SilkPerformer 2011
Java Framework Developer Guide

Borland®
(A MICRO FOCUS COMPANY)

MICRO FOCUS®

Contents

Tools and Samples

Explains the tools, sample applications and test projects that SilkPerformer provides for testing Java and .NET.

Introduction to SilkPerformer SOA Edition

This introduction serves as a high-level overview of the different test approaches and tools, including Java Explorer, Java Framework, .NET Explorer, and .NET Framework, that are offered by SilkPerformer Service Oriented Architecture (SOA) Edition.

SilkPerformer SOA Edition Licensing

Each SilkPerformer installation offers the functionality required to test .NET and Java components. Access to Java and .NET component testing functionality is however only enabled through SilkPerformer licensing options. A SilkPerformer SOA Edition license is required to enable access to component testing functionality. Users may or may not additionally have a full SilkPerformer license.

What You Can Test With SilkPerformer SOA Edition

With SilkPerformer SOA Edition you can thoroughly test various remote component models, including:

- Web Services
- .NET Remoting Objects
- Enterprise JavaBeans (EJB)
- Java RMI Objects
- General GUI-less Java and .NET components

Unlike standard unit testing tools, which can only evaluate the functionality of a remote component when a single user accesses it, SilkPerformer SOA Edition can test components under concurrent access by up to five virtual users, thereby emulating realistic server conditions. With a full SilkPerformer license, the number of virtual users can be scaled even higher. In addition to testing the functionality of remote components, SilkPerformer SOA Edition also verifies the performance and interoperability of components.

SilkPerformer SOA Edition assists you in automating your remote components by:

- Facilitating the development of test drivers for your remote components
- Supporting the automated execution of test drivers under various conditions, including functional test scenarios and concurrency test scenarios
- Delivering quality and performance measures for tested components

SilkPerformer offers the following approaches to creating test clients for remote components:

- Visually, without programming, through Java Explorer and .NET Explorer
- Using an IDE (Microsoft Visual Studio)
- Writing Java code
- Recording an existing client
- Importing JUnit or NUnit testing frameworks
- Importing Java classes
- Importing .NET classes

Provided Tools

Offers an overview of each of the tools provided with SilkPerformer for testing Java and .NET.

SilkPerformer .NET Explorer

SilkPerformer .NET Explorer, which was developed using .NET, enables you to test Web Services, .NET Remoting objects, and other GUI-less .NET objects. .NET Explorer allows you to define and execute complete test scenarios with different test cases without requiring manual programming; everything is done visually through point and click operations. Test scripts are visual and easy to understand, even for staff members who are not familiar with .NET programming languages.

Test scenarios created with .NET Explorer can be exported to SilkPerformer for immediate reuse in concurrency and load testing, and to Microsoft Visual Studio for further customization.

SilkPerformer Visual Studio .NET Add-On

The SilkPerformer Visual Studio .NET Add-On allows you to implement test drivers in Microsoft Visual Studio that are compatible with SilkPerformer. Such test drivers can be augmented with SilkPerformer features that facilitate test organization, verification, performance measurement, test data generation, and reporting.

Tests created with the Add-On can be run either within Microsoft Visual Studio, with full access to SilkPerformer's functionality, or within SilkPerformer, for concurrency and load testing scenarios.

The Add-On offers the following features:

- Writing test code in any of the main .NET languages (C#, VB.NET, or Managed C++).
- Testing Web services / .NET Remoting objects and redirecting HTTP traffic over the SilkPerformer Web engine to take advantage of features such as modem simulation and IP-address multiplexing. SOAP envelopes can also be explored using TrueLog Explorer.
- Defining virtual users and their transactions through .NET custom attributes. A BDL script is generated automatically based on the custom attributes that have been applied to classes/methods.
- Running TryScript tests from within Microsoft Visual Studio with the ability to watch virtual user output in a tool window within Microsoft Visual Studio.
- Exploring the results of Try Scripts.

.NET Resources

- *http://msdn.microsoft.com/net*
- *http://www.gotdotnet.com*

SilkPerformerJava Explorer

SilkPerformerJava Explorer, which was developed using Java, enables you to test Web Services, Enterprise JavaBeans (EJB), RMI objects, and other GUI-less Java objects. Java Explorer allows you to define and execute complete test scenarios with multiple test cases without requiring manual programming. Everything can be done visually via point and click operations. Test scripts are visual and easy to understand, even for personnel who are not familiar with Java programming.

Test scenarios created with Java Explorer can be exported to SilkPerformer for immediate reuse in concurrency and load testing.

Note: Java Explorer is only compatible with JDK versions 1.2 and later (v1.4 or later recommended).

Java Resources

- *http://java.sun.com*
- *http://www.javaworld.com*

SilkPerformer Workbench

Remote component tests that are developed and executed using Java Explorer or .NET Explorer can be executed within SilkPerformer Workbench. SilkPerformer is an integrated test environment that serves as a central console for creating, executing, controlling and analyzing complex testing scenarios. Java Explorer and .NET Explorer visual test scripts can be exported to SilkPerformer by creating SilkPerformer Java Framework and .NET Framework projects. While Java Explorer and .NET Explorer serve as test-beds for functional test scenarios, SilkPerformer can be used to run the same test scripts in more complex scenarios for concurrency and load testing.

In the same way that SilkPerformer is integrated with Java Explorer and .NET Explorer, SilkPerformer is also integrated with SilkPerformer's Visual Studio .NET Add-On. Test clients created in Microsoft Visual Studio using SilkPerformer's Visual Studio .NET Add-On functionality can easily be exported to SilkPerformer for concurrency and load testing.

> **Note:** Because there is such a variety of Java development tools available, a Java tool plug-in is not feasible. Instead, SilkPerformer offers features that assist Java developers, such as syntax highlighting for Java and the ability to run the Java complier from SilkPerformer Workbench.

In addition to the integration of SilkPerformer with .NET Explorer, Java Explorer, and Microsoft Visual Studio, you can use SilkPerformer to write custom Java and .NET based test clients using SilkPerformer's powerful Java and .NET Framework integrations.

The tight integration of Java and .NET as scripting environments for SilkPerformer test clients allows you to reuse existing unit tests developed with JUnit and NUnit by embedding them into SilkPerformer's framework architecture. To begin, launch SilkPerformer and create a new Java or .NET Framework-based project.

In addition to creating test clients visually and manually, SilkPerformer also allows you to create test clients by recording the interactions of existing clients, or by importing JUnit test frameworks or existing Java/.NET classes. A recorded test client precisely mimics the interactions of a real client.

> **Note:** The recording of test clients is only supported for Web Services clients.

To create a Web Service test client based on the recording of an existing Web Service client, launch SilkPerformer and create a new project of application type `Web Services/XML/SOAP`.

Sample Applications for testing Java and .NET

The sample applications provided with SilkPerformer enable you to experiment with SilkPerformer's component-testing functionality.

Sample applications for the following component models are provided:

- Web Services
- .NET Remoting
- Java RMI

Public Web Services

Several Web Services are hosted on publicly accessible demonstration servers:

- *http://demo.borland.com/BorlandSampleService/BorlandSampleService.asmx*
- *http://demo.borland.com/OrderWebServiceEx/OrderService.asmx*

- *http://demo.borland.com/OrderWebService/OrderService.asmx*
- *http://demo.borland.com/AspNetDataTypes/DataTypes.asmx*

> **Note:** *OrderWebService* provides the same functionality as *OrderWebServiceEx*, however it makes use of SOAP headers in transporting session information, which is not recommended as a starting point for Java Explorer.

.NET Message Sample

The .NET Message Sample provides a .NET sample application that utilizes various .NET technologies:

- Web Services
- ASP.NET applications communicating with Web Services
- WinForms applications communicating with Web Services and directly with .NET Remoting objects.

To access the .NET Message Sample:

If you have SilkPerformer SOA Edition: Go to **Start** > **Programs** > **Silk** > **SilkPerformer SOA Edition 2011** > **Sample Applications** > **.NET Framework Samples** .

If you have SilkPerformer Enterprise Edition: Go to **Start** > **Programs** > **Silk** > **SilkPerformer 2011** > **Sample Applications** > **.NET Framework Samples** .

.NET Explorer Remoting Sample

The .NET Remoting sample application can be used in .NET Explorer for the testing of .NET Remoting.

To access the .NET Explorer Remoting Sample:

If you have SilkPerformer SOA Edition: Go to **Start** > **Programs** > **Silk** > **SilkPerformer SOA Edition 2011** > **Sample Applications** > **.NET Explorer Samples** > **.NET Explorer Remoting Sample** .

If you have SilkPerformer Enterprise Edition: Go to **Start** > **Programs** > **Silk** > **SilkPerformer 2011** > **Sample Applications** > **.NET Explorer Samples** > **.NET Explorer Remoting Sample** .

DLL reference for .NET Explorer: `<SilkPerformer installpath>\.NET Explorer\SampleApps \RemotingSamples\RemotingLib\bin\debug\RemotingLib.dll`.

Java RMI Samples

Four Java RMI sample applications are included:

- A simple RMI sample application that is used in conjunction with the sample Java Framework project (`...Samples/Java Framework/RMI`).

  To start the ServiceHello RMI Server, go to: **Start** > **Programs** > **Silk** > **SilkPerformer 2011** > **Sample Applications** > **Java Samples** > **RMI Sample - SayHello**.
- Two simple RMI sample applications (available at `<SilkPerformer installpath>\Java Explorer\SampleApps\`)
- A more complex RMI sample that uses RMI over IIOP is also available. For details on setting up this sample, go to: **Start** > **Programs** > **Silk** > **SilkPerformer 2011** > **Sample Applications** > **Java Samples** > **Product Manager**. This sample can be used with the sample test project that is available at `...Samples/Java Framework/RMI/IIOP` .

Java RMI can be achieved using two different protocols, both of which are supported by Java Explorer:

- Java Remote Method Protocol (JRMP)
- RMI over IIOP

Java Remote Method Protocol (JRMP)

A simple example server can be found at: `<SilkPerformer installpath>\Java Explorer\SampleApps`

Launch the batch file `LaunchRemoteServer.cmd` to start the sample server. Then use the Java Explorer **Start Here Wizard** to begin testing RMI objects. Select **RMI** and click **Next**.

The next dialog asks for the RMI registry settings and a classpath where the RMI interfaces for the client can be found. Here are the settings to be used for this example:

Host: `localhost`

Port: `1099`

Client Stub Class: `<SilkPerformer installpath>\Java Explorer\SampleApps\lib\sampleRmi.jar`

RMI over IIOP

A simple example server can be found at: `<SilkPerformer installpath>\Java Explorer\SampleApps`.

Launch the batch file `LaunchRemoteServerRmiOverIiop.cmd` to start the sample server.

Use the Java Explorer **Start Here Wizard** to begin testing RMI objects. Select `Enterprise JavaBeans/RMI over IIOP` and click **Next**.

The next step asks for the JNDI settings and a classpath where the RMI interfaces for the client can be found. Here are the settings to be provided for this example:

Server: `Sun J2EE Server`

Factory: `com.sun.jndi.cosnaming.CNCtxFactory`

Provider `URL: iiop://localhost:1050`

Stub Class: Click **Browse** and add the following jar file: `<SilkPerformer installpath>\Java Explorer\SampleApps\lib\sampleRmiOverIiop.jar`.

Sample Test Projects

The following sample projects are included with SilkPerformer. To open a sample test project, open SilkPerformer and create a new project. The **Outline Project workflow** dialog opens. Select the application type **Samples**.

.NET Sample Projects

.NET Remoting

This sample project implements a simple .NET Remoting client using the SilkPerformer .NET Framework. The .NET Remoting test client, written in C#, comes with a complete sample .NET Remoting server.

Web Services

This sample shows you how to test SOAP Web Services with the SilkPerformer .NET Framework. The sample project implements a simple Web Services client. The Web Services test client, written in C#, accesses the publicly available demo Web Service at: *http://demo.borland.com/BorlandSampleService/ BorlandSampleService.asmx*

Java Sample Projects

JDBC

This sample project implements a simple JDBC client using the SilkPerformer Java Framework. The JDBC test client connects to the Oracle demo user *scott* using Oracle's "thin" JDBC driver. You must configure connection settings in the `databaseUser.bdf` BDL script to run the script in your environment. The sample accesses the EMP Oracle demo table.

RMI/IIOP

This sample project implements a Java RMI client using the SilkPerformer Java Framework. The test client uses IIOP as the transport protocol and connects to a RMI server provided as a sample application. For detailed instructions on setting up this sample project, see `<SilkPerformer installpath>\SampleApps\RMILdap\Readme.html`.

The Java RMI server can be found at: `<SilkPerformer installpath>\SampleApps\RMILdap`.

RMI

This sample project implements a Java RMI client using the SilkPerformer Java Framework. The test client connects to a RMI server provided as a sample application. For detailed instructions on setting up this sample project, see `<SilkPerformer installpath>\SampleApps\RMILdap\Readme.html`.

To access the Java RMI server:

If you have SilkPerformer SOA Edition: Go to **Start** > **Programs** > **Silk** > **SilkPerformer SOA Edition 2011** > **Sample Applications** > **Java Samples** > **RMI Sample - SayHello** .

If you have SilkPerformer Enterprise Edition: Go to **Start** > **Programs** > **Silk** > **SilkPerformer 2011** > **Sample Applications** > **Java Samples** > **RMI Sample - SayHello**.

Testing Java Components

The Java Framework encourages efficiency and tighter integration between QA and development by enabling developers and QA personnel to coordinate their development and testing efforts while working entirely within their specialized environments.

Testing Java Components Overview

With the Java Framework developers work exclusively in their Java programming environments while QA staff work exclusively in SilkPerformer. There is no need for staff to learn new tools.

Java developers typically build applications and then hand them off to QA for testing. QA personnel are then tasked with testing Java applications end-to-end and component-by-component. Typically QA personnel are not given clients (test drivers) to test applications and they typically are not able to code such test clients themselves. This is where Java Explorer and the Java Framework are effective. Java Explorer offers a means of visually scripting test clients. In effect Java Explorer acts like a test client and can be used to interact with the application under test.

All Java components can be tested with Java Explorer, but the focus lies on the following three Java components: Enterprise JavaBeans (EJBs), Web Services, and Remote Method Invocation (RMI).

The Java Framework enables users to run stand-alone Java test code, to use other tools to invoke Java test code, or to execute test code from an exported standalone console.

Working With JDK Versions

Because multiple Java Developer Kit (JDK) versions are available, you need to test components against all versions. Both SilkPerformer's Java Explorer and Java Framework support testing components of various vendors and of different JDK versions.

Java Framework Approach

The Java Framework approach to component testing is ideal for developers and advanced QA personnel who are not familiar with BDL (Benchmark Description Language), but are comfortable working with a Java development tool. With this approach, Java code is used by the QA department to invoke newly created methods from SilkPerformer.

You can generate Java Framework BDL code using the SilkPerformer JUnit import tool. The import tool produces BDL code that can invoke customer test code or customer JUnit testcode. It can also be used to directly invoke a client API.

Plug-In for Eclipse

SilkPerformer offers a plug-in for Eclipse developers that automatically generates all required BDL code from within the Eclipse SDK. Developers simply write their code in Eclipse and implement certain methods for integrating with the SilkPerformer Java Framework. The plug-in then creates all required BDL scripting that the QA department needs to invoke newly created methods from SilkPerformer. The plug-in for Eclipse enables developers and QA personnel to better coordinate their efforts, consolidating test assets and

enabling both testers and developers to work within the environments with which they are most comfortable.

Java Explorer Overview

Java Explorer is a GUI-driven tool that is well suited for QA personnel who are proficient with SilkPerformer in facilitating analysis of Java components and thereby creating SilkPerformer projects, test case specifications, and scripts from which tests can be run.

Developers who are proficient with Java may also find Java Explorer helpful for quickly generating basic test scripts that can subsequently be brought into a development tool for advanced modification.

Java Explorer emulates Java clients. When working with Web services, Java Explorer achieves this through the use of proxies, which are conversion encoding/decoding modules between the network and client. Proxies communicate with servers by converting function calls into SOAP (XML) traffic, which is transferred through HTTP. Requests are decoded on the remote computer where the component resides. The XML request is then decoded into a real function call, and the function call is executed on the remote server. The results are encoded into XML and sent back to Java Explorer through SOAP, where they are decoded and presented to the user. What you see as the return result of the method call is exactly what the server has sent over the wire through XML.

Java Framework

As a powerful extension to the SilkPerformer Benchmark Description Language, the Java Framework enables you to implement user behavior in Java. When testing an existing Java application you do not need to spend much time creating test scripts. The only effort required is embedding existing Java source code into the framework.

Java Framework Overview

To generate a benchmark executable and run a load test with the help of the Java Framework, two source files are normally required:

- A Java class that implements the behavior of a virtual user
- A SilkPerformer test script that invokes Java method calls

Both source files need to have the same name, with extensions indicating the file types (`.bdf` for the SilkPerformer test script and `.java` for the Java source file).

With SilkPerformer it is possible to call Java functions on any Java object from a BDL context with any number and type of parameter and return parameter. When a Java function returns a complex object, a handle for the Java object is returned in its place. The handle can be passed to other function calls as an input parameter, or public methods may be invoked on the object. There is also support for static methods, cast operations, and exception handling.

Java-based tests are driven by BDL scripts. BDL implements a set of functions that enable you to:

- Set JVM options
- Load a JVM
- Load any Java classes
- Invoke methods on Java classes

The `silk.performer` package provides the `SilkPerformer` class, which enables the Java implementation to communicate with the runtime system. It implements a set of testing methods that are also available in BDL.

What You Can Do With Java Framework

The SilkPerformer Java Framework allows you to do the following:

- *Use the SilkPerformer Java Recorder to record/replay Java-based executables.* The Java recorder generates Java Framework projects (BDL code plus recorded Java Framework scripts). Record/replay scripts can be compared using TrueLog Explorer.
- Reuse existing Java test code for load testing using one of the following methods:

  - By generating Java Framework BDL code using the SilkPerformer JUnit import tool. The import tool produces BDL code that can invoke customer test code or customer JUnit test code. It can also directly invoke a client API.
  - *By calling Java test code from a Java class that is in turn controlled by a Java Framework BDL script.*
  - *By calling Java test code directly from BDL.*
- Create new Java test code:

- Test code can be created visually using Java Explorer. Code can then be exported to SilkPerformer. Java Explorer produces BDL code and Java Framework script.
- Manual coding in a third-party Java IDE. Code can then be used to create a Java Framework project using one of the three approaches listed above.
- Manual coding via SilkPerformer to create Java test code. SilkPerformer offers a Java compiler, syntax highlighting, and other helpful features. Everything except recorded scripts can be scripted manually.

Java Framework Architecture

The following figure illustrates the architecture of the SilkPerformer Java Framework.

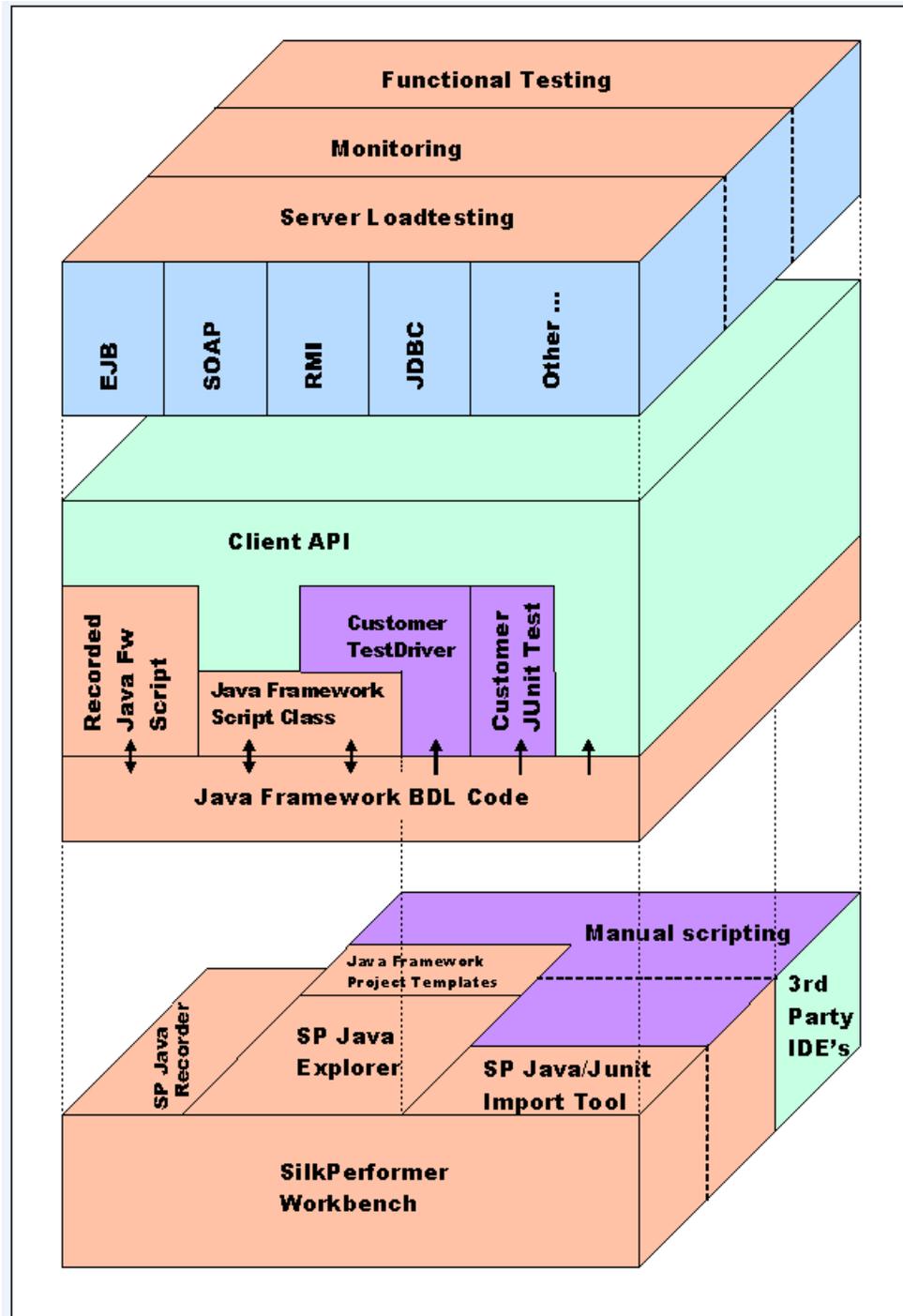The figure is divided into three main sections:

- The upper box illustrates supported technologies and test types.
- The middle box illustrates layers of Java Framework code.
- The lower box illustrates available tools.

Supported Technologies and Test Types (upper box in figure) In one dimension, the upper box shows some of the supported technologies. The second dimension illustrates the areas of testing in which Java Framework can be used.

Layers of Java Framework Code (middle box in figure) The middle box shows the layers of Java Framework code and its connection to various client APIs. The Java Framework API is the "membrane" between BDL code and Java code, symbolized by arrows in the figure. Some of the arrows point in one direction (see "Runtime-to-Java Communication"). Some of the arrows show that communication is possible from Java to the runtime (i.e., Java-to-Runtime communication).

*Available Tools* (lower box in figure below) In one dimension, the lower box shows SilkPerformer tools for generating Java Framework code. The second dimension indicates if code generation is done with the help of SilkPerformer tools or 3rd party tools.

In the figure below, pink represents SilkPerformer technologies; green represents third party tools; purple indicates customer code; and blue represents supported technolgies.

Functional Testing

Monitoring

Server Loadtesting

EJB SOAP RMI JDBC Other ...

Client API

Recorded Java Fw Script

Java Framework Script Class

Customer TestDriver

Customer JUnit Test

Java Framework BDL Code

Manual scripting

Java Framework Project Templates

3rd Party IDE's

SP Java Recorder

SP Java Explorer

SP Java/Junit Import Tool

SilkPerformer Workbench

Anatomy of a Java User

The below diagram illustrates the components that are required to run a virtual Java user. A BDL script is executed by SilkPerformer. Using the framework, Java classes can be plugged into any BDL script. The Java Virtual Machine (JVM) is loaded dynamically into the run-time.
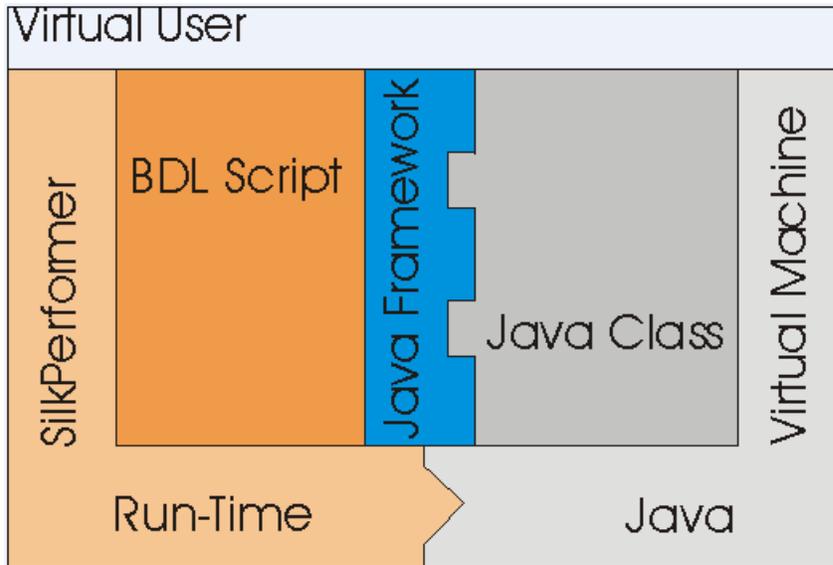
Runtime-to-Java Communication

SilkPerformer provides the following functions for invoking the methods of a Java class from within a test script:

- *JavaSetOption:* Sets options for the Java Virtual Machine.
- *JavaCreateJavaVM:* Initializes and starts the Java Virtual Machine (JVM).
- *JavaLoadObject:* Instantiates a Java object and returns a handle on it.
- *JavaLoadString:* Instantiates a Java string object and returns a handle on it.
- *JavaCallMethod:* Invokes either a dynamic or a static Java method.
- *JavaFreeObject:* Frees a Java object.
- *JavaCastObject:* Casts an object to a type of your choice.
- *JavaGetBoolean:* Retrieves the Boolean return value.
- *JavaGetFloat:* Retrieves the float return value.
- *JavaGetNumber:* Retrieves the int return value.
- *JavaGetObject:* Retrieves a handle on the object returned by the last call of `JavaCallMethod` on the specified object or class.
- *JavaGetString:* Retrieves the string return value
- *JavaGetUserObject:* Returns the handle on the `JavaUser` object loaded by `JavaUserInit` command.
- *JavaSetChar:* Sets a char parameter for the next function or constructor call.
- *JavaGetChar:* Retrieves the ordinal number of the char return value.
- *JavaSetBoolean:* Sets a Boolean parameter for the next function or constructor call.
- *JavaSetFloat:* Sets a float parameter for the next function or constructor call.
- *JavaSetNumber:* Sets a numeric parameter for the next function or constructor call.
- *JavaSetObject:* Sets an object parameter for the next function or constructor call.
- *JavaSetString:* Sets a string parameter for the next function or constructor call.
- *JavaSetByteArray:* Convenience function derived from `JavaSetString` function.
- *JavaSetCharArray:* Convenience function derived from `JavaSetString` function.
- *JavaRegisterException:* Registers a Java exception string under a custom error number.
- *JavaUnregisterErrorNumber:* Unregisters an exception message.
- *JavaGetLastException:* Retrieves the last exception message.

Original Java Framework functions:

- *JavaUserInit:* Calls the `JavaUserInit` method of the Java class that implements the behavior of the virtual user.
- *JavaUserRun:* Calls the `JavaUserRun` method of the Java class that implements the behavior of the virtual user.
- *JavaUserMethod:* Calls an additional method of the Java class that implements the behavior of the virtual user.
- *JavaUserShutdown:* Calls the `JavaUserShutdown` method of the Java class that implements the behavior of the virtual user.
- *JUnitCallMethod:* Invokes a JUnit conform method of a class derived from `junit.framework.TestCase`. Prepares the method execution by invoking the `setUp()` method and finishes the test by invoking the `tearDown()` method.

Java-to-Runtime Communication

SilkPerformer Class

The `SilkPerformer` class is used to access runtime information and to forward user information to the runtime. Here is the functionality offered by the `SilkPerformer` class:

- Measure functions for timers and counters
- Synchronization of users (checkpoint = rendezvous) *(global functions)*
- Extended file functions *(file load functions)*
- Output functions for logging and errors *(Print, Write and Rep functions)*
- Global attribute get and set functions accessible from both Java and BDL *(AttributeSet, AttributeGet functions)*
- Test functions *(Get functions)*
- Random functions *(Rnd functions)*
- Date time format functions *(format functions*

Class Files

The `silk.performer` package contains classes that enable the Java user implementation to communicate with the SilkPerformer runtime environment. The class files are bundled in the `javaUserFramework.zip` archive. See the `classfiles` folder in the installation home directory.

For detailed information on all Java Framework classes and methods, view the Java documentation (HTML).

- `SilkPerformer` class
- FileEx
- MeasureCounter
- MeasureTimer
- PrintStream
- ReflectionUtil

Java Framework Template

A reference implementation of a BDL script called `NewJavaFramework.bdf` is included.

Sample Java Class

A sample Java class belonging to a BDL script called `NewJavaFramework.java` is included.

Sample Applications

Three online samples are available. The sample files and related documentation are available at: `<my documents>\Silk\SilkPerformer2011\Samples\JavaFramework\`.

- *Java Framework sample:* This sample demonstrates the basic usage of the Java Framework API.
- *Java RMI and JDBC:* These samples demonstrate how to use the Java Framework API for RMI and JDBC replay.
- *BankSample:* This sample demonstrates some advanced features of the Java Framework API.

Installation Requirements

A Java Virtual Machine compatible with Sun Microsystem's JDK1.1 or JDK1.2 (Java Platform 2) must be installed on each agent machine. A Sun Java Runtime Engine is available in the `Extras` section of the SilkPerformer installation package. Note however that a full installation of a Java environment is recommended so that created Java test classes can be compiled. (See *http://java.sun.com/products* for details)

Using Sun Microsystem's JVM1.2, check your `PATH` environment settings for `.../jre/bin/classic` or `.../jre/bin/hotspot`.

Programming With Java Framework

Explains how to implement a Java Framework test project using SilkPerformer.

1. Add a new Java Framework script to your project. Within SilkPerformer, choose **File** > **New** > **Java Framework Scripts (.bdf/.java)**.
2. Enter a name for the **Java class**, for example `Testclass`. Then **Save** the `.bdf` and `.java` templates.
3. Open the Java class by double clicking it in the **Project** tree menu. Implement the desired Java test code in one or more member functions of the Java class. For example, `public void doFoo()` throws `SilkPerformerException`. If you prefer, you can open and modify the Java class in an IDE instead. Note that Java Explorer offers powerful scripting functionality.
4. Modify the BDF file and make sure that the methods of the Java class are called with correct parameters. Use `JavaSet<Datatype>` functions to set the function parameters. Use `JavaGet<Datatype>` functions to retrieve return values.
5. Open **Profile** settings and configure a Java Virtual Machine for the test run. If the Java class requires additional Java archives, add them to the classpath. The project directory is added to the classpath automatically and does not have to be configured explicitly.
6. Invoke a Try Script run. The Java and BDL script will then be compiled automatically.

Visual Programming With Java Explorer

Java ExplorerRather than scripting test classes by hand, Java Explorer offers a convenient means of creating projects for the testing of Web Services, Enterprise Java Beans, and remote objects. Any existing Java Explorer project can be exported to SilkPerformer.

Structure of a Java Framework Project

A Java Framework project consists of at least one BDL script. Additionally, a project may contain one or more Java source files, class files (compiled Java files), and/or JAR files (collections of class files).

The structure of the BDL script follows the standard syntax. It usually consists of an `Init` transaction, in which the Java environment is initialized, one or several transactions defining user behavior, and an `end`

transaction in which the Java environment is deleted. The BDL script is used as a starting point for the Java runtime initialization while later in the script. execution of virtual user behavior, or at least part of it, is transferred to the Java environment. This is done by instantiating Java objects within the VM and invoking methods on them.

A Java test class template and corresponding test script can automatically be generated by choosing **File > New > Java Framework Scripts (.bdf/.java)**. A dialog box opens. Enter the name of the Java test class that is to be generated.

BDL scripts for invoking existing Java test classes and existing JUnit test classes can easily be created using the Java / JUnit import tool.

User Group Definitions

The users to be simulated are defined in the `workload` section of the test script. A virtual user is denoted by the transactions to be called, along with their frequency.

Usually, user behavior implementation is divided into at least three separate parts: initialization, user actions, and shutdown. Note the user group definition in the BDL sample below:

```
dcluser
  user
    JavaUser
  transactions
    TInit          : begin;
    TMyJavaTrans  : 1;
    TEnd           : end;
```

Here is a BDL sample that includes multiple transactions:

```
dcluser
  user
    JavaUser
  transactions
    TInit            : begin;
    TMyTransaction1 : 5;
    TMyTransaction2 : 2;
    TMyTransaction3 : 8;
    TEnd             : end;
```

Initialization Transaction

The first transaction executed in a test enables virtual machine initialization. The following function calls must be performed within this transaction:

`JavaCreateJavaVM()` initializes and starts the Java Virtual Machine.

`JavaSetOption()` starts the Java Virtual Machine. Several parameters are required, including home directory, Java version, and classpath. These may be configured in the Java section of the active profile or scripted by the `JavaSetOption` command.

`JavaLoadObject` is used to create a Java object. It is necessary to instantiate the class where the Java test code resides. For example, if the Java test code was implemented in a class called `Test.java`, the following method call would instantiate the class and store a handle on it in a global variable. The handle is used whenever a reference to the test class is required.

```
hTestObj := JavaLoadObject("Test");
```

If the Java test class calls `SilkPerformer` functions, then a reference to the `SilkPerformer` context object must be passed to the testclass. The `SilkPerformer` context object must be instantiated first, then the handle on it can be passed to the constructor of the test class by the `JavaSetObject` command.

```
hPerf := JavaLoadObject("silk/performer/SilkPerformer");
JavaSetObject(JAVA_STATIC_METHOD, hPerf);
```

The corresponding code in the test class must be a constructor that takes the `SilkPerformer` context object as a parameter. It makes sense to store the context object as a member variable so that it can be used by all method calls in the Java test class.

```
private SilkPerformer SilkPerformer = null;
public Test(SilkPerformer perf)
{
  SilkPerformer = perf;
}
```

To avoid memory leaks, references to Java objects in BDL should be freed by the `JavaFreeObject` command. The handle on the `SilkPerformer` context object is only used in the `TInit` transaction, so it can be freed at the end of the transaction. Note that only the reference to the object is freed, the Java object itself remains alive as long as there are Java references to it. The handle on the instantiated test object is needed throughout the test. It is freed in the `TEnd` transaction.

```
JavaFreeObject(hPerf);
```

Implementation of the initial transaction should resemble the following:

```
var
  hTestObj   : number;

dcltrans
  transaction TInit
  var
    hPerf : number;
  begin
    JavaCreateJavaVM();

    hPerf := JavaLoadObject("silk/performer/SilkPerformer");
    JavaSetObject(JAVA_STATIC_METHOD, hPerf);
    hTestObj := JavaLoadObject("Test");
    JavaFreeObject(hPerf);
  end TInit;
```

Java Method Calls

The main transactions allow you to define most of the actions that a virtual user is to perform during a test. These actions are defined in any member method of the Java test class, which is defined in the respective Java script. To call a test method called `doFoo` for example, the main transaction in the test script would contain the following function call:

`JavaCallMethod(hTestObj, "testFoo")` calls the `testFoo` method of the Java test class that is defined in the respective Java source file, for example `Test.java`.

Here is an example `TMyJavaTrans` transaction:

```
dcltrans
  transaction TMyJavaTrans
  begin
    JavaCallMethod(hTestObj, "doFoo");
  end TMyJavaTrans;
```

> **Note:** To create a custom timer measure for a Java or JUnit method call, specify the timer name as the third optional parameter of the respective `JavaCallMethod` or `JUnitCallMethod` command.

Parameterized Java Method Calls

As long as test code is parameterized in the Java test class, it may be sufficient to call simple Java methods that do not take parameters or have return values. Often however it is desirable to customize test code in BDL and execute it in Java. Attribute functions can be used to pass parameters from BDL to Java code, but with SilkPerformer it is also possible to call test methods in a Java test class that expect or return simple or complex parameters.

To call a test method that takes parameters, use the following function calls prior to invoking the test method:

- `JavaSetNumber( in hObject : number, in nParam : number, in sType : string optional );`
- `JavaSetBoolean( in hObject : number, in bParam : boolean );`
- `JavaSetFloat( in hObject: number, in fParam : float, in sType : string optional );`
- `JavaSetObject( in hObject : number, in hParam : number, in sType : string optional );`
- `JavaSetString( in hObject : number, in sParam : string allownull );`
- `JavaSetChar( in hObject : number, in nParam : number );`
- `JavaSetByteArray( in hObject : number, in sParam : string allownull, in nLen : number optional );`
- `JavaSetCharArray( in hObject : number, in sParam : string allownull, in nLen : number optional );`

The first parameter of the functions must be a valid handle on a Java object, usually the handle on the Java test object that was retrieved in the `TInit` transaction.

The data type of the first parameter of the Java test method must match the first usage of the `JavaSet*` function, and so on. `In` parameters are only valid for the following method call on a referenced object.

To call a test method that returns a parameter, use the following function calls after invoking the test method:

- JavaGetBoolean(in hObject: number): boolean;
- JavaGetFloat(in hObject: number): float;
- JavaGetNumber(in hObject: number): number;
- JavaGetObject(in hObject : number): number;
- JavaGetString(in hObject : number, out sBuffer : string, in nBufLen : number optional);
- JavaGetChar(in hObject: number): number;

Here is an example parameterized Java method call:

```
dcltrans
  transaction TMyJavaTrans
  var
    fValue : float;
  begin
    // set the first parameter
    JavaSetString(hTestObj, "1");
    // set the second parameter
    JavaSetNumber(hTestObj, 1);
    // invoke the method
    JavaCallMethod(hTestObj, "doFoo");
    // retrieve the result
    fValue := JavaGetFloat(hTestObj);
    Print("doFoo returned "+String(fValue));
  end TMyJavaTrans;
```

End Transaction

The end transaction, executed at the end of a test, implements user termination. This transaction should free all references on all remaining Java test objects.

`JavaFreeObject(in hObject: number)` takes a handle on a Java object as a parameter and frees all references to it.

Here is an example end transaction:

```
dcltrans
transaction TEnd
  begin
    JavaFreeObject(hTestObj);
  end TEnd;
```

Advanced Usage

Using a Java Framework script template requires that you have only a basic knowledge of the available Java Framework functions. If a Java test driver is already available for reuse, then a template project can be generated in which the Java test class acts as an interface between the original test driver and the SilkPerformer script.

Java test drivers can also be invoked directly from SilkPerformer scripts. Normally, you have to update the BDL script, the Java test class, and the Java test driver. By calling the Java test driver directly, you avoid having to update the Java test class. To call a test driver directly from BDL, additional syntax is often required to call static and overloaded methods or to reuse complex objects.

Static Java Method Calls

Static methods can be invoked just like any other method using the `JavaCallMethod` function. Since static methods are defined on a Java class and not on a Java object, the constant `JAVA_STATIC_METHOD` must be used in place of a handle to a Java object, for example `hTestObj`. The second parameter defining the method name now begins with the fully qualified class name, where the method is defined, then a '.' symbol, and then the method name. A fully qualified class name means that the package that contains the class must also be specified. Use the '/' symbol as a separator between subpackages.

`JavaCallMethod(JAVA_STATIC_METHOD, "test/mypackage/Test.doFoo")` calls the static `doFoo` method of the `Test` class in the `mypackage` subpackage in the package test.

Like member methods, static methods may also expect and return parameters. To define an input parameter for a static method, use the following function calls prior to invoking the test method:

- `JavaSetNumber(JAVA_STATIC_METHOD, in nParam : number, in sType : string optional );`
- `JavaSetBoolean(JAVA_STATIC_METHOD, in bParam : boolean );`
- `JavaSetFloat(JAVA_STATIC_METHOD, in fParam : float, in sType : string optional );`
- `JavaSetObject(JAVA_STATIC_METHOD, in hParam : number, in sType : string optional );`
- `JavaSetString(JAVA_STATIC_METHOD, in sParam : string allownull );`
- `JavaSetChar(JAVA_STATIC_METHOD, in hObject : number, in nParam : number );`
- `JavaSetByteArray(JAVA_STATIC_METHOD, in hObject : number, in sParam : string allownull, in nLen : number optional );`
- `JavaSetCharArray(JAVA_STATIC_METHOD, in hObject : number, in sParam : string allownull, in nLen : number optional );`

The datatype of the first parameter of the static Java test method must match the first usage of the `JavaSetParameter` function, and so on. `In` parameters are only valid for the following static method call.

To call a static Java method that returns a parameter, use the following function calls after invoking the test method:

- `JavaGetBoolean(JAVA_STATIC_METHOD): boolean;`
- `JavaGetFloat(JAVA_STATIC_METHOD): float;`
- `JavaGetNumber(JAVA_STATIC_METHOD): number;`
- `JavaGetObject(JAVA_STATIC_METHOD): number;`
- `JavaGetString(JAVA_STATIC_METHOD, out sBuffer : string, in nBufLen : number optional );`
- `JavaGetChar(JAVA_STATIC_METHOD, in hObject: number ): number;`

Here is an example of a parameterized static Java method call:

```
dcltrans
  transaction TMyStaticJavaTrans
  var
    fValue : float;
  begin
    ThinkTime(0.2);
    // set the first parameter
    JavaSetString(JAVA_STATIC_METHOD, "1");
    // set the second parameter
    JavaSetNumber(JAVA_STATIC_METHOD, 1, JAVA_BYTE);
    // invoke the method
    JavaCallMethod(JAVA_STATIC_METHOD, "test/mypackage/Test.doFoo");
    // retrieve the result
    fValue := JavaGetFloat(JAVA_STATIC_METHOD);
    Print("doFoo returned "+String(fValue));
  end TMyStaticJavaTrans;
```

Additional Samples

```
<public user documents>\Silk\SilkPerformer2011\Samples\JavaFramework
\JavaFrameworkSample.bdf
```

```
<public user documents>\Silk\SilkPerformer2011\Samples\JavaFramework
\JavaFrameworkSample.java
```

Distinguishing Java Methods

Input parameters specified with `JavaSet*` functions must exactly match the formal data type of the Java function. Some BDL functions, such as `JavaSetNumber`, can be used for more than one Java data type. The third optional parameter of the BDL function allows you to distinguish between the various Java data types (in the following example, `byte`, `short`, `int`, and `long_`.

Java code example

```
public void doFoo(byte b) {}
public void doFoo(int i) {}
```

BDL code example

```
JavaSetNumber(hTestObj, 127, JAVA_BYTE);
JavaCallMethod(hTestObj, "doFoo");
```

If the actual parameters specified by `JavaSet*` functions and the formal parameters of the Java method do not match, a `java.lang.NoSuchMethodError` exception will be thrown.

The following sample shows such a `NoSuchMethodError`.

```
Native: 1007 - Java Exception, Native Error 3:
java.lang.NoSuchMethodError: doFoo, Test, doFoo((F))
```

The log output shows that no method named `doFoo` defined in a class `Test` could be found with the method signature `doFoo((F))`. As abbreviations for the method parameters, standard Java element type encoding is used.

Additional Samples

`<public user documents>\Application Data\Silk\SilkPerformer2011\Samples \JavaFramework\JavaFrameworkSample.bdf`

`<public user documents>\Application Data\Silk\SilkPerformer2011\Samples \JavaFramework\JavaFrameworkSample.java`

Instantiating Java Objects

The `JavaLoadObject` function is used to instantiate Java objects, and additionally obtain handles on Java objects. Such handles can later be used to call methods on objects or they can be used as input parameters for other method calls. For the `JavaLoadObject` function, you must provide fully qualified names of classes to instantiate them as parameters. The '/' character must be used as a package separator.

For example, to instantiate a Java object of class `java.util.ArrayList` in a `TMain` transaction, the BDL code looks like this:

```
transaction TMain
var hList : number;
begin
  // invoking constructor of java.util.ArrayList
  hList := JavaLoadObject("java/util/ArrayList");
  // use the object
  // ...
  // free reference on the object when done
  JavaFreeObject(hList);
end TMain;
```

If a Java class has a constructor that takes parameters, use `JavaSet*` functions to set those parameters. In the Java Framework, constructor calls are treated like static method calls, so use the constant `JAVA_STATIC_METHOD` instead of an object handle as the first parameter for `JavaSet*` functions.

For example, to invoke the constructor `ArrayList(int initialCapazity)` of class `java.util.ArrayList` in a `TMain` transaction, the BDL code looks like this:

```
transaction TMain
var hList : number;
begin
  // putting necessary parameters on the stack
  JavaSetNumber(JAVA_STATIC_METHOD, 100);
  // invoking constructor of java.util.ArrayList
  hList := JavaLoadObject("java/util/ArrayList");
  // use the object
  // ...
  // free reference on the object
  JavaFreeObject(hList);
end TMain;
```

The `JavaLoadString` function was introduced for convenience; it simplyfies the instantiation of `java.lang.String` objects.

To create a custom timer measure for a Java object instantiation, specify the timer name as the third optional parameter of the respective `JavaLoadObject` or `JavaLoadString` command.

Java Objects as Return Parameters

Java method calls can return complex Java objects. The `JavaGetObject` function can be used to retrieve a handle on such objects. Java objects can also be used as input parameters for other Java method calls. The `JavaSetObject` function is used for this purpose.

Input parameters specified with the `JavaSetObject` function must exactly match the formal parameters of the Java method call. If the formal and actual data type do not match exactly, but their assignments are compatible, then a kind of cast operation must be performed. For example, the formal data type is `java.lang.Object`. The actual datatype is `java.lang.String`. Strings are assignment compatible to objects.

One time cast operation: The third optional parameter of the `JavaSetObject` function allows you to specify the data type that is to be used for the next Java method call.

```
transaction TMyJavaTrans
var
  hValue :number;
  hVector :number;
begin
  hValue := JavaLoadString("myValue");
  hVector := JavaLoadObject("java/util/Vector");
  JavaSetObject(hVector, hValue, "java/lang/Object");
  JavaCallMethod(hVector, "add");
end TMyJavaTrans;
```

A `java.util.Vector` object expects a `java.lang.Object` object as parameter for its add method. So, as shown above, the Java string `hValue` must be treated as a `java.lang.Object` object for the call of the add method. Therefore the third optional parameter of `JavaSetObject` is set to `java/lang/Object`.

Permanent cast operation: If a Java object that is referenced in BDL code needs to have a specific data type for several Java method calls, then the `JavaCastObject` function can be used to permanently change the data type of the object.

```
transaction TMyJavaTrans
var
  hValue :number;
  hVector1 :number;
  hVector2 :number;
begin
  hVector1 := JavaLoadObject("java/util/Vector");
  hVector2 := JavaLoadObject("java/util/Vector");
  hValue := JavaLoadString("myValue");
  JavaCastObject(hValue, "java/lang/Object");

  JavaSetObject(hVector1, hValue);
  JavaSetObject(hVector2, hValue);
  JavaCallMethod(hVector1, "add");
  JavaCallMethod(hVector2, "add");
end TMyJavaTrans;
```

The actual datatype of a referenced Java object in BDL is slightly different from what is commonly considered to be the actual datatype of an object in the Java world. If an object is retrieved from a collection class, then the actual data type of the referenced object equals the formal data type of the method used to access the object.

```
transaction TMyJavaTrans
var
  hObj :number;
  hVector :number;
begin
```

```
  hVector := JavaLoadObject("java/util/Vector");
    … /* some methods filling the Vector */
  JavaSetNumber(hVector, 0);
  JavaCallMethod(hVector, "elementAt");
  hObj := JavaGetObject(hVector);
end TMyJavaTrans;
```

So in the BDL context, the actual data type of the `hObj` reference is `java/lang/Object`, because the formal return value of the `elementAt` method is `java/lang/Object`. The real actual data type of the Java object in the Java virtual machine is unknown.

You can pass a null object to a Java method call. The following function call can be used to define a null parameter:

```
JavaSetObject(hTestObj, JAVA_NULL_OBJECT);
```

Exception Handling of Java Method Calls

Each exception that is not handled by the invoked Java method itself is thrown to the BDL script where a native Java error is raised. The raised error contains the name of the exception class and the exception message. However, the raised error does not contain the stack trace of the exception.

The exception stack trace, which belongs to the native Java error, is written to the virtual user log file.

```
  JavaCallMethod()
  {
    handle       = 0x012fdde8
    Meth. name   = "doFakeScenario"
    return value = false
    Exception    = "java.lang.RuntimeException: something unexpected thrown
in my java code
    at Test.anotherSubMethod(Test.java:40)
    at Test.aSubMethod(Test.java:35)
    at Test.doFakeScenario(Test.java:47)
"
  }
```

Developers often rely on output written to the command console when debugging Java applications. When doing a Try Script run, all command console output of the Java application is captured as `RepMessage` with severity informational. These messages can be viewed in the VU output console and the virtual user report file.

Advanced Exception Handling

Each exception that is not caught by the Java test driver is thrown to the BDL script, where an error under facility Native Java (1007) is raised. In most cases the error subcode is `0`. To distinguish different Java exceptions in BDL, error subcodes can be assigned to specific exception messages. Use `JavaRegisterException` to assign an error subcode to a specific Java exception. These error subcodes can be helpful for other BDL functions, such as `ErrorAdd`, which can be used to change the severity of errors.

Java Test Class

Within a SilkPerformer test script, the Java test class enables you to implement the behavior of virtual users in Java. You can use all available Java classes to create user behavior that is as realistic as possible; you can also use a number of SilkPerformer functions that can be accessed through an instance of the `SilkPerformer` class.

Java Test Class Overview

A Java test class template can be generated automatically by choosing **New** > **Java Framework Scripts (.bdf/.java)**. A dialog box opens, requesting the name of the Java test class that is to be generated.

To enable use of SilkPerformer functions, the implementation of the Java test class must contain the following:

- Instance of the `SilkPerformer` class
- Constructor to enable access to the SilkPerformer runtime system

Additionally, there can be any number of member methods containing Java test code. An automatically generated Java test class will contain one member method, for example:

```
public void doFoo()
```

These elements are detailed below:

Required Package

To access several functions that are provided by the SilkPerformer runtime system, you must import the `SilkPerformer` class. This class is provided in the `silk.performer` package. The `javaUserFramework.zip` file, located in the `Classfiles` subfolder of the SilkPerformer `home` directory, is automatically included in the classpath.

Default classpath entries such as `javaUserFramework.zip` are defined in `perfrun.xml` in the SilkPerformer home directory.

At the beginning of your Java script, import the `SilkPerformer` class using the following statement:

```
import silk.performer.SilkPerformer;
```

Class Definition

The Java test class is defined immediately following the import statement. If `Test` is chosen as the name of the Java test class, the class definition in the Java script will appear as follows:

```
public class Test
{
   ...
}
```

Members

As a member of the Java test class, an instance of the `SilkPerformer` class enables access to each of the functions provided by the SilkPerformer runtime system. The member is initialized when the constructor of the Java test class is called.

```
private SilkPerformer SilkPerformer = null;
```

Constructor

The constructor of the Java test class is required to assign a `SilkPerformer` context object, which is an instance of the `SilkPerformer` class, to the previously defined member called `SilkPerformer`. The `SilkPerformer` context object is passed to the constructor through the constructor's parameter. The constructor definition should resemble the following:

```
public Test(SilkPerformer perf)
{
   SilkPerformer = perf;
}
```

Example Test Method

Any number of member or static methods may be called within a SilkPerformer test script. The methods may take any basic or complex parameters and have any return values. The only restriction is that arrays of basic and complex parameters are not supported.

As a member method example, here is the implementation of the `doFoo()` method:

```
public void doFoo() throws SilkPerformerException
{
  if (SilkPerformer != null)
    SilkPerformer.Print("Method \"Foo\" entered.");
  SilkPerformer.MeasureStart("Foo");
  SilkPerformer.MeasureStop("Foo");
}
```

Note: Any method invoked on the `SilkPerformer` context object can throw a `SilkPerformer` exception, for example when the execution of the test is stopped. `SilkPerformerException` must not be caught by the test code. Exceptions must be thrown on the method caller, which is the SilkPerformer runtime.

Java Class SilkPerformer

The `SilkPerformer` class represents the general class for communication with the test controller. It allows you to access the SilkPerformer runtime system from the test class implementation. A `SilkPerformer` object encapsulates methods and information dedicated to a virtual user implemented as a Java class.

JVM Option Settings

The `JavaSetOption`(<option id>, <option string>) BDL function can be used to determine options such as Java version, the Java home directory, and the class path. This function and the allowed option identifiers are defined in the `Java.bdh` file.

It is recommended that you set Java options in the profile settings. When Java options are set in both profile settings and BDL the settings made in BDL take precedence.

Note: When using the Eclipse plug-in, the Java settings defined in Eclipse have precedence over all other settings.

Option Identifiers

- `JAVA_VERSION` - (must be defined) Determines the JVM version (1.1.x, 1.2.x, 1.3.x, or 1.4.x).
- `JAVA_HOME` - (recommended) Determines the installation directory of the JVM that should be used for replay. Loading of the JVM is independent of the `PATH` environment, so it is possible to switch between JVMs without changing the system `PATH` environment.
- `JAVA_CLASSPATH` - (recommended) Determines the class path for the JVM. The system class path is appended by default (`-classpath`).
- `JAVA_DLL` - Allows you to specify an individual DLL file path for implementing the JVM that is to be loaded.
- `JAVA_VERBOSE` - Enables the verbose option (`-verbose`) of the JVM. By default, this option is disabled.
- `JAVA_DISABLE_COMPILER` - Disables the JIT (Just In Time compiler) of the JVM. By default the JIT is enabled.

- `JAVA_CMDLINE_OPTIONS` - Enables you to specify any commandline options that are to be passed to the JVM.
- `JAVA_USE_SYSTEM_CLASSPATH` - Determines that the system class path should be appended. By default it is appended.
- `JAVA_DESTROY_JVM_ON_SHUTDOWN` - When not set (the default), the current thread is detached from the JVM upon shutdown, but the JVM is not destroyed. When the option is set, all Java threads are killed and the JVM is destroyed upon shutdown. Note that to do this, JVMDI must be enabled by setting `-Xdebug -Xnoagent -Djava.compiler=NONE` in the commandline options.
- `JAVA_SUPPRESS_EXCEPTIONS` – When not set (the default), Java exceptions raise BDL errors. When set, Java exceptions are suppressed and must be checked manually using the `JavaGetLastException()` command

Examples

- `JavaSetOption`(JAVA_VERSION, JAVA_V14); // JVM v1.4.x

  `JavaSetOption`(JAVA_HOME, "j2sdk1.4.0_01");

  `JavaSetOption`(JAVA_CLASSPATH, "c:/myApplication/classes;c:/myTools/tools.zip");

  `JavaSetOption`(JAVA_DISABLE_COMPILER, YES);

  `JavaSetOption`(JAVA_CMDLINE_OPTIONS, ENABLE_JVMDI);

Java User Implementation *(deprecated)*

This topics lists deprecated technology that is outdated and has been replaced by newer technology.

The Java User methodology simplifies use of the Java Framework. The `Java User` class works as an additional layer between BDL and the Java application under test. It defines a `JavaUserInit()`, a `JavaUserShutdown()` function, and optionally, a `JavaUserRun()` or custom methods, all with Boolean return values.

The `Init` transaction in BDL must call the `JavaUserInit()` function, the `End` transaction must call the `JavaUserShutdown()` function. Any further BDL transactions may either call the `JavaUserRun()` function or one of the custom methods. A SilkPerformer object, for enabling Java-to-runtime Communication, is automatically passed to the `Java User` class.

- `Public JavaUserImplementation(SilkPerformer perf) // Constructor`
- `Public Boolean JavaUserInit() throws Exception`
- `Public Boolean JavaUserRun() throws Exception // optional`
- `Public Boolean anyMethodName throws Exception // optional`
- `Public Boolean JavaUserShutdown() throws Exception`

Testing Web Services with SilkPerformer

This chapter explains the basics of SOAP-based Web Services and details how they can be tested.

SOAP Overview

Simple Object Access Protocol (SOAP) is a lightweight XML-based protocol that is used to exchange information in decentralized, distributed application environments. SOAP messages can be transmitted in any way that applications require, as long as both client and server agree upon the method. However, the current specification describes only a single transport protocol binding: HTTP.

SOAP fits perfectly into the world of Internet applications and improves Internet inter-operability for application services. In essence, SOAP packages method calls into XML strings and delivers them to component instances through HTTP. *[1]*

SOAP is not based on Microsoft technology. It is an open standard drafted by UserLand, Ariba, Commerce One, Compaq, Developmentor, HP, IBM, IONA, Lotus, Microsoft, and SAP. SOAP 1.1 was presented to the W3C in May 2000 as an official Internet standard.

Microsoft is one of SOAP's greatest advocates and has incorporated it as a standard interface in its .NET architecture.

SOAP client requests are encapsulated within HTTP POST (or M-POST) packets. The following example is taken from the Internet-draft specification:

```
POST /StockQuote HTTP/1.1
Host: www.stockquoteserver.com
Content-Type: text/xml;
charset="utf-8"
Content-Length: nnnn
SOAPAction: "Some-URI"

<?xml version="1.0"?>
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
   <SOAP-ENV:Body>
       <m:GetLastTradePrice xmlns:m="Some-URI">
           <symbol>DIS</symbol>
       </m:GetLastTradePrice>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The first four lines in the above example are standard HTTP. `POST` is the HTTP verb that is required for all HTTP messages. The `Content-Type` and `Content-Length` fields are required for all HTTP messages that contain payloads. The `Content-Type text/xml` indicates that the payload is an XML message to the server, or a firewall capable of scanning application headers.

The additional HTTP header (`SOAPAction`) is mandatory for HTTP-based SOAP messages, and can be used to indicate the intent of a SOAP HTTP request. The value is a URI that identifies the intent. The content of `SOAPAction` header fields can be used by servers, for example firewalls, to appropriately filter SOAP request messages in HTTP. The empty string ("") header field value indicates that the intent of the SOAP message is provided by the HTTP Request-URI. No value means that there is no indication as to the intent of the message.

The XML code is also straightforward; the elements `Envelope` and `Body` offer a generic payload packaging mechanism; the element `<GetLastTradePrice>` contains an element called `<symbol>`, which contains a stock ticker symbol. The purpose of this request is to get the last trading price of a specific stock, in this case Disney (DIS).

The program that sends this message needs only to understand how to frame a request in a SOAP-compliant XML message and how to send it through HTTP. In the example below, the program knows how to format a request for a stock price. The HTTP server receiving this message knows that it is a SOAP message because it recognizes the HTTP header `SOAPAction`; it then proceeds to process the message.

SOAP defines two types of messages (calls and responses) to allow clients to request remote procedures and to allow servers to respond to such requests. The previous example is an example of a call; the following response example comes in answer to the call:

```
HTTP/1.1 200 OK
Content-Type: text/xml;
charset="utf-8"
Content-Length: nnnn

<?xml version="1.0"?>
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
   <SOAP-ENV:Body>
       <m:GetLastTradePriceResponse xmlns:m="Some-URI">
           <Price>34.5</Price>
       </m:GetLastTradePriceResponse>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The first three lines are standard HTTP. The first line indicates a response code to the previous `POST` request, the second and third lines indicate content type and length.

XML headers enclose actual SOAP payloads. The XML element `<GetLastTradePriceResponse>` contains a response to the request for a trading price; its child element is `<return>`, which indicates the value to be returned to the request.

Testing SOAP Over HTTP-Based Web Services

SilkPerformer offers three options for testing SOAP over HTTP based services:

- Recording/replaying HTTP traffic
- .NET Explorer in combination with SilkPerformer .Net Framework.
- Java Explorer in combination with SilkPerformer Java Framework

Your environment and prerequisites will determine which of these options is best for your needs, for example the "Java Explorer in combination withSilkPerformer Java Framework" approach is best for Java environments.

Recording/Replaying HTTP Traffic

Recording the SOAP protocol over HTTP is as straightforward as recording any Web application running in a browser. The application that is recorded is the application that executes the SOAP Web Service calls. This can either be a client application or a part of the Web application itself.

Creating a new project

The first step is to create a new SilkPerformer project of the `Web Services/XML/SOAP` application type:

This application type automatically configures its profile settings so that `SOAPAction` HTTP headers are used by SOAP-based applications when calling Web Services for recording.

Creating the application profile

The next step is to create an application profile for the client application that you want to record. Simply create a new application profile that is configured for WinSock recording.

Recording a Script

Begin recording a new script with your created application profile. Interact with your client application; the recorder will record all SOAP requests that are executed over HTTP/HTTPS. When you are finished, close the application and save the recorded script.

Script Customization

For each SOAP request that is recorded you will see a scripted `WebHeaderAdd` and `WebUrlPostBin` API call (see the example below).

```
WebHeaderAdd("SOAPAction", "\"http://tempuri.org/Login\"");
WebUrlPostBin(
  "http://localhost/MessageWebService/MessageService.asmx",
  "<?xml version=\"1.0\" encoding=\"utf-8\"?>"
  "<soap:Envelope xmlns:soap=\"http://schemas.xmlsoap.org/soap/envelope/\"
      "xmlns:xsi=\"http://www.w3.org/2001/XMLSchema-instance\"
      "xmlns:xsd=\"http://www.w3.org/2001/XMLSchema\">"
    "<soap:Body>"
      "<Login xmlns=\"http://tempuri.org/\">"
        "<sUsername>myuser</sUsername>"
        "<sPassword>mypass</sPassword>"
      "</Login>"
    "</soap:Body>"
  "</soap:Envelope>", STRING_COMPLETE, "text/xml; charset=utf-8");
```

You can either customize the input parameter of each Web Service call by manually changing the script or you can use the more convenient method of performing customizations with TrueLog Explorer. To do this, run a Try Script. Then use TrueLog Explorer's XML control to customize the XML nodes that represent the input parameters.

Replaying the Script

Once you have finished script customization, you can replay your script, either in another Try Script run, as part of baseline identification, or in a test.

As the Web Service calls are performed along with Web API functions, you will receive the same measures you receive when testing any Web application, including detailed protocol-specific statistics.

Java Explorer

Java Explorer allows users to create test cases using point and click operations. Java Explorer provides support for the following technologies:

- SOAP Web Services
- RMI
- EJB
- General GUI-less Java objects

Java Explorer can be used to export complete SilkPerformer projects that make use of the SilkPerformer Java Framework. Java Explorer itself can only run previously defined test scenarios in animated mode. With exported projects however, SilkPerformer can perform real tests with multiple virtual users.

> **Note:** Java Explorer currently uses the Axis SOAP stack (*http://www.apache.org*) to generate Web Service client proxy classes.

Java Framework

With the Java Framework, SilkPerformer offers a powerful means of simulating virtual users whose behavior is defined with the Java programming language. Arbitrary Java classes can be instantiated within the framework and methods defined within classes may be invoked.

The behavior of virtual users running in the Java framework can be scripted manually using SilkPerformer or another Java IDE. A more convenient method however is to use Java Explorer to define virtual user behavior through its point and click interface.

References

[1]	Session, Roger
	SOAP. An overview of the Simple Object Access Protocol, March 2000
	http://www.objectwatch.com
[2]	W3C
	Simple Object Access Protocol (SOAP) 1.1, Dec 2000
	http://www.w3.org/TR/SOAP/
[3]	UN/CEFACT, OASIS
	Enabling Electronic Business with ebXML, December 2000
	http://www.ebxml.org/white_papers/whitepaper.htm
[4]	Geyer, Carol
	ebXML Integrates SOAP Into Messaging Services Specification, March 2001
	http://www.ebxml.org/news/pr_20010222.htm
[5]	Open Financial Exchange
	Open Financial Exchange Specification 2.0, April 2000
	http://www.ofx.net/

Index