT_SIZE_WINDOW	The window can not be resized.

Exception value	Description
E_CANT_START_APP	The application cannot be started.
E_COL_COUNT_INVALID	The specified value is not a valid character count.
E_COL_NUM_INVALID	The specified value is not a valid character position.
E_COL_START_EXCEEDS_END	The starting character exceeds the end character position.
E_COLUMN_INDEX_INVALID	The specified index is not a valid column index. All `DataGrid` methods that use *DataGridCell*, *DataGridRow*, or *DataGridCol* as a parameter may see this exception raised.
E_COLUMN_NAME_INVALID	The specified index is not a valid column index. All `DataGrid` methods that use *DataGridCell*, *DataGridRow*, or *DataGridCol* as a parameter may see this exception raised.
E_CONTROL_NOT_RESPONDING	The control is not responding. Raised after checking whether a specified action took place.
E_COORD_OFF_SCREEN	The specified mouse coordinate is off the screen.
E_COORD_OUTSIDE_WINDOW	The specified coordinate is outside the window. This exception is never raised if the *OPT_VERIFY_COORD* option is set to `FALSE`.
E_CURSOR_TIMEOUT	The cursor timeout period was reached before the correct cursor appeared.
E_DELAY_INVALID	The specified delay is not valid.
E_FUNCTION_NOT_REGISTERED	The function called is a user-defined function that hasn't been registered by the application.
E_GRID_HAS_NO_COL_HDR	The specified `DataGrid` has no column header. All `DataGrid` methods that use *DataGridCell*, *DataGridRow*, or *DataGridCol* as a parameter may see this exception raised.
E_GUIFUNC_ID_INVALID	The specified function is not a valid function.
E_INTERNAL	Internal Silk Test Classic error.
E_INVALID_REQUEST	Invalid argument count or argument, or wrong number of arguments.
E_ITEM_INDEX_INVALID	The specified index is not a valid item index.
E_ITEM_NOT_FOUND	The specified item was not found.
E_ITEM_NOT_VISIBLE	The specified item is not visible.
E_KEY_COUNT_INVALID	The repeat count used in the key specification is not a valid number.
E_KEY_NAME_INVALID	The specified key name is not valid.
E_KEY_SYNTAX_ERROR	The syntax used in the key specification is not valid.
E_LINE_COUNT_INVALID	The specified line count is not valid.
E_LINE_NUM_INVALID	The specified line number is not valid.

Exception value	Description
E_LINE_START_EXCEEDS_END	The specified start line exceeds the end line number.
E_MOUSE_BUTTON_INVALID	The specified mouse button is not valid
E_NO_ACTIVE_WINDOW	No window is active.
E_NO_COLUMN	`GuptaTable` exception.
E_NO_DEFAULT_PUSHBUTTON	The dialog box does not have a default button.
E_NO_FOCUS_WINDOW	No window has the input focus.
E_NO_SETFOCUS_CELL	`GuptaTable` exception.
E_NO_SETFOCUS_COLUMN	`GuptaTable` exception.
E_NO_SETTEXT_CELL	`GuptaTable` exception.
E_NOFOCUS_CELL	No cell in the Gupta table has input focus.
E_NOFOCUS_COLUMN	No column in the Gupta table has input focus.
E_NOFOCUS_ROW	No row in the Gupta table has input focus.
E_NOT_A_TABLEWINDOW	The specified window is not a Gupta table.
E_OPTION_CLASS_MAP_INVALID	The mapping specified with the *OPT_CLASS_MAP* option is not valid.
E_OPTION_EVTSTR_LENGTH	The length of the event string given in *OPT_MENU_INVOKE_POPUP* was too long.
E_OPTION_NAME_INVALID	The specified agent option does not exist.
E_OPTION_TOO_MANY_TAGS	The maximum number of tags was exceeded when specifying buttons and menu items using one or more of these options: • *OPT_CLOSE_CONFIRM_BUTTONS* • *OPT_CLOSE_WINDOW_BUTTONS* • *OPT_CLOSE_WINDOW_MENUS*
E_OPTION_TYPE_MISMATCH	Mismatch between type of agent option and type of specified value.
E_OPTION_VALUE_INVALID	The specified agent option is not valid.
E_OUT_OF_MEMORY	The system has run out of memory.
E_POS_INVALID	The specified position is not valid.
E_POS_NOT_REACHABLE	The specified position cannot be reached. It is out of range of the object.
E_RESOURCE_NOT_FOUND	The widget does not contain the specified resource.
E_ROW_INDEX_INVALID	The specified index is not a valid row index. All `DataGrid` methods that use *DataGridCell*, *DataGridRow*, or *DataGridCol* as a parameter may see this exception raised.
E_SBAR_HAS_NO_THUMB	The scroll bar thumb can not be clicked to scroll a page because the scroll bar does not have a thumb.
E_SQLW_BAD_COLUMN_NAME	A bad column name was specified for the Gupta table.

Exception value	Description
E_SQLW_BAD_COLUMN_NUMBER	A bad column number was specified for the Gupta table.
E_SQLW_BAD_ROW_NUMBER	A bad row number was specified for the Gupta table.
E_SQLW_CANT_ENTER_TEXT	`GuptaTable` exception.
E_SQLW_INCORRECT_LIST	`GuptaTable` exception.
E_SQLW_NO_EDIT_WINDOW	`GuptaTable` exception.
E_SQLW_TABLE_WINDOW_HIDDEN	`GuptaTable` exception.
E_SQLW_TOO_BIG_LIST	`GuptaTable` exception.
E_SYSTEM	A system operation has failed.
E_TAG_SYNTAX_ERROR	The tag syntax is not valid: invalid coordinate or index, multiple indices specified, the window part is not the last part of the tag, or the tilde (~) is not followed by a child window.
E_TIMER	The specified timer operation is redundant. For example, a pause operation specified for a stopped timer.
E_TRAP_NOT_SET	Attempted to clear a trap that was not set.
E_UNSUPPORTED	The specified method is not supported on the current platform.
E_VAR_EXPECTED	A function or method call has not passed a variable for a required parameter or an expression failed to specify a variable required by an operator.
E_VERIFY	User-specified verification failed.
E_WINDOW_INDEX_INVALID	The tag uses an invalid index number.
E_WINDOW_NOT_ACTIVE	The specified window is not active.
E_WINDOW_NOT_ENABLED	The specified window is not enabled.
E_WINDOW_NOT_EXPOSED	The specified window is not exposed.
E_WINDOW_NOT_FOUND	The specified window is not found. Raised by any method that operates on a window, except `Exists`.
E_WINDOW_NOT_UNIQUE	The specified identifier does not represent a unique window. Raised by any method that operates on a window. Affected by the value set with the *OPT_VERIFY_UNIQUE* option. If you receive this exception, you might try using a slightly modified tag syntax to refer to a window with a non-unique tag. You can either include an index number after the object, as in `Dbox ("Cancel[2]")`, or you can specify the window by including the text of a child that uniquely identifies the window, such as `Dbox/uniqueText/...`, where the unique text is the tag of a child of that window.
E_WINDOW_SIZE_INVALID	The window size is too big for the screen or it is negative.

Exception value	Description
E_WINDOW_TYPE_MISMATCH	The specified window is not valid for this method. Raised when the type of window used is not the type the method accepts.

Troubleshooting Java Applications

This section provides solutions for common reasons that might lead to a failure of the test of your standalone Java application or applet. If these do not solve the specific problem that you are having, you can enable your extension manually.

The test of your standalone Java application or applet may fail if the application or applet was not ready to test, the Java plug-in was not enabled properly, if there is a Java recognition issue, or if the Java applet does not contain any Java controls within the **JavaMainWin**.

Why Is My Java Application Not Ready To Test?

This functionality is supported only if you are using the Classic Agent.

If your Java application is not ready to test, enable the extension for the application and restart the application.

1. On the **Basic Workflow** bar, click **Enable Extensions**. The **Enable Extensions** dialog box opens.
2. On the **Enable Extensions** dialog box, select the Java application for which you want to enable extensions.
3. Click **OK**. The **Enable Extensions** dialog box closes.
4. Close and restart the Java application.
5. When the application has finished loading, click `Test`.

Why Can I Not Test a Java Application Which Is Started Through a Command Prompt?

This functionality is supported only if you are using the Classic Agent.

If you are starting your standalone Java application through a **Command Prompt** window, close and re-open the **Command Prompt** window when you restart your application.

If you have forgotten to close and re-open the **Command Prompt** window, use the **Basic Workflow** bar to enable the extension again, making sure that you close and re-open both your Java application and the **Command Prompt** window before you click **Test** on the **Test Extension Settings** dialog box.

1. On the **Basic Workflow** bar, click **Enable Extensions**. The **Enable Extensions** dialog box opens.
2. On the **Enable Extensions** dialog box, select the Java application for which you want to enable extensions.
3. On the **Extension Settings** dialog box, click **OK**.
4. Close your Java application and the **Command Prompt** window.
5. Open a **Command Prompt** and restart your application.
6. When the application has finished loading, click **Test**.

What Can I Do If My Java Application Not Contain Any Controls Below JavaMainWin?

This functionality is supported only if you are using the Classic Agent.

If your Java application (or applet) does not contain any Java children within `JavaMainWin`, your tests against the application will fail. However, you might configure the Java extension to prevent this kind of failure. Record against Java controls to make sure that the extension is enabled. For example, record a push button as a `JavaAWTPushButton` or a `JavaJFCPushButton`.

How Can I Enable a Java Plug-In?

This functionality is supported only if you are using the Classic Agent.

If the browser that you are using has a plug-in enabled, or if the applet uses a plug-in, you must check the **Java Plug-in** check box on the **Extension Settings** dialog box. In all other cases, uncheck the **Java Plug-in** check box.

In Internet Explorer, click **Tools** > **Internet Options** and then click the **Advanced** tab, to determine if Internet Explorer has a plug-in enabled. Scan the **Settings** list to see if a third party plug-in, such as Java (Sun), has been enabled.

What Can I Do If the Java Plug-In Check Box Is Not Checked?

This functionality is supported only if you are using the Classic Agent.

If a plug-in is enabled for the browser, and the applet is using a plug-in, but you did not check the **Java Plug-In** check box, check the **Java Plug-In** check box and enable the extension again.

1. On the **Basic Workflow** bar, click **Enable Extensions**. The **Enable Extensions** dialog box opens.
2. On the **Extension Settings** dialog box, make sure `DOM` is the **Primary Extension**.
3. Check the **Java Plug-in** check box.
4. Click **OK**.
5. Close and restart your Java application.
6. When the application has finished loading, click **Test**.

What Can I Do When I Am Testing an Applet That Does Not Use a Plug-In, But the Browser Has a Plug-In Loaded?

This functionality is supported only if you are using the Classic Agent.

When you are testing an applet that does not use a plug-in, but the browser has a plug-in loaded, disable the plug-in and enable the extension again.

1. In the browser that you are using, disable all plug-ins.
2. In the **Basic Workflow** bar, click **Enable Extensions** and enable the extension for the applet again.
3. In the **Extension Settings** dialog box, uncheck the **Java Plug-in** check box.

What Can I Do If the Silk Test Java File Is Not Included in a Plug-In?

If the `SilkTest_Java3.jar` file is not included in the `lib/ext` directory of the plug-in that you are using:

1. Locate the `lib/ext` directory of the plug-in that you are using and check if the `SilkTest_Java3.jar` file is included in this folder.
2. If the `SilkTest_Java3.jar` file is not included in the folder, copy the file from the `javaex` folder of the Silk Test installation directory into the`lib\ext` directory of the plug-in.

What Can I Do If Java Controls In an Applet Are Not Recognized?

Silk Test Classic cannot recognize any Java children within an applet if your applet contains only custom classes, which are Java classes that are not recognized by default, for example a frame containing only an image. For information about mapping custom classes to standard classes, see *Mapping Custom Classes to Standard Classes*. Additionally, you have to set the Java security privileges that are required by Silk Test Classic.

Multiple Machines Testing

This section provides help and troubleshooting information for testing on multiple machines.

Remote Testing and Default Browser

This functionality is supported only if you are using the Classic Agent.

Silk Test Classic uses the registry to determine which version of IE is installed on your machine; this means that the correct default browser is selected when you choose Internet Explorer 6 or Internet Explorer 7.

If you are doing remote testing, you will have to set up your default browser for your target test machine yourself. The reason this is because both versions of Internet Explorer use the same `domex.dll` and therefore Silk Test Classic does not select the default browser at all.

Setting Up the Recovery System for Multiple Local Applications

Problem

By default, the recovery system will only work for the single application assigned to the const `wMainWindow`. With distributed testing, you can get recovery on multiple applications by using `multitestcase` instead of `testcase`.

You might ask whether you can get the recovery system to work on multiple applications that are running locally using `multitestcase` locally. The answer is no; `multitestcase` is for distributed testing only.

But you can use the following solution instead, using `testcase`.

Solution

To get recovery for multiple local applications, set up your frame file to do the following:

1. Get standard `wMainWindow` declarations for each application. The easiest way is to select **File** > **New** > **Test Frame** for each application, then combine the `wMainWindow` declarations into a single frame file or include them with use.
2. Make the global `wMainWindow` a variable of type `WINDOW`, rather than a constant.
3. Assign one of the windows to `wMainWindow` as a starting point.
4. Create a LIST OF WINDOW and assign the `wMainWindow` identifier for each application you are dealing with to it.
5. Define a `TestcaseEnter` function so that you reassign the `wMainWindow` variable and call `SetAppState` on each `MainWin` in turn.
6. Define a `TestcaseExit` function so that you reassign the `wMainWindow` variable and call `SetBaseState` on each `MainWin` in turn.
7. Then use `DefaultBaseState`, or your own base state if you want, with each of your test cases. In your test case, use `SetActive` each time you switch from one application to the other.

Example

The example consists of two sample files. The sample files are for the Classic Agent. If you want to use the example with the Open Agent, you have to change the sample code. For the sample script file, see *two_apps.t*. For the sample include file, see `two_apps.inc`. The example uses two demo applications shipped with Silk Test Classic, the Text Editor and the Test Application. To see that the recovery system is working for both applications, turn on the two debugging options in **Runtime Options** and look at the transcript after running the test script.

The first test case has an intentional error in its last statement to demonstrate the recovery system. The test case also demonstrates how to move data from one application to another with `Clipboard.GetText` and `Clipboard.SetText`.

Because the recovery system is on, the DefaultBaseState will take care of invoking each application if it is not already running and will return to the DefaultBaseState after each test case, even if the test case fails.

You can print the sample files out or copy them to the **Clipboard**, then paste them into Silk Test Classic. You might have to do some cleanup where the indentation of lines is incorrect in the pasted file.

two_apps.t

The following sample script file for the Classic Agent shows how you can locally test multiple applications. To use the sample with the Open Agent, you have to change the sample code, for example you have to replace all tags with locators.

```
testcase Test1 () appstate DefaultBaseState
  //SetActive each time you switch apps
  TestApplication.SetActive()
  TestApplication.File.New.Pick ()
  MDIChildWindow1.TextField1.SetPosition (1, 1)
  MDIChildWindow1.TextField1.TypeKeys ("In Test Application MDI Child Window
#1.")
  //SetActive each time you switch apps
  TextEditor.SetActive ()
  TextEditor.File.New.Pick ()
  TextEditor.ChildWin("(untitled)[1]").TextField("#1")
    .TypeKeys ("In Text Editor untitled Document window.<Enter>")
  //SetActive each time you switch apps
  TestApplication.SetActive()
  LIST OF STRING lsTempStrings
  lsTempStrings = MDIChildWindow1.TextField1.GetMultiText()
  Clipboard.SetText([LIST OF STRING]lsTempStrings)
```

```
  //SetActive each time you switch apps
  TextEditor.SetActive()
  TextEditor.ChildWin("(untitled)
[1]").TextField("#1").SetMultiText(Clipboard.GetText(),2)
  TextEditor.VerifyCaption("FooBar")

testcase Test2 () appstate DefaultBaseState
  wMainWindow = TestApplication
  TestApplication.SetActive()
  TestApplication.File.New.Pick ()
  MDIChildWindow1.TextField1.SetPosition (1, 1)
  MDIChildWindow1.TextField1.TypeKeys ("In Test Application MDI Child Window
#1.")
  wMainWindow = TextEditor
  TextEditor.SetActive ()
  TextEditor.File.New.Pick ()
  TextEditor.ChildWin("(untitled)[1]").TextField("#1")
    .TypeKeys ("In Text Editor untitled Document window.<Enter>")
  wMainWindow = TestApplication
  TestApplication.SetActive()
  LIST OF STRING lsTempStrings
  lsTempStrings = MDIChildWindow1.TextField1.GetMultiText()
  Clipboard.SetText([LIST OF STRING]lsTempStrings)
  wMainWindow = TextEditor
  TextEditor.SetActive()
  TextEditor.ChildWin("(untitled)
[1]").TextField("#1").SetMultiText(Clipboard.GetText(),2)
```

two_apps.inc

The following sample include file for the Classic Agent shows how you can locally test multiple applications. To use the sample with the Open Agent, you have to change the sample code, for example you have to replace all tags with locators.

```
// two_apps.inc
// define wMainWindow as a window global var
// and assign one of the apps (your pick) as a starting point.
window wMainWindow = TextEditor
const wMainWindow = TextEditor //replace default def

// Create a list of app MainWins
list of window lwApps = {...}
TextEditor
TestApplication
// Define your own TestCaseEnter.
TestCaseEnter ()
     window wCurrentApp
     for each wCurrentApp in lwApps
          wMainWindow = wCurrentApp
          SetAppState()

// Define your own TestCaseExit.
TestCaseExit (BOOLEAN bException)
  if bException
    ExceptLog()
  window wCurrentApp
  for each wCurrentApp in lwApps
    wMainWindow = wCurrentApp
    if (wCurrentApp.Exists())
      SetBaseState()

window MainWin TextEditor
  tag "Text Editor"
```

```
// The working directory of the application when it is invoked
const sDir = "C:\QAP40"
// The command line used to invoke the application
const sCmdLine = "C:\PROGRAMFILES\<SilkTest install directory>\\SILKTEST
\TEXTEDIT.EXE"

// The first window to appear when the application is invoked
// const wStartup = ?

// The list of windows the recovery system is to leave open
// const lwLeaveOpen = {?}
Menu File
  tag "File"
  MenuItem New
    tag "New"
  MenuItem Open
    tag "Open"
  MenuItem Close
    tag "Close"
  MenuItem Save
    tag "Save"
  MenuItem SaveAs
    tag "Save As"
  MenuItem Print
    tag "Print"
  MenuItem PrinterSetup
    tag "Printer Setup"
  MenuItem Exit
    tag "Exit"
  Menu Edit
    tag "Edit"
  MenuItem Undo
    tag "Undo"
  MenuItem Cut
    tag "Cut"
  MenuItem Copy
    tag "Copy"
  MenuItem Paste
    tag "Paste"
  MenuItem Delete
    tag "Delete"
  Menu Search
    tag "Search"
  MenuItem Find
    tag "Find"
  MenuItem FindNext
    tag "Find Next"
  MenuItem Replace
    tag "Replace"
  MenuItem GotoLine
    tag "Goto Line"
  Menu Options
    tag "Options"
  MenuItem Font
    tag "Font"
  MenuItem Tabs
    tag "Tabs"
  MenuItem AutomaticIndent
    tag "Automatic indent"
  MenuItem CreateBackups
    tag "Create backups"
  Menu xWindow
    tag "Window"
```

```
  MenuItem TileVertically
    tag "Tile Vertically"
  MenuItem TileHorizontally
    tag "Tile Horizontally"
  MenuItem Cascade
    tag "Cascade"
  MenuItem ArrangeIcons
    tag "Arrange Icons"
  MenuItem CloseAll
    tag "Close All"
  MenuItem Next
    tag "Next"
  Menu Help
    tag "Help"
  MenuItem About
    tag "About"

window MessageBoxClass MessageBox
  tag "~ActiveApp/[DialogBox]$MessageBox"
  PushButton OK
    tag "OK"
  PushButton Cancel
    tag "Cancel"
  PushButton Yes
    tag "Yes"
  PushButton No
    tag "No"
  StaticText Message
    mswnt tag "#2"
    tag "#1"

window ChildWin Untitled
  tag "(untitled)"
  parent TextEditor
  TextField TextField1
    tag "#1"

window DialogBox Open
  tag "Open"
  parent TextEditor
  StaticText FileNameText
    tag "File Name:"
  TextField FileName1
    tag "File Name:"
  ListBox FileName2
    tag "File Name:"
  StaticText DirectoriesText
    tag "Directories:"
  StaticText CQap40Text
    tag "c:\qap40"
  ListBox CQap40
    tag "c:\qap40"
  StaticText ListFilesOfTypeText
    tag "List Files of Type:"
  PopupList ListFilesOfType
    tag "List Files of Type:"
  StaticText DrivesText
    tag "Drives:"
  PopupList Drives
    tag "Drives:"
  PushButton OK
    tag "OK"
  PushButton Cancel
    tag "Cancel"
```

```
  PushButton Network
    tag "Network"

window MainWin TestApplication
  tag "Test Application"
// The working directory of the application when it is invoked
const sDir = "C:\QAP40"

// The command line used to invoke the application
const sCmdLine = "C:\QAP40\TESTAPP.EXE"

// The first window to appear when the application is invoked
// const wStartup = ?

// The list of windows the recovery system is to leave open
// const lwLeaveOpen = {?}
Menu File
  tag "File"
  MenuItem New
    tag "New"
  MenuItem Close
    tag "Close"
  MenuItem Exit
    tag "Exit"
  MenuItem About
    tag "About"
Menu Control
  tag "Control"
  MenuItem CheckBox
    tag "Check box"
  MenuItem ComboBox
    tag "Combo box"
  MenuItem ListBox
    tag "List box"
  MenuItem PopupList
    tag "Popup list"
  MenuItem PushButton
    tag "Push button"
  MenuItem RadioButton
    tag "Radio button"
  MenuItem StaticText
    tag "Static text"
  MenuItem Scrollbar
    tag "Scrollbar"
  MenuItem Textfield
    tag "Textfield"
  MenuItem DrawingArea
    tag "Drawing area"
  MenuItem KeyboardEvents
    tag "Keyboard events"
  MenuItem Cursors
    tag "Cursors"
  MenuItem ListView
    tag "List view"
  MenuItem PageList
    tag "Page list"
  MenuItem StatusBar
    tag "Status bar"
  MenuItem ToolBar
    tag "Tool bar"
  MenuItem TrackBar
    tag "Track bar"
  MenuItem TreeView
    tag "Tree view"
```

```
  MenuItem UpDown
    tag "Up-Down"
Menu Menu
  tag "Menu"
  MenuItem TheItem
    tag "The item"
  MenuItem TheAcceleratorItem
    tag "The accelerator item"
  Menu TheCascadeItem
    tag "The cascade item"
    MenuItem Item1
      tag "Item1"
    MenuItem Item2
      tag "Item2"
  MenuItem Check
    tag "Check"
  MenuItem Uncheck
    tag "Uncheck"
  MenuItem TheCheckItem
    tag "The check item"
  MenuItem Enable
    tag "Enable"
  MenuItem Disable
    tag "Disable"
  MenuItem TheEnableItem
    tag "The enable item"
  Menu Submenu1
    tag "Submenu1"
    MenuItem Item1
      tag "Item1"
    MenuItem Item2
      tag "Item2"
  Menu Submenu2
    tag "Submenu2"
    MenuItem Item1
      tag "Item1"
    MenuItem Item2
      tag "Item2"
  Menu Submenu3
    tag "Submenu3"
    MenuItem Item1
      tag "Item1"
    MenuItem Item2
      tag "Item2"
    MenuItem ThePopupMenu
      tag "The popup menu"
    MenuItem Check
      tag "Check"
    MenuItem Uncheck
      tag "Uncheck"
    MenuItem TheCheckItem
      tag "The check item"
    MenuItem Enable
      tag "Enable"
    MenuItem Disable
      tag "Disable"
    MenuItem TheEnableItem
      tag "The enable item"
    MenuItem AddMenu
      tag "Add menu"
    MenuItem ClearMenus
      tag "Clear menus"
    Menu DisabledMenu
      tag "DisabledMenu"
```

```
    MenuItem Item1
      tag "Item1"
    MenuItem Item2
      tag "Item2"
    Menu Menu5
      tag "#5"
    MenuItem MenuItem1
      tag "#1"
    MenuItem MenuItem2
      tag "#2"
  Menu xWindow
    tag "Window"
    MenuItem Cascade
      tag "Cascade"
    MenuItem Tile
      tag "Tile"
    MenuItem ArrangeIcons
      tag "Arrange Icons"
    MenuItem CloseAll
      tag "Close All"
    MenuItem ChangeCaption
      tag "Change Caption"
    MenuItem SysModal1
      tag "SysModal 1"
    MenuItem SysModal2
      tag "SysModal 2"
    MenuItem SysModal3
      tag "SysModal 3"
    MenuItem N1MDIChildWindow1
      tag "1 MDI Child Window #1"

window ChildWin MDIChildWindow1
  tag "MDI Child Window #1"
  parent TestApplication
  TextField TextField1
    tag "#1"
```

Objects

This section provides help and troubleshooting information for objects.

Does Silk Test Classic Support Oracle Forms?

This functionality is supported only if you are using the Classic Agent.

Silk Test Classic handles Oracle Forms applications as any Java applet that consists of custom classes.

All children of the applet are seen as `CustomWins`, with native class names such as 'oracle.ewt.*' and 'oracle.forms.*'. You need to declare winclasses for any classes that you plan to use, and you can only interact with classes through scripting. For more efficient declaration of classes, use the `CaptureAllClasses`() function instead of clicking **Record** > **Class** to record each class separately.

As with any application consisting of custom classes, if there are objects that Silk Test Classic does not see, check **Show All Classes** to see if that exposes the ignored objects. If so, then you should add those classes to the `[ClassList]` section of `extend\JavaEx.ini`. Uncheck **Show All Classes** before recording window classes or declarations.

To get started, take a look at our guidelines for when and how to record classes.

If you do not want to record classes for these `CustomWin` objects, you can click **Record** > **Class** and then uncheck the **Show All Classes** check box in the lower left corner of the dialog box.

Mouse Clicks Fail on Certain JFC and Visual Café Objects

This functionality is supported only if you are using the Classic Agent.

Because of timing issues between the Agent and the application under test, you might experience problems with menu picks on Java Foundation Class (JFC) or Symantec Visual Café objects. As a workaround, try setting keyboard and mouse delays. Use the values specified below as good starting points, and then experiment with different delays as needed until you find the timing that works best for your system configuration and application.

For …	Set …
Java Foundation Class objects	keyboard delay = 0.01 second mouse delay = 0.01 second
Visual Café objects	keyboard delay = 0.03 second mouse delay = 0.03 second

My Sub-Menus of a Java Menu are being Recorded as JavaDialogBoxes

This functionality is supported only if you are using the Classic Agent.

Occasionally, a sub-menu of a Java menu exceeds the boundaries of its application so that part of the sub-menu extends beyond the borders of the application. In this situation, Silk Test Classic records these very large sub-menus as `JavaDialogBoxes`, not as part of the menu.

Try dragging the `JavaMainWin` to a larger size before recording or maximizing the application.

Other Problems

This section provides help and troubleshooting information for problems that are not covered by another section.

Adding a Property to the Recorder

1. Write a method.
2. Add a property to the class.
3. Add the property to the list of property names.

For example, if you have a text field that is `ReadOnly` and you want to add that property to the recorder you can do the following:

1. Write the method `Boolean IsReadOnly()` for the `TextField` class.
2. Add the property, `bReadOnly` to the class.
3. Add `bReadOnly` to the list of property names.
4. Compile. `bReadOnly` will appear in the **Recorder** after you compile.

```
Winclass TextField : TextFieldBOOLEAN IsReadOnly()
STRING sOriginalText = this.GetText()
STRING sNewText = "xxx"
this.SetText(sNewText)
if this.GetText()==sOriginalText
return TRUE
```

```
else
return FALSE
property bReadOnly
BOOLEAN Get()
return this.IsReadOnly()
LIST OF STRING IsPropertyNames = {…}
"bReadOnly"
```

Application Hangs When Playing Back a Menu Item Pick

This functionality is supported only if you are using the Classic Agent.

Problem

Your application under test (AUT) hangs when playing back a `Pick()` method call against a menu item or menu.

Solution

Try setting the agent option `OPT_PLAY_MODE` to `Win32`:

```
Agent.SetOption (OPT_PLAY_MODE, "Win32")
```

This option is not part of the **Agent Options** dialog box, so you must set it by scripting. To set it globally, create a `TestCaseEnter()` function and set it there.

Cannot Access Some of the Silk Test Classic Menu Commands

This functionality is supported only if you are using the Classic Agent.

Problem

You cannot use several of the menus/menu commands.

Solution

You may be using Silk Test Runtime, a stripped down version of Silk Test Classic. To check what version you are using, click **Help** > **About**.

Cannot Double-Click a Silk Test Classic File and Open Silk Test Classic

Problem

Silk Test Classic does not open automatically when you double-click a `.t`, `.inc`, `.s`, `.g.t`, `.pln`, `.res`, `.stp`, or `.vtp` file.

Cause

During the install process, Silk Test Classic is associated with these file types. However if these file type associations have been changed after Silk Test Classic setup, these file types may not be opened with Silk Test Classic when double-clicking such a file.

Note: File type associations are only available for Microsoft Windows platforms.

Solution

You can either manually associate these file types with Silk Test Classic in Windows, under **Start** > **Settings** > **Control Panel** > **Folder Options**, or reinstall Silk Test Classic.

Cannot Extend AnyWin, Control, or MoveableWin Classes

The `AnyWin`, `Control`, and `MoveableWin` classes are logical (virtual) classes that do not correspond to any actual GUI objects, but instead define methods common to the classes that derive from them. This means that Silk Test Classic never records a declaration that has one of these classes.

Furthermore, you cannot extend or override logical classes. If you try to extend a logical class, by adding a method, property or data member to it, that method, property, or data member is not inherited by classes derived from the class. You will get a compilation error saying that the method, property, or data member is not defined for the window that tries to call it.

You can also not override the class, by rewriting existing methods, properties, or data members. Your modifications are not inherited by classes derived from the class.

Cannot Find the Quick Start Wizard

This functionality is supported only if you are using the Classic Agent.

Problem

You cannot find the **Quick Start Wizard**.

Solution

Beginning with Silk Test Classic 6.0, the **Quick Start Wizard** is turned off by default. You can turn the **Wizard** back on by opening `partner.ini` and replacing these two lines:

```
[Wizard]
AutoInitWizard=FALSE
```

With the following:

```
[Wizard]
AutoInitWizard=TRUE
WizardEnabled=TRUE
```

Restart Silk Test Classic. The Wizard is available when you click **File** > **New** > **Testframe**.

Cannot open results file

Problem

Silk Test Classic crashes while running a script and reports the error `Can't open results file`.

Solution

While Silk Test Classic is running a script, it temporarily stores results in a journal file (`.jou`) which is converted to a `.res` file when the script finishes running.

Delete all `.jou` files in the same directory as the script. (You do not have to delete your results files.)

Restart Silk Test Classic and run your script again.

Cannot Play Back Picks of Cascaded Sub-Menus for an AWT Application

This functionality is supported only if you are using the Classic Agent.

For AWT, Silk Test Classic can pick cascaded menu items only up to the third level, for example:

```
TestApplication.Menu.TheCascadeItem.Su\menu1.Item2
```

It cannot pick deeper sub-menus, such as the following:

```
TestApplication.Menu.TheCascadeItem.Su\menu1.Submenu2.Item2
```

For JVM 1.1.x, Silk Test Classic may only be able to pick menu items up to the 2nd level, for example:

```
TestApplication.Menu.TheCascadeItem.It\m2
```

Cannot Record Second Window

This functionality is supported only if you are using the Classic Agent.

When Silk Test Classic records a popup list or list box select that causes an `OnMouseDown` event to spawn a second window, it cannot record that second window. This is usually due to the Javascript that launches a second window and changes the underlying value of the popup list or list box.

You must edit the recorded scripts in order for play back to run correctly.

Common DLL Problems

This functionality is supported only if you are using the Classic Agent.

Here are some issues that could come up if you are calling DLL functions in a script.

Error in results file: dll not found

This usually means that your path does not include the directory containing the DLL. If you are running remotely, make sure that the path on the machine running the agent includes the DLL directory.

Error after compile: dll not found

In the DLL declaration, use the fully qualified path of the DLL, not just the file name.

Error in results file: function <name> not found in dll

The most likely scenario is that the DLL is a C++ library and the function name has been mangled. To use functions in a C++ library, you need to wrap the functions with the C wrapper and recompile. Then Silk Test Classic can access the function in the library.

If this is not the problem, there might be a typo in the function name in the DLL.

Error in results file: dll could not be loaded

Make sure the directory containing the DLL is on the path.

Warning in results file: String buffer size was increased from x to 256 characters

If the user calls a DLL function with an output string buffer that is less then the minimum size of 256 characters, the original string buffer is resized to 256 characters and a warning is printed. This warning, `String buffer size was increased from x to 256 characters` (where x is the length of the

given string plus one) alerts the user to a potential problem where the buffer used might be shorter than necessary.

Difficulty creating DLLs to use with Silk Test

Only specific data types are compatible with 4Test. These data types are listed in *C data types for DLL functions*.

If your DLL calls have data types not supported by 4Test, then the functions must be wrapped such that only compatible data types are used for the return type and arguments of the function. Any data types can be used inside the DLL function.

Common Scripting Problems

Here are some common problems that occur with scripts.

Typographical errors

It is very easy to make typographical errors that the 4Test compiler cannot catch. If a line of code does nothing, this might be the problem.

Global variables with unexpected values

When you write a function that uses global variables, make sure that each variable has an appropriate value when the function exits. If another function uses the same variable later, and it has an unexpected value on entry to the function, an error could occur.

To check that a variable has a reasonable value on entry to a function, set a breakpoint on the line that calls the function and use the command **View** > **Global Variables** to check the variable's value.

Uninitialized variables

Silk Test Classic does not initialize variables for you. So if you have not initialized a variable on entry to a function, it will have the value `<unset>`. It is better to explicitly give a value to a variable than to trust that another function has already initialized it for you. Also, remember that 4Test does not keep local variables around after a function exits; the next time the function is called, its local variables could be uninitialized.

If you are in doubt about whether a variable has a reasonable value at a particular point, set a breakpoint there and use **View** > **Global Variables** or **View** > **Local Variables** to check the variable's value.

Global and local variables with the same name

It is usually not good programming practice to give different variables the same names. If a global and local variable with the same name are in scope (accessible) at the same time, your code can only access the local variable.

To check for repeated names, use **View** > **Local Variables** and **View** > **Global Variables** to see if two variables with the same name are in scope simultaneously.

Incorrect values for loop variables

When you write a for loop or a while loop, be sure that the initial, final, and step values for the variable that controls the loop are correct. Incrementing a loop variable one time more or less than you really want is a common source of errors.

To make sure a control loop works as you expect, use **Debug** > **Step** Into to step through the execution of the loop one statement at a time, and watch how the value of the loop variable changes using **View** > **Local Variables**.

Checking the precedence of operators

The order in which 4Test applies operators when it evaluates an expression may not be what you expect. Use parentheses, or break an expression down into intermediate steps, to make sure it works as expected. You can use View/Expression to evaluate an expression and check the result.

Incorrect uses of break statements

A break statement transfers control of the script out of the innermost nested for, for each, while, switch, or select statement only. In other words, break exits from a single loop level, not from multiple levels. Use **Debug** > **Step** Into to step through the script one line at a time and ensure that the flow of control works as you expect.

Infinite loops

To check for infinite loops, step through the script with **Debug** > **Step Into**.

Code that never executes

To check for code that never executes, step through the script with **Debug** > **Step Into**.

Conflict with Virus Detectors

Problem

Silk Test Classic will occasionally have problems on machines running virus detectors that use heuristic or algorithmic virus detection in addition to the standard pattern recognition. What happens is that while Silk Test Classic is running, the virus detector identifies Silk Test Classic as displaying "virus-like" behavior, and kills or otherwise disables the agent. This leads to unpredictable and inconsistent behavior in Silk Test Classic, including loss of communications with the agent and inconsistent test results or object recognition.

Solution

To avoid this problem the only solution is to temporarily disable the virus detector while Silk Test Classic is running.

Displaying the Euro Symbol

Problem

You want to display the Euro (€) symbol.

Solution

Download a Euro-enabled font from Microsoft. Double check that you can see the Euro symbol by opening Notepad on the machine where you installed the font and entering the ASCII code for the Euro symbol. As long as you see the symbol in notepad, you should be able to see it within Silk Test Classic.

In Silk Test Classic, click **Options** > **Editor Font** and be sure that your font is set to Arial, Courier New, or Times New Roman.

Do I Need Administrator Privileges to Run Silk Test Classic?

You require the following privileges to install or run Silk Test Classic:

- To install Silk Test Classic, you must have local administrator privileges.
- To install Silk Test Classic on a Windows server, you must have domain-level administrator privileges.
- To run Silk Test Classic with the Classic Agent, you must have administrator privileges.
- To run Silk Test Classic with the Open Agent, you must have administrator privileges, if you have installed Silk Test Classic into the `Program Files` folder.
- To run Silk Test Classic with the Open Agent, you do not need to have administrator privileges, if you have installed Silk Test Classic into a different location than the `Program Files` folder.

Note: If User Account Control (UAC) is activated on your system, we recommend that you install Silk Test Classic into a different location than the `Program Files` folder.

General Protection Faults

Problem

When recording or running tests, you get a `General Protection Fault (GPF)` or `Invalid Page Fault (IPF)` in `agent.exe` or `partner.exe`.

Solution

It can be very difficult to pin down the cause of these problems. It might involve a combination of your machine's configuration, other applications that are running, and the network's configuration. The best approach is to gather the diagnostic information described below and send it to Technical Support with a detailed description of what scenario led to the error.

Capture the system diagnostics	When the system error message displays, chose the option to capture detailed information on the error. Write the information down.
Capture a debug.log file	**1.** Ensure that no Silk Test Classic or Agent processes are running. **2.** Open a DOS prompt window. **3.** Change your working directory to your Silk Test Classic installation directory. **4.** Delete or rename `c:\debug.log` if the file exists. **5.** Set the following environment variable: `set QAP_DEBUG_AGENT=1`. **6.** Start the Agent manually: `start .\agent`. **7.** Start Silk Test Classic manually: start `.\partner`. **8.** Go through the scenario to reproduce the problem. **9.** The file `c:\debug.log` file will be created. **10.** Send this file as an attachment to your email to Technical Support.
Monitor CPU and RAM usage	When reproducing this error to gather the diagnostics above, also run a system resource monitor to check on CPU and RAM usage. Note whether CPU or RAM is being exhausted.
Note your system configuration	When sending in these diagnostics, note the version of Silk Test Classic, the operating system and version, and the machine configuration (CPU, RAM, disk space).

Running Global Variables from a Test Plan Versus Running Them from a Script

Problem

When running from a test plan, global variables don’t keep their value from one test case to another.

When test cases are run from a script, global variables are initialized once at the beginning and do not get reset while the script is being run. On the other hand, when you run test cases from a test plan, all global variables get re-initialized after each test case. This is because the Agent reinitializes itself before running each test case. Consequently, you may find that global variables are not as useful when running from a test plan.

Solution

A workaround is to use the `FileWriteLine` or `FileWriteValue` function to write the values of the global variables out to a file, then use the `FileReadLine` or `FileReadValue` function to read the value back into each variable in each test case.

Ignoring a Java Class

This functionality is supported only if you are using the Classic Agent.

If you are using the Java extension and you want to ignore a class, you must edit your `javaex.ini` file. Add the following line to your `javaex.ini` file:

```
myclass=FALSE
```

where `myclass` is the full class name of the class to be ignored.

Use the value `FALSE` and not the value `IGNORE`.

Include File or Script Compiles but Changes are Not Picked Up

Problem

You compile an include file or script, but changes that you made are not used when you run the script.

Solutions

Did you change the wrong include file?	Make sure that the include file you are compiling is the same as the file that is being used by the script. Just because you have an include file open and have just compiled it does not mean that it is being used by the script. The include file that the script will use is either specified in Runtime Options (Use Files field) or by a use statement in the script.
Is there a time-stamp problem?	If the time stamp for the file on disk is later than the machine time when you do **Run** > **Compile**, then the compile does not actually happen and no message is given. This can happen if two machines are sharing a file system where the files are being written out and the time on the machines is not synchronized. By default, Silk Test Classic only compiles files that need compiling, based on the date of the existing object files and the system clock. This way, you don't have to wait to recompile all files each time a change is made to one file. If you need to, you can force Silk Test Classic to compile all files by selecting **Run** > **Compile All**. **Run** > **Compile All** compiles the script or suite and all dependent include files, even if they have not changed since they were last compiled. It also compiles files listed in the **Use Files** field in the **Runtime Options** dialog and the compiler constants declared in the **Runtime Options** dialog. Finally, it compiles the include files loaded at startup, if needed.

Are your object files corrupted?

Sometimes a Silk Test Classic object (`.ino` or `.to`) file can become corrupted. Sometimes a corrupted object file can cause Silk Test Classic to assume that the existing compile is up to date and to skip the recompile without any message.

To work around this, delete all `.ino` and `.to` files in the directories containing the `.inc` and `.t` files you are trying to compile, then compile again.

Library Browser Not Displaying User-Defined Methods

Problem

You add a description for a user-defined method and a user-defined function to `4test.txt`. After restarting Silk Test Classic, the new description for the function displays in the Library Browser, but not the description for the method. So you know that the modified `4test.txt` file is being used, but your user-defined method is not being displayed in the **Library Browser**.

Solutions

Only methods defined in a class definition (that is, in your include file where your class is defined) will display in the **Library Browser**. For example, `MyAccept` will be displayed.

```
winclass DialogBox:DialogBox
Boolean MyAccept()
...
```

Methods you define for an individual object are not displayed in the **Library Browser**. For example, `MyDialogAccept` will not display.

```
DialogBox MyDialog
tag "My Dialog"
Boolean MyDialogAccept()
...
```

In order to display in the **Library Browser**, the description in your `4test.txt` file must have a return type that matches the return type in your include file declaration. If the `4test.txt` description has no `returns` statement, then the declaration must be for a return type of void (either specified explicitly or by defaulting to type void). Otherwise, the description will not display in the **Library Browser**.

For more information about adding information to the **Library Browser**, see *Adding to the Library Browser*.

Maximum Size of Silk Test Classic Files

The following size limits apply:

- The limit for `.inc`, `.t`, and `.pln` files (and their associated backup files, `.*_`) is 64K lines.
- The size limit for the corresponding object files (`.*o`) depends on the amount of available system memory.
- The Silk Test Classic editor limits lines to 1024 characters.
- The maximum size of a single entry in a .res file is 64K.
- Test case names can have a maximum of 127 characters. When you create a data-driven test case, Silk Test Classic truncates any test case name that is greater than 124 characters.

Playing Back Mouse Actions

This functionality is supported only if you are using the Classic Agent.

Under 32-bit Windows, the following methods take an optional BOOLEAN argument, `bRawEvent`, that specifies how mouse actions are played back:

AnyWin methods	• Click • DoubleClick • MoveMouse • MultiClick • PressMouse • ReleaseMouse
Pushbutton method	• Click

By default, `bRawEvent` is FALSE. When FALSE, Silk Test Classic uses the standard Windows messaging mechanism (journal playback) to perform actions. Usually this works fine. If your test plays back correctly, use the default.

There are times, however, when this doesn't work and your test won't play back correctly. In such situations, set `bRawEvent` to TRUE. When TRUE, Silk Test Classic uses a low-level mechanism to perform the actions. Operations involving mouse dragging are more likely to work correctly using the low-level mechanism. But this mechanism hasn't been tested as thoroughly as journal playback, so you should use it only when the default fails.

You can have all playback use the low-level mechanism by setting OPT_PLAY_MODE to Win32:

```
Agent.SetOption (OPT_PLAY_MODE, "Win32")
```

To turn this off, set OPT_PLAY_MODE to Normal:

```
Agent.SetOption (OPT_PLAY_MODE, "Normal")
```

Recorder Does Not Capture All Actions

Problem

While recording, the Silk Test Recorder does not capture all actions in your application under test, though you complete the actions.

Cause

The application under test may be "going too fast" and the Silk Test Recorder may not be able to keep up.

Solution

Slow down the interactions with your application while recording. Record a test case at the speed of the Silk Test Recorder.

Recording two SetText () Statements

This functionality is supported only if you are using the Classic Agent.

While using the **Record Actions** dialog box, you capture entering `John` into a text field. Silk Test Classic may record the following statements:

```
<identifier>.SetText ("J")
<identifier>.SetText ("John")
```

This is not an error. The recorder may capture several `SetText` statements without impacting playback.

Relationship between Exceptions Defined in 4test.inc and Messages Sent To the Result File

Silk Test Classic calls `LogError` automatically when it raises an exception that you have not handled. By reading `4test.inc` you can find that Silk Test Classic has a list of exceptions like:

```
E_ABORT = -10100,
E_TBL_HAS_NO_ROW_HDR = -30100,
E_WINDOW_NOT_FOUND = -27800
```

Since exception numbers can apply to more than one exception, it can be helpful to query on a particular exception number via `ExceptNum()` to decide how to handle an error. If you need to query on a specific exception message, you can use `ExceptData()`. We recommend using `MatchStr()` with `ExceptData()`.

To find the `E_...` constant for any 4Test exception, you can use:

```
[-] do
    <code that causes exception>
[-] except
[ ] LogWarning ("Exception number: {[EXCEPTION]ExceptNum ()}")
[ ] reraise
```

This will print out the exception constant in the warning.

Be sure to remove the `LogWarning do..except` block after you have found the `E_...` constant.

The 4Test Editor Does Not Display Enough Characters

Problem

While you can edit 4Test files outside of Silk Test Classic and create lines with more than 1024 characters, the Silk Test 4Test Editor (4Test Editor) does not let you edit or extend these lines.

The line limit of the 4Test Editor is 1024 characters.

Solution

Use the `<Shift+Enter>` continuation character to break the line into smaller lines.

Silk Test Classic Support of Delphi Applications

This functionality is supported only if you are using the Classic Agent.

While there is no support for Delphi controls "out of the box", virtually all of the Delphi objects can be class mapped to standard controls.

The following code sample shows the class mapping for the Classic Agent classes:

```
[ClassMap]
DialogBox,0x50000044,0x50000044=Ignore
TBitBtn=PushButton
TButton=PushButton
TCheckBox=CheckBox
TComboBox=ComboBox
TDBCheckBox=CheckBox
TDBComboBox=ComboBox
TDBEdit=TextField
TDBListBox=ListBox
TDBLookupComboBox=ComboBox
TDBLookupListBox=ListBox
```

```
TDBMemo=TextField
TDBRadioGroup=Ignore
TEdit=TextField
TFlyingPanel=Toolbar
TGroupBox=StaticText
TGroupButton=RadioButton
TListBox=ListBox
TListView=ListView
TMaskEdit=TextField
TMemo=TextField
TPageControl=PageList
TPanel=Ignore
TRadioButton=RadioButton
TRadioGroup=Ignore
TRichEdit=TextField
TRicherEdit=TextField
TScrollBar=ScrollBar
TStatusBar=StatusBar
TTabControl=PageList
TTreeView=TreeView
TUpDown=UpDown
```

Notes

Silk Test Classic can work with Delphi objects in a variety of ways. The amount of functionality you achieve depends on how deep you want to get involved. You can even create an extension (external) for Delphi objects. Delphi supports DLL calling, and you can use DLL's created in C/C++ in your Delphi application. Class mapping will work in many instances, but not with every object.

If class mapping doesn't work, you can try any of the following workarounds:

1. Using `SendMessage` with the Clipboard.

- Delphi is built with VCL. The VCL (Visual Component Library) is similar to MFC in that all of the classes of objects that Delphi can create are in this library. Instead of C++ it is written in Object Pascal. The VCL source code is shipped with the Delphi product. In the VCL source, you can go to the definition of the object class that you want to support for and add message handlers (windows API messages) for various messages that you define.
- For example, add a message handler that says that if any object of this class receives a message called `QAP_GetValue`, get the contents of the listbox, send a message back to the process that sent the message, and send it the value. On the Silk Test Classic side of things you define a new class to support the object and add a method that sends/receives the message to the supported object.
- For example, here is sample code of a message handler on the Delphi side:

```
procedure QAP_GetValue (var Msg: TMessageRecord);
var
ValueToReturn : string;
begin
CopyToClipBoard;
Msg.Result := true;
end;
```

- Here is sample code for the `Window` class on the 4Test side:

```
winclass DelphObj : Control
LIST OF STRING GetContents ()
if (SendMessage (this.hWnd, QAP_GetListContents, NULL, NULL))
return Clipboard.GetText ()
else
RaiseError (1, "Couldn't get the contents of {this},
SendMessageEvent not processed correctly")
```

2. Using the Extension Kit, create a DLL that does the same thing as above, except passing values directly from application to application rather than relying on the clipboard. This method is preferred over the above because of speed and data type stability.
3. Use low level 4Test events relying on coordinates to create methods. Silk Test Classic low-level recording should only be used when you want to use recording rather than hand scripting.

Open Agent

To test Delphi applications with the Open Agent you could use the custom control support. For additional information, see *Custom Controls (Open Agent)*.

Stopping a Test Plan

Problem

You want to abort a test plan programmatically without using `exit`. Calling `exit` just aborts the script and continues on to the next test case.

Solution

You can call

```
[ ] @("$StopRunning") ()
```

from a test case or a recovery system function such as `ScriptExit()`, which is called for each test case in the test plan, or `TestCaseExit()`.

This call will stop everything without even invoking the recovery system. Calling it will generate the following exception message, with no call stack: `Exception -200000`

A Text Field Is Not Allowing Input

This functionality is supported only if you are using the Classic Agent.

Problem

A text field does not accept input from `TypeKeys` and `SetText` or allow a paste from the **Clipboard**.

For example, in the following script, the **Password** text field does not get any text set in it:

```
EnterNetworkPassword.SetActive ()
EnterNetworkPassword.Password.SetText ("mypassword")
EnterNetworkPassword.OK.Click ()
```

Solution

Make a DLL call to `SendMessage`, which is declared in `msw32.inc`, in the following way:

```
use "msw32.inc"
...
Clipboard.SetText ({"mypassword"})
EnterNetworkPassword.Password.DoubleClick ()
SendMessage (EnterNetworkPassword.Password.hWnd,WM_PASTE, 0,0)
```

By using the API message `WM_PASTE` in a `SendMessage` call, the text field will get populated with the text that is on the **Clipboard**.

Using a Property Instead of a Data Member

Data members are resolved (assigned values) during compilation. If the expression for the data member includes variables that will change at run-time, then you must use a property instead of that data member.

Using File Functions to Add Information to the Beginning of a File

In Silk Test Classic 5.5 SP1 or later, there is no file open mode that allows you to insert information into the beginning of a file. If you use FM_UPDATE, you can read in part of your file before writing, but any write function calls will overwrite the rest of the file.

If you are writing strings rather than structured data, you can use `ListRead()` and `ListWrite()` to insert information at the beginning or any other point of a file. Use `ListRead()` to read the contents of the file into a list, insert the new information at the head or any other point of the list, and use `ListWrite()` to write it back out.

```
[-] LIST OF STRING lsNewInfo = {...}
[ ] "*New line one*"
[ ] "*New line two*"
[ ] "*New line three*"
[ ] LIST OF STRING lsFile
[ ] INTEGER i
[ ]
[ ] ListRead (lsFile, "{GetProgramDir ()}\Sample.txt")
[-] for i = 1 to ListCount (lsNewInfo)
[ ] ListInsert (lsFile, i, lsNewInfo[i])
[ ] ListWrite (lsFile, "{GetProgramDir ()}\Sample.txt")
[ ]
```

`Sample.txt` before the write:

```
Line 1
Line 2
Line 3
Line 4
Line 5
```

`Sample.txt` after:

```
*New line one*
*New line two*
*New line three*
Line 1
Line 2
Line 3
Line 4
Line 5
```

Why Does the Str Function Not Round Correctly?

Any decimal/float number has an internal binary representation. Unfortunately, you can never be sure if a decimal value has an exact representation in its binary pendant. If an exact binary representation is not possible (mathematical constraint), the nearest value is used and this leads to the issue where it seems the `str` function is not rounding correctly. You can workaround this issue. Use the following code to see the internal representation:

```
[ ] printf("%.a20e\n", 32.495)
[ ] printf("%.a20e\n", 31.495)
```

Troubleshooting Projects

This section provides solutions to common problems that you might encounter when you are working with projects in Silk Test Classic.

Files Not Found When Opening Project

If, when opening your project, Silk Test Classic cannot find a file in the location referenced in the project file, which is a `.vtp` file, an error message displays noting the file that cannot be found.

Silk Test Classic may not be able to find files that have been moved or renamed outside of Silk Test Classic, for example in Windows Explorer, or files that are located on a shared network folder that is no longer accessible.

- If Silk Test Classic cannot find a file in your project, we suggest that you note the name of missing file, and click **OK**. Silk Test Classic will open the project and remove the file that it cannot find from the project list. You can then add the missing file to your project.
- If Silk Test Classic cannot open multiple files in your project, we suggest you click **Cancel** and determine why the files cannot be found. For example a directory might have been moved. Depending upon the problem, you can determine how to make the files accessible to the project. You may need to add the files from their new location.

Silk Test Classic Cannot Load My Project File

If Silk Test Classic cannot load your project file, the contents of your .vtp file might have changed or your .ini file might have been moved.

If you remove or incorrectly edit the `ProjectIni=` line in the `ProjectProfile` section of your `<projectname>.vtp` file, or if you have moved your `<projectname>.ini` file and the `ProjectIni=` line no longer points to the correct location of the `.ini` file, Silk Test Classic is not able to load your project.

To avoid this, make sure that the `ProjectProfile` section exists in your `.vtp` file and that the section refers to the correct name and location of your `.ini` file. Additionally, the `<projectname>.ini` file and the `<projectname>.vtp` file refer to each other, so ensure that these references are correct in both files. Perform these changes in a text editor outside of Silk Test Classic.

Example

The following code sample shows a sample `ProjectProfile` section in a `<projectname>.vtp` file:

```
[ProjectProfile]
ProjectIni=C:\Program Files\<Silk Test install directory>
\SilkTest\Projects\<projectname>.ini
```

Silk Test Classic Cannot Save Files to My Project

You cannot add or remove files from a read-only project. If you attempt to make any changes to a read-only project, a message box displays indicating that your changes will not be saved to the project.

For example, `Unable to save changes to the current project. The project file has read-only attributes.`

When you click **OK** on the error message box, Silk Test Classic adds or removes the file from the project temporarily for that session, but when you close the project, the message box displays again. When you re-open the project, you will see your files have not been added or removed.

Additionally, if you are using Microsoft Windows 7 or later, you might need to run Silk Test Classic as an administrator. To run Silk Test Classic as an administrator, right-click the Silk Test Classic icon in the **Start Menu** and click **Run as administrator**.

Silk Test Classic Does Not Run

The following table describes what you can do if Silk Test Classic does not start.

If Silk Test Classic does not run because it is looking for the following:	You can do the following:

Project files that are moved or corrupted.	Open the `SilkTestClassic.ini` file in a text editor and remove the `CurrentProject=` line from the `ProjectState` section. Silk Test Classic should then start, however your project will not open. You can examine your `<projectname>.ini` and `<projectname>.vtp` files to determine and correct the problem.
	The following code example shows the `ProjectState` section in a sample `partner.ini` file:
	`[ProjectState]`
	`CurrentProject=C:\Program Files \<SilkTest install directory>`
	`\SilkTest\Examples\ProjectName.vtp`

A `testplan.ini` file that is corrupted.	Delete or rename the corrupted `testplan.ini` file, and then restart Silk Test Classic.

My Files No Longer Display In the Recent Files List

After you open or create a project, files that you had recently opened outside of the project do no longer display in the **Recent Files** list.

Cannot Find Items In Classic 4Test

If you are working with Classic 4Test, objects display in the correct nodes on the **Global** tab, however when you double-click an object, the file opens and the cursor displays at the top of the file, instead of in the line in which the object is defined.

Editing the Project Files

You require good knowledge of your files and how the partner and `<projectname>.ini` files work before attempting to edit these files. Be cautious when editing the `<projectname>.vtp` and `<projectname>.ini` files.

To edit the `<projectname>.vtp` and `<projectname>.ini` files:

1. Update the references to the source location of your files. If the location of your `projectname.vtp` and `projectname.ini` files has changed, make sure you update that as well. Each file refers to the other.

The ProjectProfile section in the `projectname.vtp` file is required. Silk Test Classic will not be able to load your project if this section does not exist.

1. Ensure that your project is closed and that all the files referenced by the project exist.
2. Open the `<projectname>.vtp` and `<projectname>.ini` files in a text editor outside of Silk Test Classic.

Note: Do not edit the `projectname.vtp` and `projectname.ini` files in the 4Test Editor.

3. Update the references to the source location of your files.
4. The `<projectname>.vtp` and `<projectname>.ini` files refer to each other. If the relative location of these files has changed, update the location in the files.

The `ProjectProfile` section in the `<projectname>.vtp` file is required. Silk Test Classic is not able to load your project if this section does not exist.

Recognition Issues

This section provides help and troubleshooting information for recognition issues.

How Can the Application Developers Make Applications Ready for Automated Testing?

The attributes available for a specific control in the application under test (AUT) might not be sufficient to guarantee that Silk Test Classic always recognizes the control during automated testing. In such a case the application developer can add custom attributes to the control, which can then be used as locator attributes for the control. The following examples describe how an application developer can include custom attributes in different application types:

- To include custom attributes in a Web application, add them to the html tag. Type `<input type='button' bcauid='abc' value='click me' />` to add an attribute called bcauid.
- To include custom attributes in a Java SWT application, use the `org.swt.widgets.Widget.setData`(String <varname>key</varname>, Object <varname>value</varname>) method.
- To include custom attributes in a Swing application, use the `SetClientProperty`("propertyName", "propertyValue") method.

I Cannot See all Objects in my Application even after Enabling Show All Classes

This functionality is supported only if you are using the Classic Agent.

If some of the objects in your application are derived from the AWT Object class, instead of the AWT Component Class, Silk Test Classic will not be able to find them. In this situation, we recommend using the `invokeJava` method to manipulate these objects.

java.lang.UnsatisfiedLinkError

This functionality is supported only if you are using the Classic Agent.

When recording window declarations, the following error displays:

```
SilkTest Java extension loaded, running under JDK version x
java.lang.UnsatisfiedLinkError: no qapjarex in java.library.path
```

Copy the `qapjarex.dll` from the `System32` directory into the `lib\ext` directory of the JRE installed by the application.

JavaMainWin is Not Recognized

This functionality is supported only if you are using the Classic Agent.

If Silk Test Classic sees the main window of your Java application or applet as `MainWin` instead of `JavaMainWin`, or recognizes the main window, but none of its child controls, you can run a Java status utility to help you or Technical Support diagnose the problem.

One common cause is that Silk Test Classic is not properly configured to test Java.

If you are running an applet using the plug-in for JVM 1.4+, Silk Test Classic may recognize the main window as class `DialogBox`, not `JavaMainWin`, and will not see any child objects. The solution is to class-map the top-level class of the applet to `JavaMainWin`. For example:

```
microfocus.com.appclass=JavaMainWin
```

None of My Java Controls are Recognized

This functionality is supported only if you are using the Classic Agent.

If you notice that Silk Test Classic does not recognize any of your Java controls, which means that Silk Test Classic sees them all as `CustomWin` objects, make sure you have set up your test environment correctly by:

1. Ensuring that you have configured Silk Test Classic support for Java correctly.
2. Ensuring that you have enabled the Java extension correctly, based on the runtime environment your Java application invokes, as explained in the following table:

If your application invokes:	Enable Java support by:
• `java.exe` (Java Development Kit) • `jre.exe` (Java Runtime Environment, standard version that invokes a console window) • `jrew.exe` (Java Runtime Environment, version that does not invoke a console window) • `vcafe.exe` (Symantec Visual Café 2.0) • `appletviewer.exe`	Enabling the Java extension for **Java Application** in the **Extension Options** dialog box.
Any other runtime environment that uses a different executable or dll.	Adding the .exe or .dll file as an application in the **Extension Enabler** and **Extension Options** dialog box. Enabling the Java extension for this application in the **Extension Enabler** and **Extension Options** dialog box.

Only JavaMainWin is Recognized

This functionality is supported only if you are using the Classic Agent.

If during recording of a window declaration, Silk Test Classic sees the main window as a `JavaMainWin` or `JavaDialogBox`, but does not see any child objects, make sure that your classpath references the correct Silk Test Classic .jar file.

If your application sets the Java library path using the JVM launcher directive `Djava.library.path=<path>`, you must copy `qapjarex.dll` from the `System32` directory into the location pointed to by the JVM launcher directive. Silk Test Classic should then recognize child objects.

Only Applet Seen

This functionality is supported only if you are using the Classic Agent.

If only the Java applet is seen during the recording of a window declaration, and no other objects are recognized, check the following:

- The Java extension is enabled.
- Your classpath references the correct .jar file.

Silk Test Classic Does not Record Click() Actions Against Custom Controls in Java Applets

This functionality is supported only if you are using the Classic Agent.

Security restrictions may prevent Silk Test Classic from recording `Click`() actions against custom controls in Java applets. This may happen if the applet is from an untrusted source.

Verify that you have the correct security permissions set up for your AUT. The following steps show how you can verify the settings for the test applet, installed in the `<Silk Test installation directory>/javaex/jfc11` directory:

1. Locate the plug-in used to run the Applet. If you have multiple plug-ins installed, you may find the used plug-in with the java console running(use the plug-in setting Show Console). To display the system properties with java.home pointing to the directory of used plug-in, type "s" while the console is active.

   This is usually located in the `program files/java/jre_name` directory. If you have multiple plug-ins installed, you may find the used plug-in with the java console running. Use the plug-in setting **Show Console**. To display the system properties with `java.home` pointing to the directory of used plug-in, type **s** while the console is active.
2. Locate the `lib/security` directory located under the plug-in directory.
3. Open the `java.policy` file. Verify that the following fragment is in the file, or add it to the file, if it is not.

   This assumes that you have installed Silk Test Classic into the default installation directory, `C:\program files\Silk\silktest`; if you installed Silk Test Classic into a different directory, you must change the fragment accordingly.

   ```
   grant codeBase "file:C:/program files/silk/silktest/javaex/jfc11/*"
   {
     permission java.security.AllPermission;
   };
   ```

Note: The file protocol is used here because the applet is located on the host machine. If the applet was downloaded from a URL instead, then you must substitute the appropriate http://url_name instead.

Silk Test Classic Does not Recognize a Popup Dialog Box caused by an AWT Applet in a Browser

This functionality is supported only if you are using the Classic Agent.

If an AWT applet in a browser causes a popup dialog box to appear, Silk Test Classic does not see it as a `JavaDialogBox` and does not see any of the controls within the dialog box.

Click **OptionsClass Map** to map the `AppletPopup` custom class to the `JavaDialogBox` class.

Silk Test Classic is Not Recognizing Updates on Internet Explorer Page Containing JavaScript

This functionality is supported only if you are using the Classic Agent.

.

If Silk Test Classic does not recognize updates made to an Internet Explorer page containing JavaScript, Silk Test Classic does not know that the page changed because the existing objects change, but nothing gets created or destroyed.

In such a case, call `BrowserPage.FlushCache()` in between the update methods. The `FlushCache` method is useful when a JavaScript event causes an update to existing objects on the page, but does not cause any objects to be created or removed.

Java Controls are Not Recognized

This functionality is supported only if you are using the Classic Agent.

By default, objects that are usually not relevant for testing (containers and panels) are ignored. This is done to promote efficient recording. In some situations, however, user-defined objects or third-party JavaBeans might also be ignored inadvertently.

You can access these objects for testing by recording classes for ignored objects in standalone Java applications or in Java applets.

Verify Properties does not Capture Window Properties

This functionality is supported only if you are using the Classic Agent.

If **Verify Properties** does not capture window properties for a stand-alone Java application, do not position the cursor in title bar to verify properties.

In order to use **Verify Properties** against a stand-alone Java application, position your cursor at a point within the client area of the window. Do not position the cursor in the title bar because that may prevent capturing the window properties.

Tips

This section provides general troubleshooting tips.

Owner-Draw List Boxes and Combo Boxes

This functionality is supported only if you are using the Classic Agent.

An owner-draw list/combo box is a list/combo box that has the owner-draw style bit set. This is distinct from a custom object that looks like a standard list/combo box, but is not.

The following procedure describes how developers can modify an application so that Silk Test Classic can access the text of a standard list/combo box that is owner-draw and that does not have the HasStrings style bit turned on. (If the HasStrings style bit of the owner-draw list/combo box is turned on, then you do not need to make the modifications described here.) The procedure entails modifying each owner-draw list/combo box's parent window procedure so that Silk Test Classic can query the parent about what is in the list/combo box.

To turn on the HasStrings style bit of the owner-draw list/combo box:

1. Include `owndraw.h`, which is supplied with Silk Test Classic, in your source files.
2. For each owner-draw list/combo box, add a message handler for each of the messages in `owndraw.h` (`ODA_HASTEXT`, `ODA_GETITEMTEXT`, `ODA_GETITEMTEXTSIZE`). Base the message handlers on the 'switch' statement cases below. If writing in C, add code, such as that shown below, to its parent s WndProc.

```
LONG FAR PASCAL ListParentWndProc (HWND hWnd, UINT uiMsg,WPARAM wParam,
LPARAM lParam)
{
   //Use a static for the registered message number
```

```
   static UINT uiMsgGetItem Text = 0;

   LPGETITEMTEXTSTRUCT LpGetItemText;
   USHORT usItem;
   PSZ pszItemText;

   //Register the QAP_GETITEMTEXT message if it is not registered
   if (uiMsgGetItem Text == 0)
      uiMsgGetItemText = RegisterWindowMessage("QAP_GETITEMTEXT");

   switch (uiMsg)
   {
   ...
   default;
      //Process the QAP_GETITEMTEXT message
      if (uiMsg == uiMsgGetItemText)
      {
         //lParam points to a LPGETITEMTEXTSTRUCT structure
         lpGetItemText = (LPGETITEMTEXTSTRUCT) lParam;

         //Perform the requested action
         switch (lpGetItemText->Action)
         {
         case ODA_HASTEXT:
            //Tell the QAP driver if your list box contains text
            if (your list box has text)
               lpGetItemText->bSuccess = TRUE;
            else
               lpGetItemText->bSuccess = FALSE;
            break;

         case ODA_GETITEMTEXT:
            //Return the text for the requested list item
            //(lpGetItemText->itemID is the index of the item in the
            //list/combo box -- the same number passed to LB_GETITEMDATA
            (for a list box) or CB_ GETITEMDATA (for a combo box))
            usItem = UINT (lpGetItemText->itemID);
            pszItemText = <pointer to text of item[usItem]>;
            strncpy (lpGetItemText->lpstrItemText, pszItemText,
               lpGetItemText->nMaxItemText);
           lpGetItemText->lpstrItemText[lpGetItemText->nMaxItemText-1]
               = '\0';
            lpGetItemText->bSuccess = TRUE;
            break;

         case ODA_GETITEMTEXTSIZE:
            //Return the length of the requested list item
            usItem = UINT (lpGetItemText->itemID);
            lpGetItemText->nMaxItemText = <length of item[usItem]> + 1;
            lpGetItemText->bSuccess = TRUE;
            break;
         ...
         }
      }
    }
}
```

Options for Legacy Scripts

This functionality is supported only if you are using the Classic Agent.

OldStartupFunction

You can change the way a script is initialized when it is opened, by setting the `OldStartupFunction` in the `[Runtime]` section of your `partner.ini` file. By setting the option to FALSE, you can have variables, windows, and other structures initialized in the order in which they are declared. When `OldStartupFunction` is set to TRUE, the default, windows are initialized before global variables. Therefore, you cannot initialize a window member from a global variable, regardless of how they were declared, but you can initialize a global variable from a window member.

AutoGuiTarget and WinvarInitCorrectly

If you have legacy scripts that run in distributed environments or on multiple platforms and that do not compile in this release, consider the two runtime options **AutoGuiTarget** and **WinvarInitCorrectly**. They deal with problems caused by the interaction of distributed testing and GUI-specific variable initialization. Set these options in the `[Runtime]` section of your `partner.ini` file.

You may find it useful to read about the GUI Targets field that appears on the **Runtime Options** dialog.

The following code produces a runtime error, because it is not possible to initialize the GUI-specific window member `abc.ghi` so that its value is correct for both the Windows 32 and Windows 9.x platforms:

Example 1

```
window abc
msw32STRING def = "1"
mws9x STRING def = "2"
msw32, msw9x STRING ghi = def
```

To avert this problem, the runtime option **AutoGuiTarget** was added. If set to TRUE, the default value, and when networking is not enabled, the GUI target is automatically set to the platform the test is running on. If it is FALSE or if the network is enabled, and if the GUI target includes both msw32 and msw9x, the code will continue to produce an error because no assumptions can be made about the GUI target.

If you are running in a distributed environment, where you might simultaneously connect to msw32 and msw9x, you must modify your code so that the variable (in the above example, `abc.ghi`) is defined for each individual platform.

In some cases, however, setting **AutoGuiTarget** to TRUE causes a problem. Consider the following code, which produces a compile-time error on Windows, because the GUI-specific window member `abc.TF` is defined only for msw32.

Example 2

```
window abc
msw32 TextField TF

main ()
BOOLEAN bmsw32 = (GetGuiType () == msw32) ? TRUE : FALSE

if (bmsw32)
abc.TF.SetText ("some value")
```

When the GUI target excludes msw32, a compile-time error will result, because the variable `abc.TF` is defined only for msw32. When the GUI target includes msw32, the reference to `abc.TF` is never executed, and so a runtime error does not occur. However, when **AutoGuiTarget** is set to TRUE, the GUI target excludes msw32, and so the code will not compile. For this reason, **WinvarInitCorrectly** was added.

When **WinvarInitCorrectly** is set to TRUE, the default, window variables are initialized to allow correct operation in a distributed environment. When **WinvarInitCorrectly** is FALSE, window variables are initialized as if the only GUI that will ever be used is the one connected to when your script starts running. Any subsequent connections to other GUIs may see window variables that are initialized incorrectly or not at all.

Summary

Here are some guidelines regarding legacy scripts:

- We recommend that you rewrite scripts running on multiple platforms so that window variables can resolve correctly. In this case, the default settings of both options are fine.
- Scripts similar to Example 2 will work well in a distributed environment. However, if the network is disabled, you will need to set **AutoGuiTarget** to FALSE.

In a nondistributed environment set **AutoGuiTarget** to TRUE in order to run scripts resembling Example 1. However, if you have legacy scripts with code similar to both examples, set both options to FALSE, thereby restoring initialization behavior supported in earlier releases.

Declaring an Object for which the Class can Vary

This functionality is supported only if you are using the Classic Agent.

This topic describes how you can declare an object for which the class can vary. For example, you may have text on an HTML page that is a link under certain conditions.

If the class of an object varies, then you need to choose a class for the window identifier, and include both classes in the tag. The class of the window identifier for the object determines which methods and properties you can call for the object. Therefore you should choose the class that includes the wider set of methods and properties. However, keep in mind that when the object has the other class, you should only call those methods and properties that apply to that other class.

The tag of the object should be a multitag with a tag segment for each class. The class tag should be included in square brackets in each segment. You can either use the 'multitag' statement, or you can use the 'tag' statement with pipes (|) between segments.

Example

Assume that you have text on an HTML page that is a link only under certain conditions. The caption of the text is `Inactive Text`, but when it becomes a link, the caption changes to `Active Text`.

Choose `HtmlLink` as the class of the window identifier because `HtmlLink` includes all of the `HtmlText` methods, and also includes additional methods such as `GetLocation`(). Of course, `GetLocation`() will not return a meaningful value if you call it when the object is really just `HtmlText`.

The declaration for the object would be:

```
HtmlLink TextOrLink
// recorded as 'InactiveText', since it was recorded when it
// was HtmlText
multitag "[HtmlText]InactiveText"
  "[HtmlLink]ActiveText"
```

or

```
HtmlLink TextOrLink
// recorded as 'InactiveText', since it was recorded
// when it was HtmlText
tag "[HtmlText]InactiveText|[HtmlLink]ActiveText"
```

Drag and Drop Operations

This functionality is supported only if you are using the Classic Agent.

Silk Test Classic supports drag-and-drop operations on Windows. Drag-and-drop operations have three distinct parts:

- Selecting an item by pressing a mouse button.
- Moving, or dragging, the item.
- Releasing the mouse button, thereby dropping the item at a target location.

The target location can be a logical location, that is, an identifiable object in a listview, treeview, or list box, or it can be a physical location specified by x, y coordinates in a window.

Five methods support drag-and-drop operations:

- `BeginDrag`
- `BeginDragAt`
- `EndDrag`
- `EndDragAt`
- `DragMouse`

`BeginDragAt` and `EndDragAt` are general methods that work for any window. They move an item to a physical target location and operate between windows of the same application or a different application. We recommend that you use care when recording drag-and-drop operations. Do the testcase setup carefully, and while recording, avoid extraneous movements.

`EndDrag` and `BeginDrag` apply only to list box, listview, and treeview controls. They move an item to a logical target location and operate between windows of the same application or a different application.

`DragMouse` combines the functionality of the begin and end drag methods. However, `DragMouse` operates only within a single window.

Example Test Cases for the Find Dialog Box

If you want to test the **Find** dialog box, each test case would need to perform the following tasks:

1. Open a new document file.
2. Type text into the document.
3. Position the insertion point at the top of the file.
4. Select **Find** from the **Search** menu.
5. Select the forward (down) direction for the search.
6. Make the search case sensitive.

Non-Data-Driven Test Case for the Classic Agent

```
testcase FindTest ()
TextEditor.File.New.Pick ()
DocumentWindow.Document.TypeKeys ("Test Case<HOME>")
TextEditor.Search.Find.Pick ()
Find.FindWhat.SetText ("Case")
Find.CaseSensitive.Check ()
Find.Direction.Select ("Down")
Find.FindNext.Click ()
Find.Cancel.Click ()
DocumentWindow.Document.VerifySelText (<text>)
Case
TextEditor.File.Close.Pick ()
MessageBox.No.Click ()
```

The major disadvantage of this kind of test case is that it tests only one out of the many possible sets of input data to the **Find** dialog box. To adequately test the **Find** dialog box, you must record or hand-write a separate test case for each possible combination of input data that needs to be tested. In even a small application, this creates a huge number of test cases, each of which must be maintained as the application changes.

Non-Data-Driven Test Case for the Open Agent

```
testcase Find ()
  recording
    UntitledNotepad.SetActive()
    UntitledNotepad.New.Pick()
    UntitledNotepad.TextField.TypeKeys("Test Case
<LessThan>Home")
    UntitledNotepad.TextField.PressKeys("<Left Shift>")
    UntitledNotepad.TextField.TypeKeys("<GreaterThan>")
    UntitledNotepad.Find.Pick()
    UntitledNotepad.FindDialog.FindWhat.SetText("Case")
    UntitledNotepad.FindDialog.Down.Select("Down")
    Tmp_findNotepad.Find.MatchCase.Check()
    UntitledNotepad.FindDialog.FindNext.Click()
    Tmp_findNotepad.Find.Cancel.Click()
    Tmp_findNotepad.Find.Close()
```

Declaring an Object for which the Class can Vary

This functionality is supported only if you are using the Classic Agent.

This topic describes how you can declare an object for which the class can vary. For example, you may have text on an HTML page that is a link under certain conditions.

If the class of an object varies, then you need to choose a class for the window identifier, and include both classes in the tag. The class of the window identifier for the object determines which methods and properties you can call for the object. Therefore you should choose the class that includes the wider set of methods and properties. However, keep in mind that when the object has the other class, you should only call those methods and properties that apply to that other class.

The tag of the object should be a multitag with a tag segment for each class. The class tag should be included in square brackets in each segment. You can either use the 'multitag' statement, or you can use the 'tag' statement with pipes (|) between segments.

Example

Assume that you have text on an HTML page that is a link only under certain conditions. The caption of the text is `Inactive Text`, but when it becomes a link, the caption changes to `Active Text`.

Choose `HtmlLink` as the class of the window identifier because `HtmlLink` includes all of the `HtmlText` methods, and also includes additional methods such as `GetLocation`(). Of course, `GetLocation`() will not return a meaningful value if you call it when the object is really just `HtmlText`.

The declaration for the object would be:

```
HtmlLink TextOrLink
// recorded as 'InactiveText', since it was recorded when it
// was HtmlText
multitag "[HtmlText]InactiveText"
  "[HtmlLink]ActiveText"
```

or

```
HtmlLink TextOrLink
// recorded as 'InactiveText', since it was recorded
// when it was HtmlText
tag "[HtmlText]InactiveText|[HtmlLink]ActiveText"
```

When to use the Bitmap Tool

You might want to use the **Bitmap Tool** in these situations:

- To compare a baseline bitmap against a bitmap generated during testing.
- To compare two bitmaps from a failed test.

For example, suppose during your first round of testing you create a bitmap using one of Silk Test Classic's built-in bitmap functions, `CaptureBItmap`. Assume that a second round of testing generates another bitmap, which your test script compares to the first. If the testcase fails, Silk Test Classic raises an exception but cannot specifically identify the ways in which the two images differ. At this point, you can open the **Bitmap Tool** from the results file to inspect both bitmaps.

Troubleshooting Web Applications

The test of your browser application may have failed for one of the reasons described in this section. If the suggested solutions do not address the problem you are having, you can enable your extension manually.

Why Is My Web Application Not Ready To Test?

This functionality is supported only if you are using the Classic Agent.

If your Web application is not ready to test, enable the extension for the Web application and restart the Web application.

1. On the **Basic Workflow** bar, click **Enable Extensions**. The **Enable Extensions** dialog box opens.
2. On the **Enable Extensions** dialog box, select the Web application for which you want to enable extensions.
3. Click **OK**. The **Enable Extensions** dialog box closes.
4. Close and restart the Web application.
5. When the application has finished loading, click `Test`.

What Can I Do If the Page I Have Selected Is Empty?

If the page you are testing is empty or does not contain any HTML elements, you might receive a `Could not recognize any HTML classes in your browser application` message. Your configuration might be correct, however, the automated configuration test does not support testing of blank pages or pages that do not contain HTML elements. You can manually verify that your extensions are set properly, open your application, and then record window declarations. If you can record against HTML classes, the extension is configured correctly and you are ready to set up the recovery system using the **Basic Workflow** bar.

Why Do I Get an Error Message When I Set the Accessibility Extension?

If you are using Internet Explorer to test a Web application and you have set the Accessibility extension, you might get an error message when the start page of the browser is "about:blank". To avoid getting the error message, set the start page of the browser to a different page.

HtmlPopupList Causes the Browser to Crash when Using IE DOM Extension

This functionality is supported only if you are using the Classic Agent.

Problem

Recording or playing back selections against an **HtmlPopupList** causes the browser to crash when using the IE DOM extension.

The problem occurs on browser pages that contain JavaScript. It seems to occur more quickly when recording than when playing back.

Solution

Check the **UseDocumentEvents** option in `extend\domex.ini` and make sure it is set to `FALSE`.

This setting lets you specify whether Silk Test Classic captures OnChange events for an **HtmlPopupList** in order to determine which item was selected. The default is `FALSE`.

Silk Test Classic Does Not Recognize Links

This functionality is supported only if you are using the Classic Agent.

Problem

Silk Test Classic is not seeing your links as `HtmlLink` objects.

Cause

You have configured your browser not to underline links. To recognize a link, Silk Test Classic requires the link to be underlined.

Solution

Reconfigure your browser to display links underlined.

Mouse Coordinate (x, y) is Off the Screen

This functionality is supported only if you are using the Classic Agent.

Problem

While running the DOM extension, you may receive an error message during playback saying `Mouse Coordinate (x,y) is off the screen`.

Resolution

Clear the following options on the **Verification** tab of the **Options** > **Agent Options** window:

- Verify that windows are exposed
- Verify that coordinates passed to a method are inside the window

This error message may also indicate that the DOM extension was unable to scroll an object into view, for example as part of a `MoveMouse()` call. If that is the case, then set the DOM extension option `UseScrollIntoView` to `TRUE`.

Recording a Declaration for a Browser Page Containing Many Child Objects

This functionality is supported only if you are using the Classic Agent.

When trying to record a declaration for a browser page containing many child objects, CPU usage stays at 100% and Silk Test Classic seems to lock up.

If you add the `Partner.ini` setting:

```
[Runtime]
AgentTimeout=<value in milliseconds>
```

and set it very large, for example `600000` (10 minutes), then eventually Silk Test Classic will return a declaration. The default is `240000` milliseconds (240 seconds or 4 min).

The setting must be put into `Partner.ini`, not into an option set.

For large pages, recording will be very slow, and the CPU will max out while the browser extension initially gathers the page information, and again when it transfers the information to the Agent. If you refrain from moving the mouse around and wait long enough, though, the system will eventually free up.

Recording VerifyProperties() Detects BrowserPage Properties and Children

This functionality is supported only if you are using the Classic Agent.

Problem

When you record `VerifyProperties`() against the `BrowserChild`, the recorder misses user-defined properties, and records dynamically instantiated window identifiers of child windows instead of recording the declared window identifiers.

Note: The Record Actions status line detects the window identifier of the `BrowserChild`. The problem only affects the body of the `VerifyProperties`() method that is recorded.

Solution

If you plan to record `VerifyProperties`() against a declared `BrowserChild`, then you must include the window declaration for the `BrowserChild` in a frame (.inc) file that is referenced in the **Use Files** field of the **Runtime Options** dialog box, before `extend\explorer.inc`.

Silk Test Classic Cannot See Any Children in My Browser Page

This functionality is supported only if you are using the Classic Agent.

Problem

Silk Test Classic does not recognize any children in your Web application running on Internet Explorer 6.

Solution

If you are using Internet Explorer 6 and only see a `BrowserChild` with no child objects within it, enable the third-party browser extensions:

1. In Internet Explorer, click **Tools** > **Internet Options**.
2. On the **Internet Options** dialog box, click the **Advanced** tab and scroll to the **Browsing** section.
3. Check **Enable third-party browser extensions**.
4. Restart your computer.

This Internet Explorer option is disabled by default in Internet Explorer 6.0.3, which ships with Microsoft Windows Server 2003. If you are using Internet Explorer 6.0.3, you are likely to run into this problem.

Silk Test Classic Cannot Verify Browser Extension Settings

This functionality is supported only if you are using the Classic Agent.

Problem

Silk Test Classic is not able to verify extension settings for a Web application running in Internet Explorer.

Cause

Your browser's third-party extensions are not enabled.

Solution

1. In Internet Explorer, click **Tools** > **Internet Options**.
2. On the **Internet Options** dialog box, click the **Advanced** tab and scroll to the **Browsing** section.
3. Check **Enable third-party browser extensions**.
4. Restart your computer.

Silk Test Classic Cannot Find the Web Page of the Application

This functionality is supported only if you are using the Classic Agent.

Problem

A test you have successfully run on one browser fails on a different browser because Silk Test Classic cannot find the Web page of the application on the second browser.

Cause

Web browsers truncate Web page titles if they are too long. Each browser truncates long titles in a different way. Silk Test Classic derives the tag for a page from the page's title. When you replay the test, the browser which was used to record the page declaration will recognize the tag for the page title. Other browsers might not.

The following example shows how Mozilla Firefox and Internet Explorer truncate the title of the same page.

For the Web page entitled `"Heretoday Com Information`:

Internet Explorer	`tag "http:??www.heretoday.com? . . . ?information.htm`
Mozilla Firefox	`tag "http:??www.heretoday . . . valuable?information.htm`

Solution

Use a wildcard character (`*`) to abbreviate the tag unambiguously. For example:

```
tag "*information.htm"
```

Silk Test Classic Cannot Recognize Web Objects

This functionality is supported only if you are using the Classic Agent.

Problem

When you are recording or playing back tests, Silk Test Classic does not see the objects in your Web application. Instead, it sees custom windows.

Possible Causes and Solutions

Cause	Solution
The browser extension is not enabled.	Enable the browser extension.
The browser extension is enabled, but the default browser is not correct. For example, you might have extensions enabled for Internet Explorer and Mozilla Firefox, but the default browser is Internet Explorer and you are testing Mozilla Firefox.	Change the browser type. You can change it in the **Runtime Options** dialog box or you can specify it in a script using the `SetBrowserType` function.
You are using an older version of the browser.	Upgrade your browser. For information about new features, supported platforms and versions, known issues, and work-arounds, refer to the *Release Notes*.

Silk Test Classic Recognizes Static HTML Text But Does Not Recognize Text

This functionality is supported only if you are using the Classic Agent.

Problem

While using Internet Explorer, Silk Test Classic recognizes static HTML text and tables on a page, but does not recognize text such as HTML fields or HTML buttons.

Cause

You might have nested `<form>` tags improperly, like the following sample shows:

```
<table>
...
<form>
</table>
<table>
...
</table>
</form>
```

In the example above, Silk Test Classic does not recognize the second table.

Solution

There are two options:

Check that your HTML tags are properly nested	`<table>`
	`...`
	`</table>`

```
<form>
<table>
...
</table>
</form>
```

or

```
<form>
<table>
...
</table>
<table>
</table>
</form>
```

Set the table recognition value to 0	This means that Silk Test Classic ignores the table and cell, but it will recognize the other input elements.

A Test Frame Which Contains HTML Frame Declarations Does Not Compile

This functionality is supported only if you are using the Classic Agent.

Problem

When you try to compile a test frame that contains several HTML frame declarations, you get one or more `Window is already defined` errors.

Cause

The parent window of your HTML frames is declared more than once.

Solution

For information about recording HTML frame declarations, see *Streamlining HTML Frame Declarations*.

Web Property Sets Are Not Displayed During Verification

This functionality is supported only if you are using the Classic Agent.

Problem

The **Verify Window** dialog box is not displaying the property sets for the browser extensions. The property sets are the following:

- `Color`
- `Font`
- `Values`
- `Location`

Possible Causes and Solutions

No browser extension is enabled.	Make sure that at least one browser extension is enabled.

Enhanced support for Visual Basic is enabled.	Disable Visual Basic by un-checking the **ActiveX** check box for the Visual Basic application in the **Extension Enabler** dialog box and the **Extensions** dialog box.

Why Does the Recorder Generate so Many MoveMouse() Calls?

This functionality is supported only if you are using the Classic Agent.

If you are recording against a Web page that contains DHTML popup menus or other elements with JavaScript mouse movement event handlers, such as `onmouseover`, many `MoveMouse()` calls are generated. The `MoveMouse()` calls are recorded in order to improve reliability of playback against DHTML popup menus by ensuring that Silk Test Classic exposes the menus as it navigates through them. This behavior was added in Silk Test 7.1 as a replacement for the previous requirement of holding down the **Shift** key while recording against DHTML popup menus, since that requirement would not have been easily known to users. There is no straightforward way for Silk Test Classic to know whether or not a mouse movement event handler belongs to an element, such as a popup menu, that requires exposure. Therefore a `MoveMouse()` is recorded for any such element to ensure that the event handlers are triggered during playback.

There is no setting that directs the Recorder to exclude the `MoveMouse()` calls. The extra calls should not cause problems or slow playback, but you can manually delete the calls from the scripts, as long as the target elements do not require exposure.

Using the Runtime Version of Silk Test Classic

The Silk Test Classic Runtime (Runtime) provides a subset of the functionality of Silk Test Classic. Specifically, it allows you to perform all of the tasks associated with executing tests and analyzing results. You are prohibited from editing existing automation or creating new automation. The Runtime is intended to run previously compiled files. If you update a shared file while the Runtime is open, you must close the Runtime and reopen it in order to use the updated file.

Silk Test Classic Runtime is an installation option. For additional information, refer to the *Silk Test Installation Guide*.

The *Silk Test Classic Runtime Help* includes the topics that are available from the full version of Silk Test Classic, and additional product-specific information.

Installing the Runtime Version

Silk Test Classic Runtime is an installation option. For additional information, refer to the *Silk Test Installation Guide*.

We strongly recommend that you do not install Silk Test Classic Runtime on the same machine as Silk Test Classic. Silk Test Classic runtime shares files with this product and will overwrite any other installation you already have on your machine.

Note: Silk Test Classic Runtime is sold and licensed separately from standard Silk Test Classic.

Starting the Runtime Version

You can start Silk Test Classic Runtime from the following locations:

- The command-line prompt in a DOS window. Enter `runtime.exe`. The same syntax applies as with starting Silk Test Classic from the command line.
- The Silk Test Classic GUI. You must have selected the Silk Test Classic Runtime option during installation.

When you start the Runtime, it displays minimized as an icon only; click the icon to maximize the window.

Comparing Silk Test Classic and Silk Test Classic Runtime Menus and Commands

The table below lists the menus and commands that are available for each agent in Silk Test Classic and those that are available in Silk Test Classic Runtime:

Menu Name	Command	Available in Silk Test Classic	Available in Silk Test Classic Runtime
Breakpoint	**Toggle**	Classic Agent Open Agent	No

Menu Name	Command	Available in Silk Test Classic	Available in Silk Test Classic Runtime
	Add	Classic Agent Open Agent	No
	Delete	Classic Agent Open Agent	No
	Delete All	Classic Agent Open Agent	No
Debug	**Abort**	Classic Agent Open Agent	No
	Exit	Classic Agent Open Agent	No
	Finish Function	Classic Agent Open Agent	No
	Reset	Classic Agent Open Agent	No
	Run and Debug/Continue	Classic Agent Open Agent	No
	Run to Cursor	Classic Agent Open Agent	No
	Step Into	Classic Agent Open Agent	No
	Step Over	Classic Agent Open Agent	No
Edit	**Undo**	Classic Agent Open Agent	No
	Redo	Classic Agent Open Agent	No
	Cut	Classic Agent Open Agent	No
	Copy	Classic Agent Open Agent	Yes
	Select All	Classic Agent Open Agent	Yes
	Paste	Classic Agent	No

Menu Name	Command	Available in Silk Test Classic	Available in Silk Test Classic Runtime
		Open Agent	
	Delete	Classic Agent Open Agent	No
	Find	Classic Agent Open Agent	Yes
	Find Next	Classic Agent Open Agent	Yes
	Replace	Classic Agent Open Agent	No
	Go to Line	Classic Agent Open Agent	Yes
	Go to Definition	Classic Agent Open Agent	Yes
	Find Error	Classic Agent Open Agent	Yes
	Data Driven	Classic Agent Open Agent	No
	Visual 4Test	Classic Agent Open Agent	Yes
File	**New**	Classic Agent Open Agent	No
	Open	Classic Agent Open Agent	Yes
	Close	Classic Agent Open Agent	Yes
	Save	Classic Agent Open Agent	No
	Save Object File	Classic Agent Open Agent	No
	Save As	Classic Agent Open Agent	No
	Save All	Classic Agent Open Agent	No

Menu Name	Command	Available in Silk Test Classic	Available in Silk Test Classic Runtime
	New Project	Classic Agent Open Agent	No
	Open Project	Classic Agent Open Agent	Yes
	Close Project	Classic Agent Open Agent	Yes
	Export Project	Classic Agent Open Agent	No
	Email Project	Classic Agent Open Agent	No
	Run	Classic Agent Open Agent	Yes
	Debug	Classic Agent Open Agent	No
	Check out	Classic Agent Open Agent	No
	Check in	Classic Agent Open Agent	No
	Print	Classic Agent Open Agent	Yes
	Printer Setup	Classic Agent Open Agent	Yes
	Recent Files and Recent Projects	Classic Agent Open Agent	Yes
	Exit	Classic Agent Open Agent	Yes
Help	**Help Topics**	Classic Agent Open Agent	Yes
	Library Browser	Classic Agent Open Agent	Yes
	Tutorials	Classic Agent Open Agent	Yes
	About Silk Test Classic	Classic Agent	Yes

Menu Name	Command	Available in Silk Test Classic	Available in Silk Test Classic Runtime
		Open Agent	
Include	**Open**	Classic Agent Open Agent	Yes
	Open All	Classic Agent Open Agent	Yes
	Close	Classic Agent Open Agent	Yes
	Close All	Classic Agent Open Agent	Yes
	Save	Classic Agent Open Agent	No
	Acquire Lock	Classic Agent Open Agent	No
	Release Lock	Classic Agent Open Agent	No
Options	**General**	Classic Agent Open Agent	Yes
	Editor Font	Classic Agent Open Agent	Yes
	Editor Colors	Classic Agent Open Agent	Yes
	Runtime	Classic Agent Open Agent	Yes
	Agent	Classic Agent Open Agent	Yes
	Extensions	Classic Agent	Yes
	Application Configurations	Open Agent	Yes
	Recorder	Classic Agent Open Agent	No
	Class Map	Classic Agent	Yes
	Class Attributes	Classic Agent	Yes
	Property Sets	Classic Agent Open Agent	Yes

Menu Name	Command	Available in Silk Test Classic	Available in Silk Test Classic Runtime
	TrueLog	Classic Agent Open Agent	Yes
	Silk Central URLs	Classic Agent Open Agent	Yes
	Open Options Set	Classic Agent Open Agent	Yes
	Save New Options Set	Classic Agent Open Agent	No
	Close Options Set	Classic Agent Open Agent	Yes
	Recent Options Sets	Classic Agent Open Agent	Yes
Outline	**Move Left**	Classic Agent Open Agent	No
	Move Right	Classic Agent Open Agent	No
	Transpose Up	Classic Agent Open Agent	No
	Transpose Down	Classic Agent Open Agent	No
	Expand	Classic Agent Open Agent	Yes
	Expand All	Classic Agent Open Agent	Yes
	Collapse	Classic Agent Open Agent	Yes
	Collapse All	Classic Agent Open Agent	Yes
	Comment	Classic Agent Open Agent	No
	Uncomment	Classic Agent Open Agent	No
Project	**View Explorer**	Classic Agent	Yes

Menu Name	Command	Available in Silk Test Classic	Available in Silk Test Classic Runtime
		Open Agent	
	Align	Classic Agent Open Agent	Yes
	Project Description	Classic Agent Open Agent	No
	Add File	Classic Agent Open Agent	No
	Remove File	Classic Agent Open Agent	No
Record	**Window Declarations**	Classic Agent	No
	Application State	Classic Agent Open Agent	No
	Testcase	Classic Agent Open Agent	No
	Method	Classic Agent Open Agent	No
	Actions	Classic Agent	No
	Class	Classic Agent	No
	Window Identifiers	Classic Agent	No
	Window Locations	Classic Agent Open Agent	No
	Defined Window	Classic Agent	No
	Window Tags	Classic Agent	No
Results	**Select**	Classic Agent Open Agent	Yes
	Merge	Classic Agent Open Agent	Yes
	Delete	Classic Agent Open Agent	Yes
	Extract	Classic Agent Open Agent	Yes
	Export	Classic Agent	Yes

Menu Name	Command	Available in Silk Test Classic	Available in Silk Test Classic Runtime
		Open Agent	
	Launch TrueLog Explorer	Classic Agent Open Agent	Yes
	Send to Issue Manager	Classic Agent Open Agent	Yes
	Convert to Plan	Classic Agent Open Agent	No
	Compact	Classic Agent Open Agent	Yes
	Show Summary	Classic Agent Open Agent	Yes
	Hide Summary	Classic Agent Open Agent	Yes
	View Options	Classic Agent Open Agent	Yes
	Go to Source	Classic Agent Open Agent	Yes
	View Differences	Classic Agent Open Agent	Yes
	Update Expected Value	Classic Agent Open Agent	No
	Pass/Fail Report	Classic Agent Open Agent	Yes
	Mark Failures in Plan	Classic Agent Open Agent	Yes
	Compare Two Results	Classic Agent Open Agent	Yes
	Next Result Difference	Classic Agent Open Agent	Yes
	Next Error Difference	Classic Agent Open Agent	Yes
Run	**Compile**	Classic Agent Open Agent	No

Menu Name	Command	Available in Silk Test Classic	Available in Silk Test Classic Runtime
	Compile all	Classic Agent Open Agent	No
	Run	Classic Agent Open Agent	Yes
	Debug	Classic Agent Open Agent	No
	Application State	Classic Agent Open Agent	Yes
	Testcase	Classic Agent Open Agent	Yes
	Show Status	Classic Agent Open Agent	Yes
	Abort	Classic Agent Open Agent	Yes
Testplan	**Go to Script**	Classic Agent Open Agent	Yes
	Detail	Classic Agent Open Agent	No
	Insert Template	Classic Agent Open Agent	No
	Completion Report	Classic Agent Open Agent	Yes
	Mark	Classic Agent Open Agent	Yes
	Mark All	Classic Agent Open Agent	Yes
	Unmark	Classic Agent Open Agent	Yes
	Unmark All	Classic Agent Open Agent	Yes
	Mark by Query	Classic Agent Open Agent	Yes
	Mark by Named Query	Classic Agent	Yes

Menu Name	Command	Available in Silk Test Classic	Available in Silk Test Classic Runtime
		Open Agent	
	Find Next Mark	Classic Agent Open Agent	Yes
	Define Attributes	Classic Agent Open Agent	Yes
	Run Manual Tests	Classic Agent Open Agent	No
	Upload to Silk Central	Classic Agent Open Agent	No
Tools	**Start Silk Performer**	Classic Agent Open Agent	No
	Connect to Default Agent	Classic Agent Open Agent	Yes
	Data Drive Testcase	Classic Agent	No
	Enable Extensions	Classic Agent	No
	Open Silk Central	Classic Agent Open Agent	Yes
	Open Issue Manager	Classic Agent Open Agent	Yes
View/Transcript	**Expression**	Classic Agent Open Agent	No
	Global Variables	Classic Agent Open Agent	No
	Local Variables	Classic Agent Open Agent	No
	Expand Data	Classic Agent Open Agent	No
	Collapse Data	Classic Agent Open Agent	No
	Module	Classic Agent Open Agent	No
	Breakpoints	Classic Agent Open Agent	No

Menu Name	Command	Available in Silk Test Classic	Available in Silk Test Classic Runtime
	Call Stack	Classic Agent Open Agent	No
	Transcript	Classic Agent Open Agent	No
Window	**Tile Vertically**	Classic Agent Open Agent	Yes
	Tile Horizontally	Classic Agent Open Agent	Yes
	Cascade	Classic Agent Open Agent	Yes
	Arrange Icons	Classic Agent Open Agent	Yes
	Close All	Classic Agent Open Agent	Yes
	Next	Classic Agent Open Agent	Yes
	Previous	Classic Agent Open Agent	Yes
	# filename-filepath	Classic Agent Open Agent	Yes
Workflows	**Basic**	Classic Agent Open Agent	No
	Data Driven	Classic Agent Open Agent	No

Glossary

This section provides an alphabetical list of terms that are related to Silk Test Classic and their descriptions.

4Test Classes

Classes are the core of object-oriented languages such as Visual Basic or 4Test. Each GUI object is an instance of a class of objects. The class defines the actions, or methods, that can be performed on all objects of a given type. For example, in 4Test the `PushButton` class defines the methods that can be performed on all pushbuttons in your application. The methods defined for pushbuttons work only on pushbuttons, not on radio lists.

The class also defines the data, or properties, of an object. In 4Test and Visual Basic, you can set or retrieve the value of a property directly using the dot operator and a syntax similar to standard Visual Basic.

4Test-Compatible Information or Methods

Information or methods that can be passed by value in 4Test prototypes.

Abstract Windowing Toolkit

The Abstract Windowing Toolkit (AWT) is a library of Java GUI object classes that is included with the Java Development Kit from Sun Microsystems. The AWT handles common interface elements for windowing environments including Windows.

The AWT contains the following set of GUI components:

- Button
- CheckBox
- CheckBox Group (RadioList)
- Choice (PopupList)
- Label (StaticText)
- List (ListBox)
- Scroll Bar
- Text Component (TextField)
- Menu

accented character

A character that has a diacritic attached to it.

agent

The agent is the software process that translates the commands in your scripts into GUI-specific commands. It is the agent that actually drives and monitors the application you are testing.

applet

A Java program designed to run inside a Java-compatible Web browser, such as Netscape Navigator.

application state

The state you expect your application to be in at the beginning of a test case. This is in addition to the conditions required for the base state.

attributes

In the test plan editor, attributes are site-specific characteristics that you can define for your test plan and assign to test descriptions and group descriptions. Each attribute has a set of values. For example, you define the Developer attribute and assign it the values of `Kate`, `Ned`, `Paul`, and `Susan`, the names of the QA engineers in your department.

Attributes are useful for grouping tests, in that you can run or report on parts of the test plan that have a given attribute value. For example, all tests that were developed by `Bob` can be executed as a group.

In Silk Test Classic, an attribute is a characteristic of an application that you verify in a test case. Attributes are used in the **Verify Window** dialog box, which is available only for projects or scripts that use the Classic Agent.

Band (.NET)

Each level in the grid hierarchy has one band object created to represent it.

base state

The known, stable state you expect the application to be in at the start of each test case.

bidirectional text

A mixture of characters that are read from left to right and characters that are read from right to left. Most Arabic and Hebrew characters, for example, are read from right to left, but numbers and quoted western terms within Arabic or Hebrew text are read from left to right.

Bytecode

The form of Java code that the Java Virtual Machine reads. Other compiled languages use compilers to translate their code into native code, also called machine code, that runs on a particular operating system. By contrast, Java compilers translate Java programs into bytecode, an intermediate form of code that is slower than compiled code, but that can theoretically run on any hardware equipped with a Java Virtual Machine.

call stack

A call stack is a listing of all the function calls that have been called to reach the current function in the script you are debugging.

In debugging mode, a list of functions and test cases which were executing at the time at which an error occurred in a script. The functions and test cases are listed in reverse order, from the last one executed back to the first.

child object

Subordinate object in the GUI hierarchy. A child object is either logically associated with, or physically contained by, its parent object. For example, the File menu, as well as all other menus, are physically contained by the main window.

class

GUI object type. The class determines which methods can operate on an object. Each object in the application is an instance of a GUI class.

class library

A collection of related classes that solve specific programming problems. The Java Abstract Windowing Toolkit (AWT) and Java Foundation Class (JFC) are examples of Java class libraries.

class mapping

Association of nonstandard custom objects with standard objects understood by Silk Test Classic.

Classic 4Test

Classic 4Test is one of the two outline editors you can use with Silk Test Classic. Classic 4Test is similar to C and does not contain colors. Visual 4Test, enabled by default, is similar to Visual C++ and contains colors.

To switch between editor modes, click **Edit** > **Visual 4Test** to check or uncheck the check mark. You can also specify your editor mode on the **General Options** dialog box.

client area

The internal area of a window not including scroll bars, title bar, or borders.

custom object

Nonstandard object that Silk Test Classic does not know how to interact with.

data-driven test case

A special kind of test case that receives many combinations of data from 4Test functions/test plan.

data member

Variable defined within a class or window declaration. The value of a data member can be an expression, but it is important to keep in mind that data members are resolved (assigned values) during compilation. If the expression for the data member includes variables that will change at run-time, then you must use a property instead of that data member.

declarations

See *Window Declarations*.

DefaultBaseState

Built-in application state function that returns your application to its base state. By default, the built-in `DefaultBaseState` ensures that the application is running and is not minimized, the main window of the application is open, and all other windows, for example dialog boxes and message boxes, are closed.

diacritic

1. Any mark placed over, under, or through a Latin-based character, usually to indicate a change in phonetic value from the unmarked state.
2. A character that is attached to or overlays a preceding base character.

Most diacritics are non-spacing characters that don't increase the width of the base character.

Difference Viewer

Dual-paned display-only window that lists every expected value in a test case and its corresponding actual value. Highlights all occurrences where expected and actual values differ. You display the **Difference Viewer** by selecting the box icon in the results file.

double-byte character set (DBCS)

A double-byte character set, which is a specific type of multibyte character set, includes some characters that consist of 1 byte and some characters that consist of 2 bytes.

dynamic instantiation

This special syntax is called a dynamic instantiation and is composed of the class and tag or locator of the object. For example, if there is not a declaration for the **Find** dialog box of the Text Editor application, the syntax required to identify the object looks like the following:

- Classic Agent:
  ```
  MainWin("Text Editor|$D:\PROGRAM FILES
     \<SilkTest install directory>\SILKTEST\TEXTEDIT.EXE").DialogBox("Find")
  ```
- Open Agent:
  ```
  /MainWin[@caption='Untitled - Text Editor']//DialogBox[@caption='Find']
  ```

The general syntax of this kind of identifier is:

- Classic Agent:
  ```
  class("tag").class("tag"). ...
  ```
- Open Agent:
  ```
  class('locator').class('locator'). ...
  ```

With the Classic Agent, the recorder uses the multiple-tag settings that are stored in the **Record Window Declarations** dialog box to create the dynamic tag. In the Classic Agent example shown above, the tag for the Text Editor contains its caption as well as its window ID. For additional information, see *About Tags*.

dynamic link library (DLL)

A library of reusable functions that allow code, data, and resources to be shared among programs using the module. Programs are linked to the module dynamically at runtime.

enabling

Altering program code to handle input, display, and editing of bidirectional or double-byte languages, such as Arabic and Japanese.

exception

Signal that something did not work as expected in a script. Logs the error in the results file.

frame file

See *test frame file*.

fully qualified object name

Name that uniquely identifies a GUI object. The actual format depends on whether or not a window declaration has been previously recorded for the object and its ancestors.

group description

In the test plan editor, one or more lines in an outline that describe a group of tests, not a single test. Group descriptions by default are displayed in black.

handles

A handle is an identification code provided for certain types of object so that you can pass it to a function that needs to know which object to manipulate.

hierarchy of GUI objects

Parent-child relationships between GUI objects.

host machine

A host machine is a system that runs the Silk Test Classic software process in which you develop, edit, compile, run, and debug 4Test scripts and test plans.

Host machines are always Windows systems.

hotkey

The following table lists the available hotkeys and accelerator keys for each menu:

Menu Name	Command with Hotkey	Hotkey	Accelerator Key
Breakpoint	Toggle	Alt+B+T	F5
	Add	Alt+B+A	-
	Delete	Alt+B+D	-
	Delete All	Alt+B+E	-
Debug	Abort	Alt+D+A	-
	Exit	Alt+D+X	-
	Finish Function	Alt+D+F	-
	Reset	Alt+D+E	-
	Run and Debug/Continue	Alt+D+R	F9
	Run to Cursor	Alt+D+C	Shift+F9
	Step Into	Alt+D+I	F7
	Step Over	Alt+D+S	F8
Edit	Undo	Alt+E+U	Ctrl+Z
	Redo	Alt+E+R	Ctrl+Y
	Cut	Alt+E+T	Ctrl+X
	Copy	Alt+E+C	Ctrl+C
	Paste	Alt+E+P	Ctrl+V
	Delete	Alt+E+D	Del
	Find	Alt+E+F	Ctrl+F

Menu Name	Command with Hotkey	Hotkey	Accelerator Key
	Find Next	Alt+E+N	F3
	Replace	Alt+E+E	Ctrl+R
	GoTo Line	Alt+E+G	Ctrl+G
	GoTo Definition	Alt+E+O	F12
	Find Error	Alt+E+I	F4
	Data Driven	-	-
File	New	Alt+F+N	Ctrl+N
	Open	Alt+F+O	Ctrl+O
	Save	Alt+F+S	Ctrl+S
	Save As	Alt+F+A	-
	Save All	Alt+F+L	-
	New Project	Alt+F+W	-
	Open Project	Alt+F+E	-
	Close Project	Alt+F+J	-
	Run	Alt+F+R	-
	Debug	Alt+F+D	-
	Check out	Alt+F+T	Ctrl+T
	Check in	Alt+F+K	Ctrl+K
	Print	Alt+F+P	Ctrl+P
	Printer Setup	Alt+F+I	-
	# operation file-name	Alt+F+#	Alt+F+#
	Exit	Alt+F+X	Alt+F4
Help	Help Topics	Alt+H+H	-
	Library Browser	Alt+H+L	-
	Tutorials	Alt+H+T	-
	About Silk Test Classic	Alt+H+A	-
Include	Open	Alt+I+O	-
	Open All	Alt+I+P	-
	Close	Alt+I+C	-
	Close All	Alt+I+L	-
	Save	Alt+I+S	-
	Acquire Lock	Alt+I+A	-
	Release Lock	Alt+I+R	-
Options	General	Alt+O+G	-
	Editor Font	Alt+O+D	-
	Editor Colors	Alt+O+E	-

Menu Name	Command with Hotkey	Hotkey	Accelerator Key
	Runtime	Alt+O+T	-
	Agent	Alt+O+A	-
	Extensions	Alt+O+X	-
	Recorder	Alt+O+R	-
	Silk Performer Recorder	Alt+O+F	-
	Class Map	Alt+O+M	-
	Property Sets	Alt+O+P	-
	Silk Central URLs	Alt+O+U	-
	Open Options Set	Alt+O+O	-
	Save Options Set	Alt+O+S	-
	Close Options Set	Alt+O+C	-
	# option-file-name	Alt+O+#	-
Outline	Move Left	Alt+L+V	Alt+Left Arrow
	Move Right	Alt+L+R	Alt+Right Arrow
	Transpose Up	Alt+L+A	Alt+Up Arrow
	Transpose Down	Alt+L+S	Alt+Down Arrow
	Expand	Alt+L+E	Ctrl++
	Expand All	Alt+L+X	Ctrl+*
	Collapse	Alt+L+O	Ctrl+-
	Collapse All	Alt+L+L	Ctrl+/
	Comment	Alt+L+M	Alt+M
	Uncomment	Alt+L+N	Alt+N
Project	View Explorer	Alt+P+V	-
	Align	Alt+P+A	-
	&Left	Alt+P+L	-
	&Right	Alt+P+R	-
	Project Description	Alt+P+O	-
	Add File	Alt+P+D	-
	Remove File	Alt+P+R	-
Record	Window Declarations	Alt+R+W	Ctrl+W
	Application State	Alt+R+S	-
	Testcase	Alt+R+	Ctrl+E
	Method	Alt+R+T	-
	Actions	Alt+R+A	-
	Class	Alt+R+C	-
	Window Identifiers	Alt+R+I	Ctrl+I

Menu Name	Command with Hotkey	Hotkey	Accelerator Key
	Window Locations	Alt+R+L	-
	Silk Performer Script	Alt+R+P	-
Results	Select	Alt+T+S	-
	Merge	Alt+T+M	-
	Delete	Alt+T+D	-
	Extract	Alt+T+E	-
	Export	Alt+T+X	-
	Send to Issue Manager	-	-
	Convert to Plan	Alt+T+C	-
	Compact	-	-
	Show Summary	Alt+T+H	-
	Hide Summary	Alt+T+I	-
	View Options	Alt+T+V	-
	Goto Source	Alt+T+G	-
	View Differences	Alt+T+W	-
	Update Expected Value	Alt+T+U	-
	Pass/Fail Report	Alt+T+P	-
	Mark Failures in Plan	Alt+T+F	-
	Compare Two Results	Alt+T+O	-
	Next Result Difference	Alt+T+N	-
	Next Error Difference	Alt+T+r	-
Run	Compile	Alt+U+C	Alt+F9
	Compile all	-	-
	Run All Tests	Alt+U+R	F9
	Debug	Alt+U+D	Ctrl+F9
	Application State	Alt+U+A	Alt+A
	Testcase	Alt+U+T	Alt+T
	Show Status	Alt+U+S	-
	Abort	Alt+U+B	LShift+RShift
Testplan	Goto Script	Alt+T+G	-
	Detail	Alt+T+D	-
	Insert Template	Alt+T+I	-
	Completion Report	Alt+T+C	-
	Mark	Alt+T+M	-
	Mark All	Alt+T+A	-
	Unmark	Alt+T+U	-

Menu Name	Command with Hotkey	Hotkey	Accelerator Key
	Unmark All	Alt+T+N	-
	Mark by Query	Alt+T+Q	-
	Mary by Named Query	Alt+T+R	-
	Find Next Mark	Alt+T+F	-
	Define Attributes	Alt+T+E	-
	Manual tests	Alt+T+T	-
Tools	Link Tester	Alt+S+L	-
	Start Silk Performer	Alt+S+P	-
	Data Drive Testcase	Alt+S+D	-
	Enable Extensions	Alt+S+E	-
	Silk Central Test Manager	Alt+S+H	
View/Transcript	Expression	Alt+V+E	-
	Global Variables	Alt+V+G	-
	Local Variables	Alt+V+L	-
	Expand Data	Alt+V+X	-
	Collapse Data	Alt+V+C	-
	Module	Alt+V+M	-
	Breakpoints	Alt+V+B	-
	Call Stack	Alt+V+L	-
	Transcript	Alt+V+T	-
Window	Tile Vertically	Alt+W+T	-
	Tile Horizontally	Alt+W+H	-
	Cascade	Alt+W+C	-
	Arrange Icons	Alt+W+E	-
	Close All	Alt+W+L	-
	Next	Alt+W+N	F6
	Previous	Alt+W+P	Shift+F6
	# file-file-name	Alt+W+#	-
Workflows	Basic	Alt+K+B	-
	Data Driven	Alt+K+D	-

Hungarian notation

Naming convention in which a variable's name begins with one or more lowercase letters indicating its data type. For example, the variable name *sCommandLine* indicates that the data type of the variable is *STRING*.

identifier

Name used in test scripts to refer to an object in the application. Logical, GUI-independent name. Identifier is mapped to the tag in a window declaration.

include file

File that contains window declarations and can contain constant, variable, and other declarations.

internationalization or globalization

The process of developing a program core whose feature design and code design don't make assumptions based on a single language or locale and whose source code base simplifies the creation of different language editions of a program.

Java Database Connectivity (JDBC)

Java API that enables Java programs to execute SQL statements and interact with any SQL-compliant database. Often abbreviated as JDBC.

Java Development Kit (JDK)

A free tool for building Java applets and full-scale applications. This is an environment which contains development and debugging tools, and documentation. Often abbreviated as JDK.

Java Foundation Classes (JFC)

Sun Microsystem's and Netscape's class library designed for building visual applications in Java. Often abbreviated as JFC.

JFC consists of a set of GUI components named Swing that adopt the native look and feel of the platforms they run on.

Java Runtime Environment (JRE)

Sun Microsystem's execution-only subset of its Java Development Kit. The Java Runtime Environment (JRE) consists of the Java Virtual Machine, Java Core Classes, and supporting files, but contains no compiler, no debugger, and no tools.

The JRE provides two virtual machines: `JRE.EXE` and `JREW.EXE`. The only difference is that JREW does not have a console window.

Java Virtual Machine (JVM)

Software that interprets Java code for a computer’s operating system. A single Java applet or application can run unmodified on any operating system that has a virtual machine, or VM.

JavaBeans

Reusable software components written in Java that perform a single function. JavaBeans can be mixed and matched to build complex applications because they can identify each other and exchange information.

JavaBeans are similar to ActiveX controls and can communicate with ActiveX controls. Unlike ActiveX, JavaBeans are platform-independent.

Latin script

The set of 26 characters (A through Z) inherited from the Roman Empire that, together with later character additions, is used to write languages throughout Africa, the Americas, parts of Asia, Europe, and Oceania. The Windows 3.1 Latin 1 character set covers Western European languages and languages that use the same alphabet. The Latin 2 character set covers Central and Eastern European languages.

layout

The order and spacing of displayed text.

levels of localization

The amount of translation and customization necessary to create different language editions. The levels, which are determined by balancing risk and return, range from translating nothing to shipping a completely translated product with customized features.

load testing

Testing that determines the actual, which means not simulated, impact of multi-machine operations on an application, the server, the network, and all related elements.

localization

The process of adapting a program for a specific international market, which includes translating the user interface, resizing dialog boxes, customizing features if necessary, and testing results to ensure that the program still works.

localize an application

To make an application suitable for a specific locale: for example, to include foreign language strings for an international site.

locator

This functionality is supported only if you are using the Open Agent.

The locator is the actual name of an object, to which Silk Test Classic maps the identifier for a GUI object. You can use locator keywords to create scripts that use dynamic object recognition and window declarations.

logical hierarchy

The hierarchy that is implied from the visible organization of windows as they display to the user.

manual test

In the testplan editor, a manual test is a test that is documented but cannot be automated and, therefore, cannot be run within the test plan. You might chose to include manual tests in your test plan in order to centralize the testing process. To indicate that a test description is implemented manually, you use the keyword value manual in the testcase statement.

mark

In the testplan editor, a mark is a technique used to work with one or more tests as a group. A mark is denoted by a black stripe in the margin bar of the test plan. Marks are temporary and last only as long as the current work session. Tests that are marked can be run or reported on independently as a subset of the total plan.

master plan

In the testplan editor, that portion of a test plan that contains only the top few levels of group descriptions. You can expand, which means display, the sub-plans of the master plan, which contain the remaining levels of group description and test description. The master plan/sub-plan approach allows multi-user access to a test plan, while at the same time maintaining a single point of control for the entire project. A master plan file has a .pln extension.

message box

Dialog box that has only static text and pushbuttons. Typically, message boxes are used to prompt a user to verify an action, such as `Save changes before closing?`, or to alert a user to an error.

method

Operation, or action, to perform on a GUI object. Each class defines its own set of methods. Methods are also inherited from the class's ancestors.

minus (-) sign

In a file, an icon that indicates that all information is displayed. Click on the minus sign to hide the information. The minus sign becomes a plus sign.

modal

A dialog box that presents a task that must be completed before continuing with the application. No other part of the application can be accessed until the dialog box is closed. Often used for error messages.

modeless

A dialog box that presents a simple or ongoing task. May be left open while accessing other features of the application, for example, a search dialog box.

Multibyte Character Set (MBCS)

A mixed-width character set, in which some characters consist of more than 1 byte.

Multiple Application Domains (.NET)

The .NET Framework supports multiple application domains. A new application domain loads its own copies of the common language runtime DLLs, data structure, and memory pools. Multiple application domains can exist in one operation system process.

negative testing

Tests that deliberately introduce an error to check an application's behavior and robustness. For example, erroneous data may be entered, or attempts made to force the application to perform an operation that it should not be able to complete. Generally a message box is generated to inform the user of the problem.

nested declarations

Indented declarations that denote the hierarchical relationships of GUI objects in an application.

No-Touch (.NET)

No-Touch deployment allows Windows Forms applications, which are applications built using Windows Forms classes of the .NET Framework, to be downloaded, installed, and run directly on the machines of the user, without any alteration of the registry or shared system components.

object

The principal building block of object-oriented programs. Each object is a programming unit consisting of data and functionality. Objects inherit their methods and properties from the classes to which they belong.

outline

In the test plan editor, a structured, usually hierarchical model that describes the requirements of a test plan and contains the statements that implement the requirements. The outline supports automatic, context-sensitive coloring of test plan elements.

In Silk Test Classic, the outline is a 4Test editor mode that supports automatic, context-sensitive coloring and indenting of 4Test elements. There are two ways of using the **4Test Editor**, though Classic 4Test or Visual 4Test.

Overloaded method

A method that you call with different sets of parameter lists. Overloaded methods cause naming conflicts which must be resolved to avoid runtime errors when testing Java applications.

Example of an overloaded method	How Java support resolves the naming conflict
setBounds(int i1, int i2, int i3, int i4)	setBounds(int i1, int i2, int i3, int i4)
setBounds(RECT r1)	setBounds_2(RECT r1)

parent object

Superior object in the GUI hierarchy. A parent object is either logically associated with or physically contains a subordinate object, the child. For example, the main window physically contains the File menu as well as all other menus.

performance testing

Testing to verify that an operation in an application performs within a specified, acceptable period of time. Alternately, testing to verify that space consumption of an application stays within specified limits.

physical hierarchy (.NET)

The window handle hierarchy as implemented by the application developer.

plus (+) sign

In a file, an icon that indicates that there is hidden information. You can show the information by clicking on the plus sign. The plus sign becomes a minus sign.

polymorphism

Different classes or objects performing the same named task, but with different execution logic.

project

Silk Test Classic projects organize all the resources associated with a test set and present them visually in the **Project Explorer**, making it easy to see, manage, and work within your test environment.

Silk Test Classic projects store relevant information about your project, including references to all the resources associated with a test set, such as plans, scripts, data, option sets, .ini files, results, and frame/include files, as well as configuration information, Editor settings, and data files for attributes and queries. All of this information is stored at the project level, meaning that once you add the appropriate files to your project and configure it once, you may never need to do it again. Switching among projects is easy - since you need to configure the project only once, you can simply open the project and run your tests.

properties

Characteristics, values, or information associated with an object, such as its state or current value.

query

User-selected set of characteristics that are compared to the attributes, symbols, or execution characteristics in a test plan. When the set of characteristics matches a test, the test is marked. This is called marking by query. For example, you might run a query in order to mark all tests that are defined in the `find.t` script and that were created by the developer named Bob.

recovery system

A built-in, automatic mechanism to ensure the application is in a known state. If the application is not in the expected state, a message is logged to the results file and the problem is corrected. The recovery system is invoked before and after each test case is executed.

regression testing

A set of baseline tests that are run against each new build of an application to determine if the current build has regressed in quality from the previous one.

results file

A file that lists information about the scripts and test cases that you ran. In the testplan editor, a results file also lists information about the test plan that you ran; the format of a results file mimics the outline format of the test plan it derives from. The name of the results file is `script-name.res` or `testplan-name.res`.

script

A collection of related 4Test test cases and functions that reside in a script file.

script file

A file that contains one or more related test cases. A script file has a `.t` extension, such as `find.t`.

side-by-side (.NET)

Side-by-side execution is the ability to install multiple versions of code so that an application can choose which version of the common language runtime or of a component it uses.

Simplified Chinese

The Chinese alphabet that consists of several thousand ideographic characters that are simplified versions of traditional Chinese characters.

Single-Byte Character Set (SBCS)

A character encoding in which each character is represented by 1 byte. Single byte character sets are mathematically limited to 256 characters.

smoke test

Tests that constitute a quick set of acceptance tests. They are often used to verify a minimum level of functionality before either accepting a new build into source control or continuing QA with more in-depth, time-consuming testing.

Standard Widget Toolkit (SWT)

The Standard Widget Toolkit (SWT) is a graphical widget toolkit for the Java platform. SWT is an alternative to the AWT and Swing Java GUI toolkits provided by Sun Microsystems. SWT was originally developed by IBM and is maintained by the Eclipse Foundation in tandem with the Eclipse IDE.

statement

In the testplan editor, lines that implement the requirements of a test plan. The testplan editor has the following statements:

- testcase
- script
- testdata
- include
- attribute

Statements consist of one of the preceding keywords followed by a colon and a value.

In Silk Test Classic, a statement is a method or function call or 4Test flow-of control command, such as `if..then`, that is used within a 4Test test case.

status line

Area at the bottom of the window that displays the status of the current script, the line and column of the active window (if any), and the name of the script that is currently running. When the cursor is positioned over the toolbar, it displays a brief description of the item.

stress testing

Tests that exercise an application by repeating the same commands or operation a large number of times.

subplan

Test plan that is referenced by another test plan, normally the master test plan, by using an include statement. Portion of a test plan that resides in a separate file but can be expanded inline within its master plan. A subplan may contain the levels of group description and test description not covered in the master plan. A subplan can inherit information from its master plan. You add a subplan by inserting an include statement in the master plan. A subplan file has a `.pln` extension, as in `subplan-name.pln`.

suite

A file that names any number of 4Test test script files. Instead of running each script individually, you run the suite, which executes in turn each of your scripts and all the test cases it contains.

Swing

A set of GUI components implemented in Java that are based on the Lightweight UI Framework. Swing components include:

- Java versions of the existing Abstract Windowing Toolkit (AWT) components, such as Button, Scrollbar, and List.
- A set of high-level Java components, such as tree-view, list-box, and tabbed-pane components.

The Swing tool set lets you create a set of GUI components that automatically implements the appearance and behavior of components designed for any OS platform, but without requiring window-system-specific code.

Swing components are part of the Java Foundation Class library beginning with version 1.1.

symbols

In the testplan editor, used in a test plan to pass data to 4Test test cases. A symbol can be defined at a level in the test plan where it can be shared by a group of tests. Its values are actually assigned at either the group or test description level, depending on whether the values are shared by many tests or are unique to a single test. Similar to a 4Test identifier, except that its name begins with a $ character.

tag

This functionality is available only for projects or scripts that use the Classic Agent.

The actual name or index of the object as it is displayed in the GUI. The name by which Silk Test Classic locates and identifies objects in the application.

target machine

A target machine is a system (or systems) that runs the 4Test Agent, which is the software process that translates the commands in your scripts into GUI-specific commands, in essence, driving and monitoring your applications under test.

One Agent process can run locally on the host machine, but in a networked environment, the host machine can connect to any number of remote Agents simultaneously or sequentially.

Target machines can be Windows systems.

template

A hierarchical outline in the testplan editor that you can use as a guide when creating a new test plan. Based on the window declarations in the frame file.

test description

In the testplan editor, a terminal point in an outline that specifies a test case to be executed. Test descriptions by default are displayed in blue.

test frame file

Contains all the data structures that support your scripts:

- window declarations
- user-defined classes
- utility functions
- constants
- variables
- other include files

test case

In a script file, an automated test that ideally addresses one test requirement. Specifically, a 4Test function that begins with the testcase keyword and contains a sequence of 4Test statements. It drives an application to the state to be tested, verifies that the application works as expected, and returns the application to its base state.

In a test plan, a testcase is a keyword whose value is the name of a test case defined in a script file. Used in an assignment statement to link a test description in a test plan with a 4Test test case defined in a script file.

Test case names can have a maximum of 127 characters. When you create a data driven test case, Silk Test Classic truncates any test case name that is greater than 124 characters.

test plan

In general, a document that describes test requirements. In the testplan editor, a test plan is displayed in an easy-to-read outline format, which lists the test requirements in high-level prose descriptions. The structure can be flat or many levels deep. Indentation indicates the level of detail. A test plan also contains statements, which are keywords and values that implement the test descriptions by linking them to 4Test test cases. Large test plans can be divided into a master plan and one or more sub plans. A test plan file has a .pln extension, such as `find.pln`.

TotalMemory parameter

Total amount of memory available to the Java interpreter. This is the value returned from the `java.lang.Runtime.totalMemory()` method.

Traditional Chinese

The set of Chinese characters, used in such countries or regions as Hong Kong SAR, China Singapore, and Taiwan, that is consistent with the original form of Chinese ideographs that are several thousand years old.

variable

A named location in which you can store a piece of information. Analogous to a labeled drawer in a file cabinet.

verification statement

4Test code that checks that an application is working by comparing an actual result against an expected (baseline) result.

Visual 4Test

Visual 4Test is one of the two editors you can use with Silk Test Classic. Visual 4Test, enabled by default, is similar to Visual C++ and contains colors. Classic 4Test is similar to C and does not contain colors.

To switch between editor modes, click **Edit** > **Visual 4Test** to check or uncheck the check mark. You can also specify your editor mode on the **General Options** dialog box.

window declarations

Descriptions of all the objects in the application's graphical user interface, such as menus and dialog boxes. Declarations are stored in an include file which has a .inc extension, typically the `frame.inc` file.

window part

Predefined identifiers for referring to parts of the window. Associated with common parts of `MoveableWin` and `Control` classes, such as `LeftEdge`, `MenuBar`, `ScrollBar`.

XPath

The XML Path Language (XPath) models an XML document as a tree of nodes and enables you to address parts of the XML document. XPath uses a path notation to navigate through the hierarchical structure of the XML document. Dynamic object recognition uses a `Find` or `FindAll` function and an XPath query to locate the objects that you want to test.

Index

.NET, Classic Agent
  configuring 215
  editing configuration files 216
  editing configurations with the control panel 216
  enabling support for Windows Forms applications 208
  Infragistics .dll files 214
  Infragistics controls 214
  installing Segue.SilkTest.Net.Shared.dll 213
  no-touch Windows Forms application prerequisites 210
  no-touch Windows Forms application support 210
  recording actions on DataGrid 211
  recording actions on Infragistics toolbars 217
  recording new classes 210
  setting machine zone security 212
  setting machine zone security using command prompt 212
  setting machine zone security using control panel 213
  testing applications 208
  tips 209
operator
  testplan editor 108
+ and - operators
  rules 117

0-based arrays
  about 198
4Test
  versus native Java controls 227
4Test classes
  definition 479
4Test code
  marking as GUI specific 328
4Test components
  recording overview 132
4Test Editor
  compatible information or methods 479
  not enough characters displayed 446
4Test language
  Web application classes (Classic Agent) 279
4Test methods
  comparing to ActiveX methods 200
  comparing with native methods 255
  resolving naming conflicts with native methods 237
4test.inc
  relationship with messages sent to the result file 446

A

Abstract Windowing Toolkit
  overview 479
accented characters
  definition 479
Accessibility
  adding classes 305
  enabling for the Classic Agent 305
  improving object recognition 304
  improving object recognition (Classic Agent) 305
  removing classes 306
Accessibility classes
  removing 306
accessing
  files in projects 59
accessing data
  member-of operator 299
accessing methods
  controls 200
accessing native methods
  for predefined Java classes 252
accessing properties
  controls 200
acquiring locks 107
actions
  recording 137
active object
  highlight during recording 124
ActiveX
  controls not recognized 398
  testing 195
  troubleshooting 398
ActiveX controls
  recognized inconsistently 399
ActiveX methods
  comparing to 4Test methods 200
ActiveX support
  overview 195
ActiveX/Visual Basic
  0-based arrays 198
  disabling support 202
  enabling support 196
  exception values 201
  ignoring classes 202
  loading class definition files 202
  predefined controls 196
  prerequisites 201
  setting extension options 203
  setup for testing in the browser 204
ActiveX/Visual Basic controls
  predefined classes 196
  recording new classes 201
adding classes
  Accessibility 305
adding comments
  test plan editor 108
adding files
  projects 64
adding folders
  projects 65
adding information to the beginning of a file
  using file functions 449
adding method to TextField class
  example 319
adding properties
  recorder 436
adding Tab method to DialogBox class

example 320
advanced techniques
Classic Agent 286
agent
definition 479
unable to connect 407
agent not responding 405
agent options
differences between Classic Agent and Open Agent 48
setting for Web testing 83
setting window timeout 41
agent-based clicks
compared to API-based clicks 257
AgentClass class
classes for non-window objects 311
agents
options 20
record functionality 40
setting default 38
using both agents 39
Agents
assigning to window declarations 20
comparison 51
connecting to default 39
differences 51
driving the associated applications simultaneously 167
enabling networking 166
parameter comparison 55
parameters 55
supported methods 56
supported SYS functions 56
animation mode
test cases 367
AnyWin class
cannot extend class 438
API-based clicks
compared to agent-based clicks 257
appearance
verifying by using a bitmap 141
applet
definition 480
applets
Click() actions against custom controls are not recognized 454
controls not recognized 232, 428
identifying custom controls 254
application behavior differences
supporting 324
application environment
troubleshooting 400
application hangs
playing back a menu item pick 437
application state
definition 480
application states
behavior of based on NONE 130
overview 129
recording 136
testing 137
applications
local and single 165
preparing for automated testing 452
single and remote 165
applications with invalid data
testing 148
applying masks
exclude all differences 382
exclude some differences or just selected areas 381
AppStateList
using 357
array indexing
indexed values in test scripts 226
assigning attributes
Testplan Detail dialog box 118
attaching a comment to a result set 384
attribute definitions
modifying 119
attributes
assigning to test plans 118
defining along with values 117
defining for existing classes 315
definition 315, 480
modifying definition 119
syntax 316
test plans 116
verification 315
verifying 315
attributes and values
overview 116
attributes tag
notation 302
autocomplete
using 354
AutoComplete
AppStateList 357
customizing MemberList 354
DataTypeList 357
FAQs 355
FunctionTip 357
MemberList 358
overview 354
turning off 356
automated testing
making locators easier to recognize 452
automatically generated code
data-driven test cases 146
AWT
not all objects are visible in application 452
overview 479
popup dialog box is not recognized 454
predefined classes 233
recording menus 256
AWT applications
cannot play back picks of cascaded sub-menus 439
AWT classes
predefined 233

B

band (.NET)
definition 480
base state
about 87
definition 480

based on NONE
application state behavior 130
basic workflow
Classic Agent 43
basic workflow issue troubleshooting 47, 403
Beans
definition 490
bi-directional languages
support 348
BiDi text
definition 480
bidirectional text
definition 480
bitmap comparison
excluding parts of the bitmap 380
rules 378
bitmap differences
scanning 383
Bitmap Tool
applying a mask 381
baseline and result bitmaps 378
capturing a bitmap 375
capturing bitmaps 375
comparing bitmaps 377
designate bitmap as baseline 379
designating a bitmap as a results file 379
editing masks 381
exiting from scan mode 379
mask prerequisites 381
moving to the next or previous difference 384
opening bitmap files 380
overview 374
saving captured bitmaps 377
starting 380
starting from icon 380
starting from the results file 380
starting from the Run dialog box 380
un-setting a designated bitmap 379
using masks 380
zooming windows 379
bitmaps
analyzing 374
analyzing for differences 383
baseline 378
Bitmap Tool overview 374
capturing during recording 376
capturing Zoom window in scan mode 376
comparison command rules 378
designate as baseline 379
designate as results file 379
exiting from scan mode 379
functions 378
graphically show differences between baseline and result bitmaps 383
result 378
saving captured bitmaps 377
saving masks 383
scanning differences 383
showing areas of difference 383
starting the Bitmap Tool 380
statistics 379
un-setting designated bitmaps 379
verifying 141
verifying appearance 141
viewing statistics comparing baseline and result bitmaps 379
when to use the Bitmap Tool 375, 461
zooming in on differences 384
BOOLEAN values
converting 200
borderless tables
guidelines for recognizing 276
input elements 275
levels of recognition 277
setting options for ShowBorderlessTables 277
testing 275
breakpoints
about 393
deleting 394
setting 393
setting temporary 393
viewing 394
browser based Java applications
testing 226
browser extensions
disabling 81
browser is launched
when not testing applets 404
browser object recognition
tips 274
browser pop-up windows
handling in tests that use the Classic Agent 94
browser recognized
as client/server application 208
browser size
specifying for test frames 262
browser specifiers
testing Web applications 267
browser support
localized 350
resetting to default 351
browser test failure
troubleshooting 461
BrowserChild MainWindow not found when using Internet Explorer 7.x 405
browsers
configuring 82
playback is slow when testing applications 404
troubleshooting 404
building queries
tables 153
bytecode
definition 480

C

call stack
definition 481
calling DLLs
within 4Test scripts 307
calling nested methods
InvokeJava method 253
calling Windows DLLs from 4Test
overview 306

cannot double-click
file to open Silk Test Classic 437
cannot extend
classes 438
cannot find file agent.exe 406
cannot find items
Classic 4Test 73, 451
cannot open results file 438
cannot open Silk Test Classic
by double-clicking a file 437
cannot play back picks
AWT applications 439
cannot save files
projects 72, 450
cannot start
Silk Test Classic 72, 451
captions
GUI-specific 330
objects 300
CaptureAllClasses
example result file 243
sample scripts 242
CaptureObjectClass
example result file 242
sample scripts 241
capturing a bitmap
Bitmap Tool 375
capturing bitmaps
during recording 376
Bitmap Tool 375
categorizing test plans
overview 111
change the default number of results sets 385
changes not applied
include files or scripts 443
changing element colors
result files 385
changing tags
recorded by default 299
changing text recognition level
HtmlList controls 277
charts
presenting results 388
checking the precedence of operators 396
child object
definition 481
class
definition 481
class attribute recording
enabling 302
class attributes
adding new 303
attributes tag notation 302
deleting 304
enabling recording 302
overview 302
recording 303
class declaration filter
turning off 246
turning on 245
class declarations
declaring objects with varying classes 458, 460
class definition file
Java 227
predefined for Visual Basic 196
class definition files
loading 202, 239
class hierarchy
4Test (Classic Agent) 314
class library
definition 481
class mapping
definition 481
filtering custom classes 337
style-bits 338
style-bits overview 338
class mapping example
using style-bits 339
class methods
viewing in Library Browser 360
class not loaded error 410
class properties
NumChildren alternative 316
class-property pairs
specifying 158
classes
4Test 310
attributes 302
declarations 327
defining attributes 315
defining properties 313
defining with Classic Agent 311
hierarchy (Classic Agent) 314
logical 314
not loaded error 410
overview 310
classes for non-window objects
AgentClass 311
ClipboardClass 311
CursorClass 311
Classic 4Test
cannot find items 73, 451
definition 481
Classic Agent
adding tests to the DefaultBaseState 88
assigning name 166
assigning port 166
comparison to Open Agent 51
migrating to the Open Agent 48
overview 20
setting recording preferences 124
setting the recovery system 44, 86
starting from command line 288
Classic Agent parameters
comparison to Open Agent 55
CLASSPATH
disabling when Java is installed 255
cleanup state
recording and pasting the recording 134
Click method
actions against custom controls in Java applets are not recognized 454
client area
definition 481

client/server applications
overview 204
client/server testing
challenges 204
code for template.t 191
concurrency testing 206
configuration testing 207
configurations 161
functional testing 207
multi_cs.t script 174
multi-application testing 182
multi-testcase code template 174
parallel template 174
parallel.t script 174
serially 180
template.t explained 191
testing databases 180
types of testing 206
verifying tables 205
clients
testing concurrently 178
Clipboard methods
4Test 336
code sample 337
ClipboardClass class
classes for non-window objects 311
closing windows
recovery system 90
specifying buttons 96
specifying keys 96
specifying menus 96
code that never executes 396
collection
objects 199
columns
testing in Web applications 271
combo boxes
owner-draw 455
command line
starting Classic Agent 288
comparing
result files 384
comparing bitmaps
Bitmap Tool 377
compile errors
Unicode content 353
compiling
conditional compilation 327
compiling code
conditionally 328
completion reports
generating for test plans 107
concurrency
processing 167
concurrency testing
code example 184
explanation of code example 185
overview 206
concurrent programming
threads 169
concurrently testing
clients 178
conditional compilation
result 329
conditionally compiling code
outcome 329
configuration files, Classic Agent
editing 216
configuration test failures
troubleshooting 208
configuration testing
client/server testing 207
overview 207
configurations, Classic Agent
editing with control panel 216
configuring
deciding between machine and application 217
network of computers 165
configuring applications
custom 281
Java 221
standard 281
configuring SilkBean support
host machines when testing multiple applications 284
configuring, Classic Agent
.NET 215
contact information 19
containers
invisible 338
Control class
cannot extend class 438
control classes
not recognized 335
control is not responding 406
controls
access similar to Visual Basic 200
recognized as custom controls 331
testing for Web applications 272
verifying that no longer displayed 144
controls not recognized
ActiveX 398
coordinates
adding to declarations 334
create test case
basic workflow for the Classic Agent 43
Creating a suite 364
creating classes
mapped to several classes 321
creating data-driven test cases
workflow 145
creating masks
exclude all differences 382
exclude some differences or just selected areas 381
creating new queries
combining queries 120
creating script
both agents 39
cross-platform methods
using in scripts 326
cs.inc
overview 194
CursorClass class
classes for non-window objects 311
custom applications

configuring 281
custom classes
filtering 337
mapping (Classic Agent) 333
mapping to standard classes (Classic Agent) 332
custom controls
Classic Agent 332
identifying in applets 254
identifying in Java applications 254
Java 218
non-graphical 333
supporting 331
custom Java controls
recording classes 239
recording from scripts 240
recording using the Recorder 239
custom list boxes
support 336
custom object
definition 481
custom verification properties
defining 319
Customer Care 19
customizing results 385
CustomWin
large number of objects 255

D

data member
definition 482
data members
using properties instead 448
data source
configuring DSN 150
data sources
setting up 150
setting up for data-driven 149
data-driven
workflow 145
data-driven test case
definition 482
data-driven test cases
adding to test plans 154
automatically generated code 146
creating 151
data sources 149
overview 144
passing data to 153
running 152
running test case using sample records for each table 152
selecting test case 151
setting up data sources 150
tips and tricks 147
working with 146
data-driving test cases
Oracle 151
databases
manipulating from test cases 180
testing 180
DataTypeList
using 357
DB Tester
using with Unicode content 345
DBCS
definition 482
debugger
about 391
executing a script 391
exiting 393
menus 392
starting 392
debugging
designing and testing 391
enabling transcript 395
scripts 393
step into and step over 392
test scripts 391
tips 396
view transcripts 396
declaration tags
adding location suffices 334
declarations
adding location suffices to tags 334
adding x,y coordinates 334
definition 482
dialog boxes 291
generic message box 291
main window 293
menu 293
modified 337
modifying in the Record Window Declarations dialog box 299
overview 291
windows 295
default agent
setting 38
default browser
remote testing 428
specifying 84
default error handling 415
default Java application
enabling extensions 224
default Java executable
enabling extensions 224
DefaultBaseState
adding tests that use Classic Agent 88
definition 482
function 87
keeping DOS window open 246
wMainWindow 88
defaults.inc
overview 193
DefaultScriptEnter method
overriding 91
DefaultScriptExit method
overriding 91
DefaultTestCaseEnter method
overriding 91
DefaultTestCaseExit method
overriding 91
DefaultTestPlanEnter method
overriding 91

DefaultTestPlanExit method
overriding 91
defining
custom verification properties 319
new window 297
defining a custom verification property
example 320
defining attributes
classes 315
with values 117
defining classes
Classic Agent 311
defining custom verification properties
overview 317
defining method example
adding method to TextField class 319
defining methods
examples 319
overview 317
single GUI objects 317
defining properties
classes 313
defining symbols
Testplan detail dialog box 115
defining your own exceptions 417
deleting
class attributes 304
deleting a results set 385
Delphi
applications support 446
dependent
objects 199
deriving methods
from existing methods 319
designing and recording test cases
test cases 125
DesktopWin class
using 313
determining class
java.lang.object 411
determining where values are defined
large test plans 105
DHTML
recording popup menus 258
testing popup menus 258
diacritic
definition 482
dialog box declarations
overview 291
dialog boxes
declarations 291
displaying double-byte characters 350
specifying how to invoke 302
DialogBox class
adding Tab method example 320
Difference Viewer
about 368
definition 482
differences
moving to next or previous 384
differences between Classic Agent and Open Agent
agent options 48
differences between the Classic Agent and the Open Agent
object recognition 49
disabling extensions
browser 81
disabling support
ActiveX/Visual Basic 202
display issues
Unicode content 351
distributed testing
Classic Agent 160
client/server testing configurations 161
configuration tasks 160
configuring test environment 160
parallel processing 167
reporting distributed results 177
running tests on one remote target 175
running tests serially on multiple targets 176
setting-up extensions 195
specifying a network protocol 160
specifying target machine driven by a thread 176
statement types 172
supported networking protocols for the Classic Agent 164
troubleshooting 194
using templates 174
dividing test plans
master plan and sub-plans 105
dlls
aliasing names 307
calling from within 4Test scripts 307
common problems 439
definition 483
overview 306
passing arguments to functions 308
using support files 309
do...except
statements 329
do...except statements to trap and handle exceptions 418
do...except to handle exceptions 156
Document Object Model
advantages 268
useful information 269
Document Object Model extension
description 268
documenting manual tests
test plans 103
documenting user-defined methods
examples 361
DOM
advantages 268
useful information 269
DOM extension
description 268
HtmlPopupList crashes browser 462
DOM extensions
comparison to VO 268
setting options 262
double-byte character set
definition 482
double-byte characters
displaying 349
displaying in dialog boxes 350

displaying in the Editor 350
issues 345
double-byte files
reusing single-byte 346
downloads 19
drag and drop 458
DSN
configuring for data-driven test cases 150
Dynamic HTML
recording popup menus 258
testing popup menus 258
dynamic instantiation
definition 482
recording without window declarations 129
dynamic link library
definition 483
dynamic tables
about 270
dynamically windowed controls
working with 199

E

embedded browser applications
enabling extensions (Classic Agent) 78
enabling
definition 483
enabling extensions
automatically using basic workflow 43, 76
manually on target machines 77
Enabling extensions manually on a Host Machine 76
enabling recording
class attributes 302
enabling Sun Java plug-in 225
enabling support
ActiveX/Visual Basic 196
entering testdata statement
manually 109
error during playback
Sheridan command buttons 398
error handling
custom 415
default 415
error messages
handling differences 323
troubleshooting 405
error-handling
writing a function 421
errors
class not loaded 410
handling 415
navigating to 386
errors and the results file 368
errors when calling nested methods 414
Euro symbol
displaying 441
evenly sized tool bars
modifying icon declarations 334
evenly spaced tool bars
modifying icon declarations 334
examples
adding a method to TextField class 319
adding Tab method to DialogBox class 320
exception
defining your own 417
definition 483
handling using do...except 156
exception values
ActiveX/Visual Basic 201
errors 422
excluded characters
recording 159
replay 159
executables
GUI-specific 330
existing files with Unicode content
specifying file formats 346
existing tests
adding to projects 62
Exists method returns false when object exists 411
exporting results to a structured file for further manipulation 387
expressions
about 395
evaluating 395
using 395
extending class hierarchy
overview 310
extension dialog boxes
adding test applications 79
Extension Enabler
deleting applications 81
Extension Enabler dialog box
comparison with Extensions dialog box 81
extensions
automatically configurable 74
adding for JVM 225
disabling 81
enabling automatically using basic workflow 43, 76
enabling for AUTs 74
enabling for HTML applications 78
enabling manually on target machines 77
host machines 75
overview 74
set manually 75
setting-up for distributed testing 195
target machines 75
verifying settings 80
Extensions
deleting applications 81
Extensions dialog box
comparison with Extension Enabler dialog box 81

F

FAQs
deciding between 4Test methods and native methods 255
disabling CLASSPATH 255
invoking Java code 256
Java 255
many Java CustomWin objects 255
recording AWT menus 256
recording classes 255

saving changes to javaex.ini 255
testing JavaScript objects 256
using Java plug-in outside JVM 256
file
frame 483
include 489
file format issues
Unicode content 353
file formats
about 345
existing files with Unicode content 346
new files with Unicode content 347
file types
Silk Test Classic 63
files
adding to projects 64
moving in a project 66
removing from projects 67
files not displayed
recent files 73, 451
files not found
projects 71, 450
filtering
custom classes 337
filtering methods
cutoff classes 237
filtering properties
cutoff classes 237
filtering properties and methods
turning off class declaration filter 246
turning on class declaration filter 245
Find dialog
example test cases 459
finding values
test cases 152
fix incorrect values in a script 386
folders
adding to projects 65
available controls 65
moving in a project 66
removing from projects 66
renaming in projects 66
font pattern database
generating 343
fonts
displaying differently 352
specifying for test frames 262
forward case-sensitive search
setup example 153
frame declarations
streamlining 260
frame file
definition 483
frequently asked questions
deciding between 4Test methods and native methods 255
disabling CLASSPATH 255
invoking Java code 256
Java 255
recording AWT menus 256
recording classes 255
saving changes to javaex.ini 255
testing JavaScript objects 256
to many Java CustomWin objects 255
using Java plug-in outside JVM 256
Frequently Asked Questions
AutoComplete 355
Fully customize a chart 388
fully qualified object name
definition 483
functional test design
incremental 206
functional testing
overview 207
functionality not supported
Open Agent 406
functions
troubleshooting 410
FunctionTip
using 357

G

general protection faults
troubleshooting 442
generating completion reports
test plans 107
generating pass/fail reports
test plan results file 389
generic message box
declarations 291
generic message box declaration
overview 291
GetMachineData
multi-application testing example 185
GetText
code sample 337
getting started
Silk Test Classic 18
global and local variables with the same name 396
global variables
GUI specifiers 328
overview 168
protecting access 169
running from test plan versus running from script 442
global variables with unexpected values 396
globalization
definition 489
glossary
overview 479
graphical controls
support 332
group description
definition 483
groups
sharing projects 59
GUI objects
hierarchy 484
recording methods 318
GUI specifiers
4Test code 328
global variables 328
inheritance 328
overview 291, 327

syntax 329
usages 329
GUI-specific captions
support 330
GUI-specific executables
supporting 330
GUI-specific menu hierarchies
support 331
GUI-specific objects
support 330
GUI-specific tags
creating 322

H

handles
definition 484
handling GUI differences
porting tests 321
hidecalls
keyword 316
hierarchical object recognition
overview 123
hierarchy of GUI objects
definition 484
host machine
definition 484
host machines
configuring SilkBean support when testing multiple applications 284
hotkey
definition 484
HTML applications
enabling extensions 78
Html class attributes
adding new 303
deleting 304
recording 303
HTML definitions
tables 271
HTML frame declarations
streamlining 260
HTML frames
declarations 271
HtmlPopupList
browser crashes when using DOM 462
Hungarian notation
definition 488

I

identifier
definition 489
identifiers
overview 300
identifiers and tags
overview 299
XML objects 278
ignored Java objects
recording classes 244
ignoring
Java objects 254
ignoring classes
ActiveX/Visual Basic 202
Java 443
images
testing in Web applications 272
IME
using 351
IME issues
Unicode content 353
IMEs
differing in appearance 353
improving
window declarations 295
improving object recognition
Accessibility 304
Accessibility (Classic Agent) 305
improving recognition
defining new window 295
improving recording
defining a new window 297
include file
definition 489
include files
changes not applied 443
conditionally loading 322
handling very large files 194
loading for different test application versions 322
maximum size 194
include scripts
changes not applied 443
incorrect use of break statements 397
incorrect values for loop variables 397
incremental test design
functional 206
index
using as tag 327
indexed values
incorrect method returns in scripts 415
indexing
schemes for 4Test and native Java methods 226
infinite loops 397
Infragistics controls, Classic Agent
.NET 214
Infragistics, Classic Agent
.dll files 214
inheritance
GUI specifiers 328
input elements
borderless tables 275
Input Method Editor
setting up 349
Input Method Editor issues
Unicode content 353
Input Method editors
differing in appearance 353
Input Method Editors
using 351
installing language support
Unicode content 349
international applications
recording identifiers 348
internationalization
configuring environment 348
definition 489
useful sites 345

internationalized content
issues with displaying 345
internationalized objects
support 344
invalid data
testing applications 148
invisible containers
about 338
filtering unnecessary classes 337
Invoke method
how to write the method 413
InvokeJava method
class not loaded error 410
invokeMethods
drawing line in multiline text field 252
invokeMethods method
JavaMainWin 247
invoking
dialog boxes 302
Java applications and applets 248
Java code from 4Test scripts 256
invoking applets
Java 248
invoking applications
Java 249
JRE 249
JRE using -classpath 250
invoking test cases
multi-application environments 183

J

JAR files
choosing 401
Java
accessing native methods for predefined classes 252
accessing non-visible objects 253
accessing objects and methods 252
applet controls not recognized 232, 428
calling nested native methods 253
cannot open Web Start application 400
controls are not recognized 453
disabling CLASSPATH 255
enabling support 221
FAQs 255
identifying custom controls 254
ignoring classes 443
ignoring objects 254
invoking applets 248
invoking applications 249
invoking from 4Test scripts 256
javaex.inc 227
launching through .lax file 225
no child objects are recognized 453
predefined class definition file 227
recording classes for custom controls 239
recording custom controls from scripts 240
recording custom controls with the Recorder 239
recording window declarations 245
security privileges 223
setting extension options using javaex.ini 230
testing custom window classes 238
testing scroll panes 254
when to record classes 235
window properties not captured for stand-alone applications 455
Java applets
disabling plug-ins 232, 427
invoking 248
setup for testing in the browser 204
supported browsers 219
Java Applets
configuring 222
Java application error
running from batch file 400
Java applications
Silk Test Java file missing in plug-in 232, 428
application not ready to test 231, 426
configuring standalone applications 222
defining lwLeaveOpen for launching 412
enabling 224
enabling extensions 225
enabling plug-ins 231, 427
identifying custom controls 254
Java Plug-in check box not checked 232, 427
keeping DOS window open 246
multitags 227
prerequisites 221
recording classes for ignored objects 244
standard names 80
testing browser-based 226
troubleshooting 231, 426
writing an Invoke method for launching 413
Java applications and applets
invoking 248
preparing for testing 226
testing 226
Java AWT
classes for the Classic Agent 219
Classic Agent 217
object recognition 219
supported controls 219
Java AWT menus
playing back 219
recording 219
Java AWT/Swing
testing standard Java objects 218
Java classes
accessing native methods 252
ignoring 443
loading class definition files 239
loading test frame files 239
recording 235
Java console
redirect output to file 246
Java controls
not recognized 453
Java controls not recognized 455
Java custom windows
testing classes 238
Java database connectivity
definition 489
using 247
Java Development Kit

definition 489
Java extension
enabling 221
options 228
java extension loses injection when using VNC 403
Java FAQs
overview 255
Java Foundation Class
playing back menus 218
recording menus 218
Java Foundation Classes
definition 489
Java objects
accessing nested 253
accessing non-visible 253
determining class 411
ignoring 254
Java output
redirect from console to file 246
Java plug-in
using outside JVM 256
Java plugins
enabling 224
Java Runtime Environment
definition 489
Java scroll panes
testing 254
Java security policy
changing 223
Java security privileges
changing 223
Java support
disabling 224
enabling 221
extending 246
manually configuring for Sun JDK 222
Sun JDK 222
supported classes 232
Java SWT and Eclipse
Classic Agent 256
Java SWT custom class attributes
deleting 304
Java Virtual Machine
definition 489
Java virtual machines
supported 219
Java-equivalent window classes
predefined 233
JavaBeans
definition 490
support 220
javaex.ini
saving changes 255
setting Java extension options 230
JavaMainWin not recognized 452
JavaScript
not recognizing updates 454
support 220
testing 256
JBuilder
configuring Silk Test Classic 223
trouble configuring 400
JDBC
definition 489
using 247
JDeveloper
configuring Silk Test Classic 223
trouble configuring 400
JDK
definition 489
invoking applications 249
JFC
definition 489
playing back menus 218
recording menus 218
JFC classes
predefined 233
JFC objects
mouse clicks fail 436
JFC popup menus
sample declarations 401
sample script 401
JRE
definition 489
invoking applications 249
invoking applications using -classpath 250
JVM
definition 489
supported 219

K

keywords
hidecalls 316

L

Language bar
only English listed 353
large test plans
determining where values are defined 105
overview 105
Latin script
definition 490
launcher application executables
support 225
launching Java applications
writing Invoke method 413
layout
definition 490
legacy scripts
options 456
Library Browser
adding information 359
adding user-defined files 360
not displaying user-defined methods 444
not-displayed Web classes 361
overview 358
source file 359
viewing class methods 360
viewing functions 360
Web browser classes not displayed 404
licenses
handling limited licenses 194

licensing
available license types 17
linking descriptions to scripts
Testplan Details dialog box 109
linking descriptions to test cases
Testplan Details dialog box 109
linking test plans to test cases
example 111
links
not recognized 462
testing 273
list box
cannot record second window 439
list boxes
custom 336
owner-draw 455
load testing
definition 490
loading include files
conditionally 322
local applications
single 165
local sub-plan copies
refreshing 106
localization
definition 490
localization levels
definition 490
localized browser support
changing include files 351
localized browsers
changing default browser include files 351
resetting support to default 351
support 350
localizing applications
definition 490
locally testing multiple applications
sample include file (Classic Agent) 430
sample script file (Classic Agent) 429
location suffix
adding to declarations tag 334
locator
definition 490
locator attributes
excluded characters 159
Windows API-based controls 280
locator recognition
enhancing 452
locks
acquiring 107
overview 107
releasing 107
test plans 107
logging Elapsed Time, Thread, and Machine Information 388
logging errors
programmatically 418
logic errors
evaluating 368
logical controls
different implementations 323
logical hierarchy
definition 491
login windows
handling 92
non-Web applications (Classic Agent) 93
Web applications 92
looking at statistics
bitmaps 379
lwLeaveOpen
defining for launching Java applications 412
specifying windows to be left open (Classic Agent) 95

M

machine handle operator
specifying 177
machine handle operators
alternative syntax 178
machine zone security, Classic Agent
setting 212
setting using command prompt 212
setting using control panel 213
main function
using in scripts 155
main window
declarations 293
manual test
definition 491
describing the state 103
manual test state
describing 103
mapping custom classes
to standard classes (Classic Agent) 332
mark
definition 491
marked tests
printing 112
marking commands
interactions 112
marking failed testcases 386
masks
applying 381
creating one that excludes all differences 382
creating one that excludes some differences or selected areas 381
editing 381
prerequisites 381
saving 383
master plan
definition 491
master plans
connecting with sub-plans 106
maximum size
Silk Test Classic files 444
MBCS
definition 492
member-of operator
using to access data 299
MemberList
customizing 354
using 358
menu
declarations 293

menu commands
cannot access 437
menu hierarchies
GUI-specific 331
menu item pick
application hangs during playback 437
merging results 386
message box
definition 491
messages sent to the result file
relationship with exceptions defined in 4test.inc 446
method
definition 491
methods
adding to existing classes 317
adding to single GUI objects 317
Agent support 56
defining 317
defining for single GUI objects 317
deriving new from existing 319
enumerating 237
recording for GUI objects 318
redefining 319
thresholds for filtering 237
troubleshooting 410
methods return incorrect indexed values in scripts 415
Microsoft Accessibility
improving object recognition 304
migrating
from the Classic Agent to the Open Agent 48
minus (-) sign
definition 491
modal
definition 492
modeless
definition 492
modified declarations
using 337
modify declarations
icons contained in an evenly sized and spaced tool bar 334
modifying declarations
Record Window Declarations dialog box 299
modifying identifiers
test frames 262
modules
viewing a list 396
mouse actions
playing back 444
mouse coordinate
off screen 462
MoveableWin
cannot extend class 438
moving files
between projects 67
on Files tab 66
moving folders
in a project 66
multi-application environments
cs.inc 194
multi-application testing
code for template.t 191
invoking example 191
invoking example explained 191
invoking test cases 183
overview 182
template.t explained 191
multi-machine testing
Terminal Server environment 181
multi-test case
statements 183
multibyte character set
definition 492
Multiple Application Domains (.NET)
definition 492
multiple applications
setting up the recovery system 428
multiple machines
driving 169
troubleshooting 428
multiple tag recording
turning off 299
multiple tests
recovering 168
multiple verifications
test cases 419
multiple-application environments
test case structure 182
multitags
Java applications 227

N

Named Query command
differences with Query 121
naming conflicts
resolving 237
native Java controls
versus 4Test 227
native Java methods
comparing with 4Test methods 255
native methods
accessing for predefined Java classes 252
enumerating 237
resolving naming conflicts with 4Test methods 237
using non-enumerated methods 237
native properties
enumerating 237
native Visual Basic objects
displayed as custom windows 399
navigating to errors 386
negative testing
definition 492
nested declarations
definition 492
nested Java objects
accessing 253
NetBIOS host
enabling networking 166
networking protocols 164
network
configuring 165
network testing
types of testing 206

networking
supported protocols for the Classic Agent 164
networks
enabling 166
enabling on Agents 166
enabling on NetBIOS host 166
enabling on remote host 167
new files with Unicode content
specifying file formats 347
no methods found
Visual Basic 399
no properties found
Visual Basic 399
no-touch (.NET)
definition 492
no-touch applications, Classic Agent
prerequisites 210
Windows Forms 210
non-graphical custom controls
support 333
non-visible Java objects
accessing 253
non-visible objects
accessing in Java 253
non-Web applications
handling login windows (Classic Agent) 93
not all actions captured
recorder 445
not enumerated methods
using 237
not recognizing updates on Internet Explorer page containing JavaScript 454
notation
attributes tag 302
notification testing
code example 1 187
code example 2 190
explanation of code example 1 189
explanation of code example 2 190
single-user example 187
single-user example explanation 189
two-user example 190
two-user example explanation 190
NumChildren
alternative class property 316

O

object
definition 492
object attributes
verifying 140
object files
advantages 289
locations 290
overview 289
object locations
recording 137
object properties
overview 138
verifying 138
verifying (Classic Agent) 139
object recognition
control class not recognized 335
differences between the Classic Agent and the Open Agent 49
hierarchical 123
improving by defining new window 295
improving with Accessibility 304
improving with Accessibility (Classic Agent) 305
Java AWT 219
objects recognized as custom controls 332
Swing 219
object-oriented programming languages
classes 221
objects
captions 300
collection 199
dependent 199
internationalized 344
not visible within application 452
properties 138
steps for verifying attributes 140
troubleshooting 435
verifying attributes 140
verifying properties 138
verifying properties (Classic Agent) 139
verifying state 141
verifying with the Verify function 139
objects recognized as custom controls
reasons 332
OCR
4Test functions 340
generating the font pattern database 343
overview 340
pattern file generation 343
SGOCRLIB.DLL 343
support 340
OCR module
files 340
overview 340
OLESSCommand class
error during click playback 398
Open Agent
comparison to Classic Agent 51
migrating to from Classic Agent 48
Open Agent parameters
comparison to Classic Agent 55
opening
TrueLog Options dialog box 372
opening projects
existing 62
operators
precedence 396
OPT_AGENT_CLICKS_ONLY
option 21
OPT_ALTERNATE_RECORD_BREAK
option 21
OPT_APPREADY_RETRY
option 21
OPT_APPREADY_TIMEOUT
option 21
OPT_BITMAP_MATCH_COUNT
option 21

OPT_BITMAP_MATCH_INTERVAL
option 22
OPT_BITMAP_MATCH_TIMEOUT
option 22
OPT_BITMAP_PIXEL_TOLERANCE
option 23
OPT_CLASS_MAP
option 23
OPT_CLOSE_CONFIRM_BUTTONS
option 23
OPT_CLOSE_DIALOG_KEYS
option 23
OPT_CLOSE_MENU_NAME
option 23
OPT_CLOSE_WINDOW_BUTTONS
option 23
OPT_CLOSE_WINDOW_MENUS
option 24
OPT_CLOSE_WINDOW_TIMEOUT
option 24
OPT_COMPATIBILITY
option 24
OPT_COMPATIBLE_TAGS
option 24
OPT_COMPRESS_WHITESPACE
option 24
OPT_DROPDOWN_PICK_BEFORE_GET
option 25
OPT_ENABLE_ACCESSIBILITY
option 25
OPT_ENSURE_ACTIVE_WINDOW
option 26
OPT_EXTENSIONS
option 26
OPT_GET_MULTITEXT_KEEP_EMPTY_LINES
option 26
OPT_ITEM_RECORD
option 26
OPT_KEYBOARD_DELAY
option 26
OPT_KEYBOARD_LAYOUT
option 26
OPT_KILL_HANGING_APPS
option 27
OPT_LOCATOR_ATTRIBUTES_CASE_SENSITIVE 27
OPT_MATCH_ITEM_CASE
option 27
OPT_MENU_INVOKE_POPUP
option 27
OPT_MENU_PICK_BEFORE_GET
option 27
OPT_MOUSE_DELAY
option 28
OPT_MULTIPLE_TAGS
option 28
OPT_NO_ICONIC_MESSAGE_BOXES
option 28
OPT_PAUSE_TRUELOG
option 28
OPT_PLAY_MODE
option 28
OPT_POST_REPLAY_DELAY
option 29
OPT_RADIO_LIST
option 29
OPT_RECORD_LISTVIEW_SELECT_BY_TYPEKEYS
option 29
OPT_RECORD_MOUSE_CLICK_RADIUS
option 29
OPT_RECORD_MOUSEMOVES
option 29
OPT_RECORD_SCROLLBAR_ABSOLUT
option 29
OPT_REL1_CLASS_LIBRARY
option 30
OPT_REMOVE_FOCUS_ON_CAPTURE_TEXT
option 30
OPT_REPLAY_HIGHLIGHT_TIME
option 30
OPT_REPLAY_MODE
option 30
OPT_REQUIRE_ACTIVE
option 30
OPT_SCROLL_INTO_VIEW
option 31
OPT_SET_TARGET_MACHINE
option 31
OPT_SHOW_OUT_OF_VIEW
option 31
OPT_SYNC_TIMEOUT
option 31
OPT_TEXT_NEW_LINE
option 32
OPT_TRANSLATE_TABLE
option 32
OPT_TRIM_ITEM_SPACE
option 32
OPT_USE_ANSICALL
option 32
OPT_USE_SILKBEAN
option 32
OPT_VERIFY_ACTIVE
option 32
OPT_VERIFY_APPREADY
option 33
OPT_VERIFY_CLOSED
option 33
OPT_VERIFY_COORD
option 33
OPT_VERIFY_CTRLTYPE
option 33
OPT_VERIFY_ENABLED
option 33
OPT_VERIFY_EXPOSED
option 34
OPT_VERIFY_RESPONDING
option 34
OPT_VERIFY_UNIQUE
option 34
OPT_WAIT_ACTIVE_WINDOW
option 34
OPT_WAIT_ACTIVE_WINDOW_RETRY
option 35
OPT_WINDOW_MOVE_TOLERANCE

option 35
OPT_WINDOW_RETRY
option 35
OPT_WINDOW_SIZE_TOLERANCE
option 36
OPT_WINDOW_TIMEOUT
option 36
OPT_WPF_CUSTOM_CLASSES
option 37
OPT_WPF_PREFILL_ITEMS
option 37
OPT_XBROWSER_SYNC_EXCLUDE_URLS
option 38
OPT_XBROWSER_SYNC_MODE
option 37
OPT_XBROWSER_SYNC_TIMEOUT
option 38
optical character recognition
4Test functions 340
OCR module 340
overview 340
pattern file generation 343
SGOCRLIB.DLL 343
options
agents 20
sets 323
options set
adding to projects 62
editing in projects 63
including in projects 62
using in projects 62
options sets
porting 323
specifying 324
Oracle DSN
data-driving test cases 151
Oracle Forms
application support 435
organizing
projects 64
outline
definition 493
overriding
default recovery system 91
owner-draw
list boxes and combo boxes 455

P

packaged projects
emailing 70
packaging
projects 68
parallel processing
spawn statement 173
statements 172
parallel statements
using 173
parallel test cases
using templates 174
parallel testing
asynchronous 171
parent object
definition 493
pass/fail chart
creating 389
passing arguments
scripts 364
to DLL functions 308
passing data
data-driven test cases 153
pattern file generation
OCR 343
optical character recognition 343
peak load testing 207
performance testing
definition 493
physical hierarchy (.NET)
definition 493
plug-ins
enabling for Java applications 231, 427
plus (+) sign
definition 493
polymorphism
concept 311
definition 493
popup list
cannot record second window 439
port numbers
Classic Agent 194
porting tests
another GUI 321
differences between GUIs 321
predefined ActiveX/Visual Basic controls
list 196
predefined attributes
test plan editor 116
predefined classes
ActiveX/Visual Basic controls 196
AWT 233
prerequisites
testing Java applications 221
printing
marked tests 112
priorLabel
Win32 technology domain 282
privileges required
Silk Test Classic 441
Product Support 19
product updates 19
project
definition 494
Project Explorer
overview 60
sorting resources 67
turning on and off 67
Unicode characters do not display 352
project files
editing 73, 451
not loaded 72, 450
project-related information
storing 58
projects
about 58
accessing files 59
adding an options set 62

adding existing tests 62
adding files 64
adding folders 65
cannot load project file 72, 450
cannot save files 72, 450
creating 43, 61
editing project files 73, 451
editing the options set 63
emailing packaged projects 70
exporting 71
files not found 71, 450
including an options set 62
moving files between 67
moving files in projects 66
moving folders in projects 66
opening existing projects 62
organizing 64
packaging 68
removing files 67
removing folders 66
renaming 65
renaming folders 66
sharing among a group 59
storing information 58
troubleshooting 71, 449
turning Project Explorer on and off 67
viewing associated files 68
viewing resources 68
working with folders 65
properties
definition 494
enumerating 237
objects 138
sets 156
thresholds for filtering 237
using instead of data members 448
verifying 315
verifying as sets 156
property list
confirming 319
property sets
combining 157
creating 157
deleting 157
editing 157
overview 156
predefined 158
protocols
networking (Classic Agent) 164

Q

queries
building 153
combining 122
combining to create new 120
creating 121
deleting 122
editing 122
including symbols 120
test plans 119
query
definition 494
Query command
differences with Named Query 121
Quick Start Wizard
cannot find 438

R

recent files
files not displayed 73, 451
recognizing borderless tables
guidelines 276
recognizing controls
as custom controls 331
recognizing objects in browsers
tips 274
recorder
adding properties 436
does not capture all actions 445
recording
4Test components 132
actions 137
application states 136
available functionality 40
AWT menus 256
changing tags recorded by default 299
classes for custom Java controls 239
cleanup stage 134
Html class attributes 303
Java classes 235
Java window declarations 245
methods for GUI objects 318
object highlighting 124
object locations 137
pasting recording from cleanup 134
remote 168
test frames 289
window identifiers 138
without window declarations 129
recording a Close method
Classic Agent 96
Recording a window declaration for a dialog 296
recording browser-page declarations
many child objects 463
recording classes
custom Java controls 239
enumerating methods and properties 237
ignored Java objects 244
when to record classes 235
recording identifiers
international applications 348
recording new classes
ActiveX/Visual Basic controls 201
recording popup menus
DHTML 258
Dynamic HTML 258
recording preferences
setting for Classic Agent 124
recording test cases
Classic Agent 45, 133
linking to scripts and test cases 135
recording test frames

Web applications 259
recording the stages
test cases 131
recording window declarations
main window 296
menu hierarchy 296
only the Java applet is seen 453
Web applications 259
recording, Classic Agent
classes for .NET controls 210
DataGrid actions 211
Infragistics toolbars actions 217
recovery system
Classic Agent 85
closing windows 90
defaults.inc file 193
definition 494
flow of control 89
modifying 91
overriding default 91
setting for the Classic Agent 44, 86
specifying new window closing procedures 95
starting the application 90
testing ability to close application dialog boxes 135
Web applications 89, 227
regression testing
definition 494
releasing locks 107
remote applications
multiple 165
networking 165
single 165
remote testing
default browser 428
removing the unused space in a results file 388
renaming
projects 65
replacing values
test cases 152
reporting
distributed results 177
reports
presenting results 388
reraise statement
error handling 415
resolving
naming conflicts 237
result files
changing the color of elements 385
comparing 384
converting to test plans 101
using 384
results
customize a chart 388
customizing 385
deleting a set 385
displaying a different set 390
errors and results file 368
exporting to a structured file 387
fixing incorrect values in a script 386
interpreting 367
logging Elapsed Time, Thread, and Machine Information 388
marking failed testcases 386
merging 386
merging overview 370
presenting 388
removing unused space in the results file 388
results file overview 367
sending to Issue Manager 388
starting the Bitmap Tool from the results file 380
storing 387
storing and exporting 387
testplan pass-fail report and chart 370
viewing an individual summary 387
results file
definition 494
overview 367
running a testplan 366
running from batch file
Dr. Watson error 400
running global variables
test plan versus script 442
running test cases
data driven 152
running tests
overview 364
running the currently active script or suite 366
Runtime
about 468
comparing with Silk Test Classic 468
installing 468
starting 468

S

sample applications
Web applications 257
sample command line
Visual Café 250
sample scripts
CaptureAllClasses 242
CaptureObjectClass 241
saving captured bitmaps
Bitmap Tool 377
saving changes
sub-plans 107
saving existing files
Save as dialog box opens 353
script
definition 494
script deadlocks
4Test handling 207
script file
definition 495
script files
saving 136
ScriptEnter method
overriding default recovery system 91
ScriptExit method
overriding default recovery system 91
scripting
common problems 440

scripts
deadlock handling 207
linking to by recording a test case 135
methods return incorrect indexed values 415
passing arguments to 364
saving 136
using main function 155
search setup example
forward case-sensitive search 153
security privileges
Java 223
selecting test cases
to data drive 151
sending results directly to Issue Manager 388
serial number 19
Set attributes
adding members 117
removing members 117
SetText
code sample 337
SetText() statements
recording two 445
setting agent options
Web testing 83
setting default Agent
Runtime Options dialog box 39
toolbar 39
setting extension options
ActiveX/Visual Basic 203
setting Java extension options
javaex.ini 230
setting options
TrueLog 372
TrueLog Explorer 372
setting the recovery system
Classic Agent 44, 86
setting up IME
Unicode content 349
setting up the recovery system
multiple local applications 428
setup steps
using the Classic Agent to test Web applications 259
SGOCRLIB.DLL
OCR 343
optical character recognition 343
shared data
specifying 108
sharing initialization files
test plans 106
Sheridan command buttons
error during click playback 398
Show All Classes check box
not all objects are visible 452
show areas of difference between a baseline and a result bitmap
graphically 383
ShowBorderlessTables
setting options 277
ShowListItem option
setting 277
side-by-side (.NET)
definition 495
Silk Test Classic
about 18
not starting 72, 451
Silk Test Classic files
maximum size 444
SilkBean
configuring on target UNIX machines 283
configuring support on host machine when testing multiple applications 284
overview 282
preparing test scripts 282
troubleshooting 284
SilkBean support
target UNIX machines 283
Simplified Chinese
definition 495
single applications
local 165
remote 165
single GUI objects
defining methods 317
single-application environments
test case structure 183
single-application tests
recovery-system file 193
single-byte character set (SBCS) 495
single-byte files
reusing as double-byte 346
smoke test 495
sorting resources
Project Explorer 67
spawn
multi-application testing example 185
spawn statement
using 173
specifiers
GUI 291
specifying
target machine for a single command 177
specifying attribute hierarchy
adding new Html class attributes 303
recording existing Html class attributes 303
specifying browser
testing Web applications 83
specifying new window closing procedures
recovery system 95
specifying windows to be left open
Classic Agent 95
SSTab control
tag declarations 278
standard applications
configuring 281
Standard Widget Toolkit (SWT) 495
starting
command line 286
starting Bitmap Tool
from icon 380
from the results file 380
starting from the command line
Silk Test Classic 286
starting Java applications through command line
troubleshooting 231, 426

starting the Bitmap Tool
Run dialog box 380
statement
definition 495
statements
do...except 329
parallel 173
type 330
status line 496
step into 392
step over 392
stopping a running testcase before it completes 366
storing and exporting results 387
storing results 387
str function
does not round correctly 449
stress testing 496
style-bits
class mapping 338
class mapping example 339
overview 338
using with class mapping 338
sub-menus of a Java menu are being recorded as JavaDialogBoxes 436
sub-plans
connecting with master plans 106
copying 106
opening 106
refreshing local copies 106
saving changes 107
subplan
definition 496
suite
creating 364
definition 496
Sun Java
enabling plug-in 225
Sun JDK
Java support 222
manually configuring Java support 222
supported browsers
testing Java applets 219
supported controls
Java AWT 219
Swing 219
Web applications 257
supported Java classes
overview 232
SupportLine 19
suppressing controls
Classic Agent 213, 256, 280
Open Agent 280
Swing
Classic Agent 217
definition 496
object recognition 219
supported controls 219
Symantec Visual Café
invoking applications from command line 250
invoking applications from IDE 251
Symantex Itools classes
predefined 234
symbols
assigning values 115
definition 496
including in queries 120
overview 113
specifying as arguments for testcase statements 115
using 113
symbolvalue
assigning to symbol 115
synchronizing threads with semaphores 170
system dialog boxes
cannot display multiple languages 352

T

table recognition 266
tables
building queries 153
dynamic 270
HTML definitions 271
testing in Web applications 271
verifying in client/server applications 205
tag
definition 496
tag declarations
SSTab control 278
tags
deciding on which form to use 323
overview 301
specifying 297
using index as tag 327
tags and identifiers
overview 260
target machine
definition 497
target machines
manually enabling extensions 77
template
definition 497
templates
test plans 99
Terminal Server
multi-machine testing 181
overview 181
test application settings
copying 80
test applications
adding to extension dialog boxes 79
deleting from Extension Enabler dialog box 81
deleting from Extensions dialog box 81
duplicating settings 80
loading different include files for different application versions 322
test case
definition 497
test case example
word processor feature 130
test case structure
multiple-application environments 182
single-application environments 183
test cases
about 125

anatomy of basic test case 126
constructing 127
data 128
data-driven 144
designing 126
designing and recording, Classic Agent 123
example word processor feature 130
finding and replacing values 152
linking to by recording a test case 135
overview 125
overview of recording the stages 131
recording overview (Classic Agent) 131
recording with the Classic Agent 45, 133
running 46, 365
running data driven 152
running in animation mode 367
saving 128
types 126
verifying 133
with multiple verifications 419
test description
definition 497
test frame file 497
test frame files
loading 239
test frames
files 261
modifying identifiers 262
overview 261
recording 289
saving 301
specifying browser size 262
specifying fonts 262
specifying username and password 262
Web applications 260
test plan 498
test plan editor
adding comments 108
predefined attributes 116
symbol definition statements 114
test plan outlines
change levels 102
indent levels 102
test plan queries
overview 119
test plan results
adding comments 102
generating pass/fail reports 389
test plan templates
inserting 103
test plans
acquiring and releasing locks 107
adding comments to results 102
adding data 108
adding data-driven test cases 154
assigning attributes and values 118
attributes and values 116
categorizing 111
changing colors 104
connecting sub-plans with master plans 106
converting results files to test plans 101
copying sub-plans 106
creating 101
creating sub-plans 106
dividing into master plan and sub-plans 105
documenting manual tests 103
editor statements 108
example outline 99
generating completion reports 107
indent and change levels in outlines 102
inserting templates 103
large test plans 105
linking 109
linking manually to a test plan 110
linking scripts to using the Testplan Detail dialog box 110
linking test cases to using the Testplan Detail dialog box 110
linking to data-driven test cases 110
linking to scripts 104, 110
linking to test cases 104, 110
linking to test cases example 111
locks 107
marking 112
marking tests 112
marking-command interactions 112
opening sub-plans 106
overview 98
predefined attributes 116
printing marked tests 112
queries 119
refreshing local sub-plan copies 106
sharing initialization files 106
stopping 448
structure 98
templates 99
user defined attributes 116
working with 101
test results
interpreting 367
reporting 177
viewing 46, 368
test scripts
debugging 391
preparing for SilkBean 282
test-cases
working with data-driven 146
testcase statements
specifying symbols as arguments 115
TestCaseEnter method
defining 412
overriding default recovery system 91
TestCaseExit method
defining 412
overriding default recovery system 91
testcases
designing 126
overview 125
types 126
testdata statement
entering manually 109
entering with Testplan Details dialog box 109
testing
application states 137

concurrency 206
configuration 207
databases 180
driving multiple machines 169
functional 207
peak load 207
strategies 205
volume 207
testing .NET applications
Classic Agent 208
testing applications
Classic Agent 160
invalid data 148
SilkBean 282
testing asynchronous in parallel 171
testing controls
comparing 4Test methods with ActiveX methods 200
Web applications 272
testing images
Web applications 272
testing in the browser
setup for ActiveX or Java applets 204
testing Java
configuring Silk Test Classic 221
prerequisites 221
testing links
Web applications 273
testing methodology
Web applications 266
testing multiple applications
configuring SilkBean support 284
overview 182
window declarations 183
testing multiple machines
overview 175
running tests serially on multiple targets 176
testing on multiple machines
Classic Agent 160
testing popup menus
DHTML 258
Dynamic HTML 258
testing recovery system
closing application dialog boxes 135
testing serially
client and server 180
testing text
Web applications 273
testing Web applications
Classic Agent setup steps 259
different browsers 267
methodology 266
specifying browser 83
testing text 273
Web page objects 268
testing XML
overview 278
Testplan Detail dialog box
defining symbols 115
linking scripts to test plans 110
linking test cases to test plans 110
Testplan Details dialog box
entering testdata statement 109
linking descriptions to scripts and test cases 109
testplan editor
operator 108
Testplan Editor
predefined attributes 116
statements 108
testplan pass-fail report and chart 370
testplan queries
overview 119
TestPlanEnter method
overriding default recovery system 91
TestPlanExit method
overriding default recovery system 91
tests
marking 112
porting to another GUI 321
running 364
running and interpreting results 364
text boxes
custom 335
Return key 326
text click recording
overview 362
text field
not allowing input 448
text fields
custom 335
return key 326
text recognition
overview 362
threads
concurrent programming 169
specifying target machines 176
synchronizing with semaphores 170
tips
recognizing objects in browsers 274
tips and tricks
data-driven test cases 147
TotalMemory parameter 498
Traditional Chinese 498
transcript
enabling 395
trapping the exception number 417
troubleshooting
4Test Editor does not display enough characters 446
ActiveX/Visual Basic 398
application environment 400
basic workflow issues 47, 403
browser is launched when not testing applets 404
browsers 404
cannot work with JBuilder or JDeveloper 400
Classic Agent 398
configuration test failures 208
custom error handling 415
error messages 405
exception handling 415
general tips 455
Java applications 231, 426
java.lang.UnsatisfiedLinkError 452
mouse clicks fail on JFC and Visual Café objects 436
not all objects are visible in application 452
objects 435

other problems 436
playback is slow with applications launched from browser 404
projects 71, 449
recognition 452
Silk Test Classic does not recognize a popup dialog box caused by an AWT applet in a browser 454
Silk Test Classic does not record Click() actions against custom controls in Java applets 454
SilkBean 284
SilkTest does not launch Java Web Start application 400
starting Java applications through command line 231, 426
testing on multiple machines 428
unable to delete file 407
verifying $Name property during playback does not work 414
Web applications 461
window not found 410
writing an error-handling function 421
troubleshooting Java applications
no controls found during testing 231, 427
troubleshooting Unicode content
characters not displayed properly 352
compile errors 353
dialog boxes cannot display multiple languages 352
fonts look different 352
IME looks different 353
only English when clicking Language bar icon 353
only pipes are recorded 351
only pipes can be entered in files 352
pipes and squares 351
pipes and squares are displayed in Win32 AUT 352
pipes and squares in the Project tab 352
Save as dialog box when saving existing files 353
Unicode characters do not display 352
VB/ActiveX applications 353
TrueLog
limitations 371
prerequisites 371
replacement characters for non-ASCII 371
setting options 372
wrong non-ASCII characters 371
TrueLog Explorer
about 371
modifying your script to resolve Window Not Found Exception 374
overview 370
setting options 372
toggling at runtime using a script 373
viewing results 373
TrueLog Options dialog box
modifying your script to resolve exceptions 374
opening 372
turning off
multiple tag recording 299
type
statements 330
typographical errors 397

U

unable to connect to agent 407
unable to delete file
troubleshooting 407
unable to start Internet Explorer 407
unicode content
configuring Microsoft Windows XP PC 348
using DB Tester 345
Unicode content
installing language support 349
setting up IME 349
support 344
troubleshooting 351
troubleshooting display issues 351
troubleshooting file format issues 353
troubleshooting IME issues 353
uninitialized variables 397
unique data
specifying 108
UNIX machines
configuring SilkBean support 283
updates 19
user defined attributes
test plans 116
user options
Web applications 262
User options for table recognition 266
user-defined methods
documentation examples 361
username
specifying for test frame
password
specifying for test frame 262
using basic workflow
enabling extensions 43, 76
using file functions
adding information to the beginning of a file 449

V

values
assigning to test plans 118
finding and replacing 152
test plans 116
variable
definition 498
variable browser not defined 408
variables
changing values 395
using 394
viewing 394
VBOptionButton
access to control methods 198
VBOptionButton control methods)
access 198
verification properties
defining 316
verification statement 498
verifications
defining properties 316
fuzzy 142

overview 138
Verify function
verifying objects 139
verify properties does not capture window properties 455
verifying
control no longer displayed 144
object attributes 140
object properties 138
window no longer displayed 144
verifying $Name property
does not work during playback 414
verifying appearance
bitmaps 141
verifying bitmaps
overview 141
verifying object attributes
steps 140
verifying objects
using Verify function 139
verifying properties
as sets 156
verifying state
objects 141
VerifyProperties()
BrowserPage properties and children detected during recording 463
view trace listing
enabling 395
viewing
test results 46, 368
viewing an individual summary 387
viewing class methods
Library Browser 360
viewing files
associated with projects 68
viewing resources
included within projects 68
viewing results
TrueLog Explorer 373
viewing statistics
comparing baseline and result bitmaps 379
virus detectors
conflicts 441
Visual 4Test
definition 498
Visual Basic
native objects displayed as custom windows 399
no methods found 399
no properties found 399
predefined class definition file 196
properties not displayed 398
testing 195
troubleshooting 398
troubleshooting application configuration 400
Visual Basic applications
standard names 80
Visual Basic support
overview 195
Visual Café
sample command line 250
Visual Café objects
mouse clicks fail 436
VO automation
changing to DOM extension 267
information for current customers 267
workaround 267
VO extension
information for current customers 267
VO extensions
comparison to DOM 268

W

Web application classes
4Test language (Classic Agent) 279
web applications
images 272
Web applications
application not ready to test 461
cannot find Web page 464
cannot recognize text 465
cannot recognize Web object 465
cannot verify browser extension settings 464
characters not displayed properly 352
children in browser page not recognized 463
Classic Agent 257
columns 271
controls 272
empty page 461
error with IE and Accessibility 461
handling login windows 92
links 273
many MoveMouse() calls 467
no HTML elements 461
recording test frames 259
recording window declarations 259
recovery system 89, 227
sample applications 257
setting DOM extension options 262
setup steps for testing with the Classic Agent 259
supported controls 257
tables 271
test frame containing HTML frame declarations does not compile 466
test frame files 261
test frames 260
testing text 273
troubleshooting 461
user options 262
Web classes
not displayed in Library Browser 361
Web pages
testing objects 268
Web property sets not displayed during verification 466
Web Start
not launching 400
Web testing
setting agent options 83
WebSync 19
what happens when you enable ActiveX/Visual Basic?
ActiveX/Visual Basic 196, 198, 201–204, 398
Win32
pipes and squares are displayed in AUT 352
priorLabel 282
window browser does not define a tag 408

window declarations
improving 295
overview 295
recording for Java 245
recording for main window 296
recording only pipes 351
recording without 129
testing multiple applications 183
XML 278
window identifiers
recording 138
window is not active 408
window is not enabled 409
window is not exposed 409
window not found
troubleshooting 410
window not found exceptions
preventing 41
setting in agent options 41
setting manually 41
window part 499
window properties
not captured by verify properties 455
window timeout
setting 41
setting in agent options 41
setting manually 41
Window Timeout
setting 199
windows
declarations 295
defining 297
verifying that no longer displayed 144
Windows API-based applications
attributes 280
overview 279
testing 279
Windows Forms
overview 209
Windows Forms applications
Classic Agent 209
Windows Forms, Classic Agent
enabling .NET support 208
no-touch application prerequisites 210
no-touch application support 210
recording new classes 210
Windows XP
unicode content 348
wMainWindow
DefaultBaseState 88
workflow
data-driven 145
workflow bars
disabling 149
enabling 149
works order number 19
wStartup
handling login windows (Classic Agent) 93

X

XML
window declarations 278
XML objects
identifiers 278
tags 278
XML recognition
setting options 278
XPath
definition 499

Z

Zoom window
capturing in scan mode 376
zooming windows
Bitmap Tool 379