SilkPerformer® .NET Explorer 2010 R2
Help

Borland®
(A MICRO FOCUS COMPANY)

MICRO FOCUS®
Leading the Evolution™

Contents

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 .NET)
- Writing Java code
- Recording an existing client
- Importing JUnit or NUnit testing frameworks
- Importing Java classes
- Importing .NET classes

Provided Tools

This section offers an overview of each of the tools that is provided with SilkPerformer SOA Edition.

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

To launch SilkPerformer .NET Explorer through SilkPerformer SOA Edition, navigate to **Start** > **Programs** > **Silk** > **SilkPerformer SOA Edition 2010 R2** > **SilkPerformer .NET Explorer**.

To launch SilkPerformer .NET Explorer through SilkPerformer Enterprise Edition, navigate to **Start** > **Programs** > **Silk** > **SilkPerformer 2010 R2** > **Development Tools** > **SilkPerformer .NET Explorer**.

Alternatively, you can launch SilkPerformer and create a new project with the application type `.NET/.NET Explorer` or `Web Services/.NET Explorer`.

To launch SilkPerformer through SilkPerformer SOA Edition, navigate to **Start** > **Programs** > **Silk** > **SilkPerformer SOA Edition 2010 R2** > **SilkPerformer SOA Workbench**.

To launch SilkPerformer through SilkPerformer Enterprise Edition, navigate to **Start** > **Programs** > **Silk** > **SilkPerformer 2010 R2** > **SilkPerformer Workbench**.

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.

To launch the Visual Studio .NET Add-On, go to **Start** > **All Programs** > **Microsoft Visual Studio** > **Microsoft Visual Studio** and create a new SilkPerformer Visual Studio project.

Alternatively, you can launch SilkPerformer and create a new project with the application type **.NET** > **.NET Framework using Visual Studio .NET Add-On**.

To launch SilkPerformer through SilkPerformer SOA Edition, navigate to **Start** > **Programs** > **Silk** > **SilkPerformer SOA Edition 2010 R2** > **SilkPerformer SOA Workbench**.

To launch SilkPerformer through SilkPerformer Enterprise Edition, navigate to **Start** > **Programs** > **Silk** > **SilkPerformer 2010 R2** > **SilkPerformer Workbench**.

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*

SilkPerformer Java Explorer

SilkPerformer Java 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 SilkPerformer 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).

To launch SilkPerformer Java Explorer through SilkPerformer SOA Edition, navigate to **Start** > **Programs** > **Silk** > **SilkPerformer SOA Edition 2010 R2** > **SilkPerformer Java Explorer**

To launch SilkPerformer Java Explorer via SilkPerformer Enterprise Edition, navigate to **Start** > **Programs** > **Silk** > **SilkPerformer 2010 R2** > **Development Tools** > **SilkPerformer Java Explorer**

Alternatively, you can launch SilkPerformer and create a new project with the application type `Java/Java Explorer` or `Web Services/Java Explorer`.

To launch SilkPerformer through SilkPerformer SOA Edition, navigate to **Start** > **Programs** > **Silk** > **SilkPerformer SOA Edition 2010 R2** > **SilkPerformer SOA Workbench**.

To launch SilkPerformer through SilkPerformer Enterprise Edition, navigate to **Start** > **Programs** > **Silk** > **SilkPerformer 2010 R2** > **SilkPerformer Workbench**.

Java Resources

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

SilkPerformer Workbench

Remote component tests that are developed and executed using SilkPerformer Java Explorer or SilkPerformer .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 Add-In for Visual Studio .NET. Test clients created in Microsoft Visual Studio .NET using SilkPerformer's Add-In 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 Visual Studio .NET, 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 SOA Edition 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**.

To launch SilkPerformer through SilkPerformer SOA Edition, navigate to **Start** > **Programs** > **Silk** > **SilkPerformer SOA Edition 2010 R2** > **SilkPerformer SOA Workbench**.

To launch SilkPerformer through SilkPerformer Enterprise Edition, navigate to **Start** > **Programs** > **Silk** > **SilkPerformer 2010 R2** > **SilkPerformer Workbench**.

Sample Applications for SilkPerformer SOA Edition

The sample applications provided with SilkPerformer enable you to experiment with SilkPerformer SOA (Service Oriented Architecture) Edition 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 2010** > **Sample Applications** > **.NET Framework Samples**.

If you have SilkPerformer Enterprise Edition, go to: **Start** > **Programs** > **Silk** > **SilkPerformer 2010** > **Sample Applications** > **.NET Framework Samples**.

.NET Explorer Remoting Sample

The .NET Remoting sample application can be used in SilkPerformer .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 2010** > **Sample Applications** > **.NET Explorer Samples** > **.NET Explorer Remoting**.

If you have SilkPerformer Enterprise Edition, go to: **Start** > **Programs** > **Silk** > **SilkPerformer 2010** > **Sample Applications** > **.NET Explorer Samples** > **.NET Explorer Remoting Sample**.

DLL reference for .NET Explorer: `<BMC TM ART 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 2010/ Sample Applications/Java Samples/RMI Sample – SayHello` or `Start/Programs/Silk/ SilkPerformer SOA Edition 2010/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 2010/Sample Applications/Java Samples/Product Manager` or `Start/Programs/Silk/SilkPerformer SOA Edition 2010/`

`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 2010/Sample Applications/Java Samples/RMI Sample - SayHello`. If you have SilkPerformer Enterprise Edition, go to: `Start/Programs/Silk/SilkPerformer 2010/Sample Applications/ Java Samples/RMI Sample - SayHello`.

SilkPerformer .NET Framework

SilkPerformer's .NET Framework enables developers and QA personnel to coordinate their development and testing efforts while allowing them to work entirely within their specialized environments: Developers work exclusively in Visual Studio while QA staff work exclusively in SilkPerformer—there is no need for staff to learn new tools. SilkPerformer's .NET Framework thereby encourages efficiency and tighter integration between QA and development. The SilkPerformer .NET Framework (.NET Framework) and .NET Add-On enable you to easily access Web services from within .NET. Visual Studio .NET offers wizards that allow you to specify the URLs of Web services. Visual Studio .NET can also create Web-service client proxies to invoke Web-service methods.

Testing .NET Components

SilkPerformer's Visual Studio .NET Add-On provides functionality to developers working in .NET-enabled languages for generating SilkPerformer projects and test scripts entirely from within Visual Studio.0

The .NET Framework Approach

The .NET Framework approach to testing is ideal for developers and advanced QA personnel who are not familiar with coding BDL (SilkPerformer's Benchmark Description Language) scripting language, but are comfortable using Visual Studio to code .NET-enabled languages such as C#, COBOL.NET, C++ .NET, and Visual Basic.NET. With SilkPerformer's Visual Studio .NET Add-On, developers can generate SilkPerformer projects and test scripts entirely from within Visual Studio by simply adding marking attributes to the methods they write in Visual Studio. The Add-On subsequently creates all BDL scripting that is required to enable the QA department to invoke newly created methods from SilkPerformer.

The .NET Explorer Approach

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

Developers who are proficient with Visual Studio .NET may also find .NET Explorer helpful for quickly generating basic test scripts that can subsequently be brought into Visual Studio for advanced modification.

Understanding the .NET Framework Platform

.NET Framework is a powerful programming platform that enables developers to create Windows-based applications. The .NET Framework is comprised of CLR (Common Language Runtime, a language-neutral development environment) and FCL (Framework Class Libraries, an object-oriented functionality library).

Visit the *.NET Framework Developer Center* for full details regarding the .NET Framework.

Working with SilkPerformer .NET Framework

The SilkPerformer .NET Framework allows you to test Web services and .NET components. The framework includes a set of the Benchmark Description Language (BDL) API functions of SilkPerformer and an add-on for Visual Studio .NET.

✎ **Note:** For additional details regarding the available BDL API functions, refer to the *Benchmark Description Language (BDL) Reference*.

The framework allows you to either code your BDL calls to .NET objects manually in SilkPerformer or use generated BDL code from the Visual Studio .NET Add-On. One benefit of the latter approach is that the developer of the .NET test driver doesn't require BDL skills, because BDL script generation is handled "behind the scenes" by the Visual Studio .NET Add-On. BDL Scripts can be launched for testing purposes from within Visual Studio .NET through the Add-On. All user output and generated output files, like TrueLogs, logs, output, and others, can be viewed from within Visual Studio .NET.

The .NET Framework allows you to route all HTTP/HTTPS traffic that is generated by a .NET component over the SilkPerformer Web engine. This feature logs TrueLog nodes for each SOAP or .NET Remoting Web request, that is made by a .NET component.

This architecture provides good separation between test driver code and the test environment. There are also mechanisms for defining interaction between BDL and .NET, so you can design a fully customizable .NET test driver from a generated SilkPerformer BDL script.

SilkPerformer .NET Framework Overview
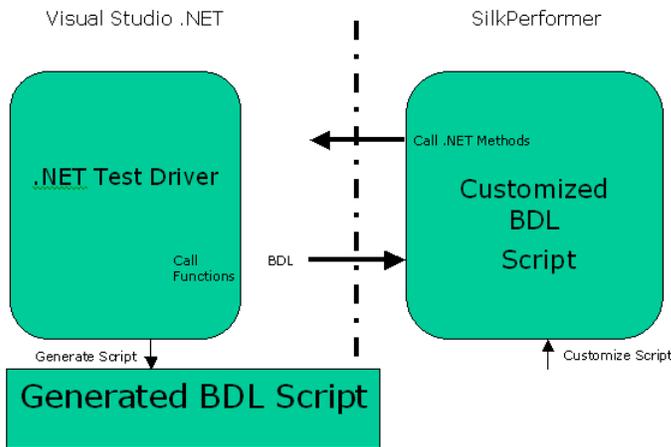
The SilkPerformer .NET Framework integration allows you to instantiate .NET objects and then call methods on them.

The Microsoft .NET *Common Language Runtime (CLR)* is hosted by the SilkPerformer virtual user process when BDF scripts contain DotNet BDL functions.

HTTP/HTTPS traffic that is generated by instantiated .NET objects can be routed over the SilkPerformer Web engine. Each WebRequest/WebResponse is logged in a TrueLog, allowing you to see what is sent over the wire when executing Web service and .NET Remoting calls.

Depending on the active profile setting, which is a .NET application domain setting, either each virtual user has its own .NET application domain where .NET objects are loaded, or alternately all virtual users in the process can share an application domain.

A .NET application domain isolates its running objects from other application domains. An application domain is like a *virtual process* where the objects running in the process are safe from interruption by other processes. The advantage of having one application domain for each virtual user is that the objects that are loaded for each user don't interrupt objects from other users, since they are isolated in their own domains.

The disadvantage is that additional application domains require additional administrative overhead of the CLR. This overhead results in longer object-loading and method-invocation times.

Intermediate Code

.NET code is not compiled into binary "machine" code. .NET code is intermediate code. Intermediate code is descriptive language that delivers instructions, for example "call this method" or "add these numbers", to functions that are available in libraries or within remote components.

.NET code runs within a machine-independent runtime, or "execution engine," which can be run on any platform—Windows, Unix, Linux, or Macintosh. So, regardless of the platform you're running, you can run the same intermediate code. The drawback of this cross-platform compatibility is that, because intermediate code must be integrated with a runtime, it's slower than compiled machine code.

.NET code calls basic Microsoft functionality that is available in .NET class libraries. These are the "base" classes. "Specific" classes, for creating Web applications, Windows applications, and Web Services are also available. In the runtime itself you also have some classes that are offered by Microsoft for building applications—all of this comprises the .NET Framework upon which intermediate code can be written using one of a number of available .NET-enabled programming languages.

It doesn't matter which language is used to create the intermediate code that delivers instructions to the available classes through the .NET runtime—the resulting functionality is the same.

Configuring the .NET Runtime

Multiple configuration files are available for configuring the .NET runtime, for example `machine.config`, `user.config`, and `netexplorer.exe.config`. These files define such attributes as the runtime version that is to be used. SilkPerformer's Visual Studio .NET Add-On offers a means of defining these configuration settings while writing test drivers.

SilkPerformer .NET Explorer supports Microsoft .NET Framework 1.0, 1.1, 2.0. .NET Explorer runs out-of-the-box with .NET Framework 1.1 or later, but if you want to run .NET Explorer with .NET Framework 1.0, it is required to switch to a different application configuration file as follows:

1. Make sure that .NET Explorer is not running.
2. Open the installation directory of .NET Explorer. The default directory is `C:\Program Files\Silk\SilkPerformer 2010 R2\DotNET Explorer`.
3. Locate the file `netexplorer.exe.config` and rename it to `netexplorer_PostFW10.exe.config`.
4. Locate the file `netexplorer_netfw10.exe.config` and rename it to `netexplorer.exe.config`.
5. Start .NET Explorer.

SOAP Web Services

A Web service is an available service on the Web that can be invoked and from which results can be returned. Although other standards exist, the widely accepted standard for Web services, which has been adopted by the W3C, is SOAP (Simple Object Access Protocol). SOAP defines a special type of XML document that is accessible over HTTP. SOAP XML documents are structured around root elements, child elements with values, and other specifications. First an XML document containing a request (a method to be invoked and the parameters) is sent out. The server responds with a corresponding XML document that contains the results.

A SOAP stack, an implementation of the SOAP standard on the client side, is comprised of libraries and classes that offer helper functions. A significant Web service testing challenge is that there are a number of SOAP stack implementations that are not compatible with one another. So although SOAP is intended to be both platform- and technolgy-independent, it is not. Web services written in .NET are however always compatible with .NET clients—they use the same SOAP stack, or library. When testing a .NET Web service however, you need to confirm if the service is compatible with other SOAP stack implementations, for example Java SOAP stack, to avoid interoperability issues.

SilkPerformer Helper Classes

.NET helper classes serve as an interface between SilkPerformer's BDL language and the .NET language. Although SilkPerformer is able to call the .NET Framework through the basic functions that it offers, helper classes are required to enable .NET to call back to SilkPerformer. With helper classes, which are generated automatically with .NET Explorer and the Visual Studio .NET Add-On, .NET developers can take full advantage of developing test code in .NET and don't need to learn BDL. The test code that developers deliver to QA, by making use of helper classes, can be called from SilkPerformer or scheduled in load tests using SilkCentral Test Manager.

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.

To launch the Visual Studio .NET Add-On, go to **Start** > **All Programs** > **Microsoft Visual Studio** > **Microsoft Visual Studio** and create a new SilkPerformer Visual Studio project.

Alternatively, you can launch SilkPerformer and create a new project with the application type **.NET** > **.NET Framework using Visual Studio .NET Add-On**.

To launch SilkPerformer through SilkPerformer SOA Edition, navigate to **Start** > **Programs** > **Silk** > **SilkPerformer SOA Edition 2010 R2** > **SilkPerformer SOA Workbench**.

To launch SilkPerformer through SilkPerformer Enterprise Edition, navigate to **Start** > **Programs** > **Silk** > **SilkPerformer 2010 R2** > **SilkPerformer Workbench**.

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*

Visual Studio .NET Add-On Installation

SilkPerformer's Visual Studio .NET Add-On requires that either of Microsoft Visual Studio 2005, 2008, 2010 is installed. The Visual Studio .NET Add-On is automatically installed with SilkPerformer Setup. The Visual Studio .NET Add-On can be removed and/or reinstalled using the SilkPerformer Add/Remove Program utility.

Starting the SilkPerformer Visual Studio .NET Add-On

Perform one of the following steps to start the SilkPerformer Visual Studio .NET Add-On:

- Click **Start** > **All Programs** > **Microsoft Visual Studio** > **Microsoft Visual Studio** and create a new SilkPerformer Visual Studio project.
- Click **Start** > **All Programs** > **SilkPerformer Workbench** and create a new project with the application type **.NET** > **.NET Framework using Visual Studio .NET Add-On**.

Tour of the UI

Explains the view tabs that are offered by .NET Explorer.

Overview

The main sections of the .NET Explorer interface are highlighted below.
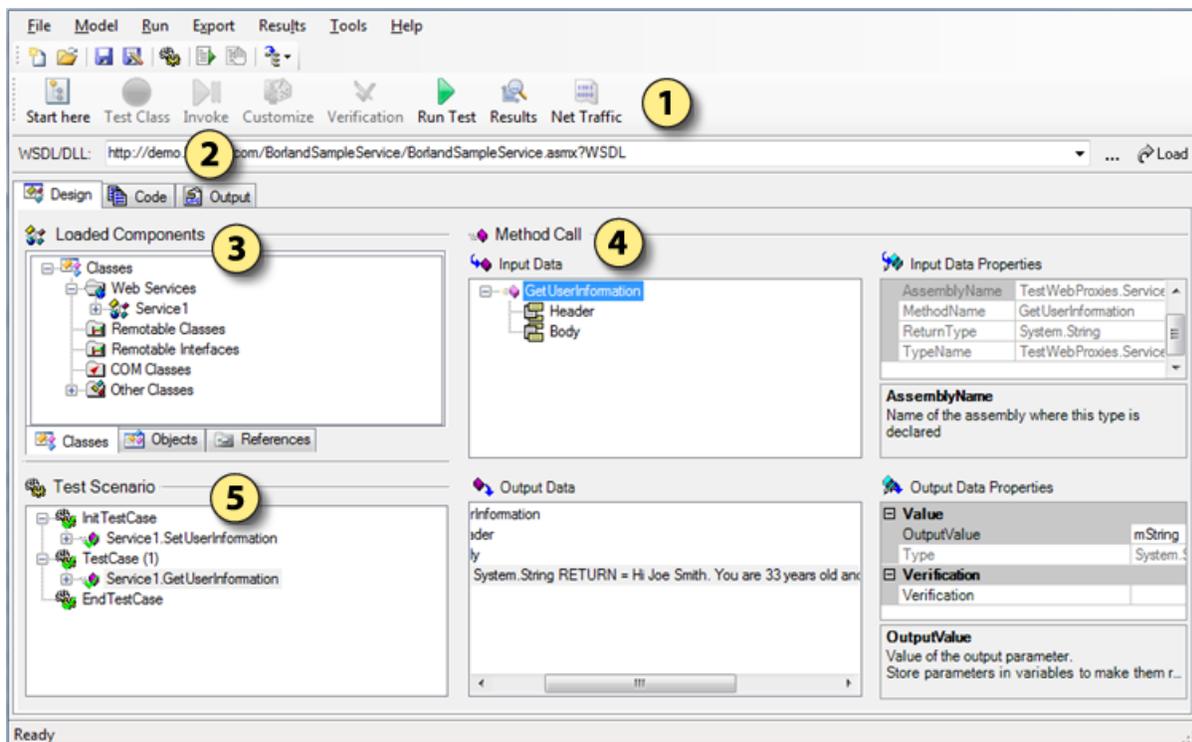
Workflow Bar

The .NET Explorer workflow bar assists you in configuring and running your tests. The buttons on the workflow bar enable you to perform the following commands:

- **Start Here**: Opens the **Load File Wizard**, which guides you through selecting the type of testing that you want to perform and loading the required components.
- **Test Class**: Allows you to create a new test case for a class, or to add methods to an existing test case. This button is enabled when you select a class in the **Loaded Components** menu tree.
- **Invoke**: Invokes the selected method in the **Loaded Components** menu tree and adds it to the selected test case in the **Test Scenario** menu tree.
- **Customize**: Lets you store data as variables. This button is enabled when you select an input parameter in the **Input Data** menu tree, or an output parameter in the **Output Data** menu tree.
- **Verification**: Allows you to define verifications for variables. This button is enabled when you select an output parameter in the **Output Data** menu tree that has been stored as a variable.
- **Run Test**: Performs an animated run of your test cases.
- **Results**: Lets you explore the results of your last test run through user report, TrueLog, error report, or log file analysis.
- **Net Traffic**: Displays the request and response network traffic of the method call that is selected in the **Test Scenario** menu tree.

WSDL/DLL Address Field

The **WSDL/DLL** address field allows for the loading of either WSDLs or .NET assemblies.

Loaded Components Pane

The **Loaded Components** pane includes three tabs:

- **Classes**: A list of loaded Web services, remotable classes/interfaces and other .NET classes. Allows you to instantiate objects, connect to remote objects, or call static methods.
- **Objects**: A list of instantiated objects, global and random variables. Allows you to call methods on those kind of objects.
- **References**: A list of loaded files (assemblies, WSDL's).

Method Call Pane

- The **Input Data** menu tree shows input parameters (both headers and bodies) for the selected method call.
- The **Input Data Properties** field shows the properties of the input parameters (value, type, pass, null, etc).
- The **Output Data** menu tree shows the output parameters (header and body) of the last/selected method call.
- The **Output Data Properties** field shows the properties of the output parameters (value, type, pass, null, etc).

Test Scenario Pane

The **Test Scenario** pane lists the current test cases along with all added method calls.

Design Tab

The **Design** tab is the primary mode used to access the majority of .NET Explorer functionality. The **Design** tab offers the display of loaded components, invocation of methods, customization of input/output data, and the design of test scenarios.

Loaded Components Pane

The **Loaded Components** pane, on the **Design** tab, distinguishes between classes and objects and displays these components within different tree menus.

On the **Classes** tab you see all loaded classes with their members (methods, fields, and properties) that can be accessed without the need of an object context. Normally these are static members and constructors.

When you instantiate a class, the created object is shown on the **Objects** tab with all instance members (methods, fields, and properties).

The two exceptions are Web services and interfaces. As an object instance is not required for Web Services (which are objects that expose static methods), all methods offered by Web services are shown on the **Classes** tab.

Interfaces are displayed in the **Remotable Interfaces** menu node. .NET Remoting allows for `WellKnown` remotable objects, which implement interfaces. Therefore it is not necessary to instantiate objects that work with `WellKnowns` (only one instance exists on the server). This is why .NET Explorer treats remotable interfaces like Web services. All members of interfaces are shown on the **Classes** tab. When you first attempt to invoke a method you must supply the endpoint of the remotable interface (as you must provide endpoints for Web services).

Classes Tab

All loaded classes and Web services are listed in the .NET Explorer **Loaded Components** pane on the **Classes** tab.

Only constructors, static methods, and members are displayed, as there is no object context required to call them.

Web services and remotable interfaces are treated as if they only have static methods.

The following folders are available in **Classes** view:

Web Services

Classes that are derived from `SoapHttpClientProtocol` are included in this folder. Asynchronous method calls are not allowed and therefore are not shown.

Remotable Classes

Classes that are derived from `MarshalByRefObjects` are included in this folder. Activation information (type and URL) can be set at the class node.

To create a remote object, call one of the constructors or use the context menu.

Remotable Interfaces

Interfaces are included in this folder. Interfaces are treated like Web services.

An activation URL to a WellKnown object can be defined for each interface. If an interface is configured for remote access to a WellKnown object, simply call a method.

COM Classes

COM classes are COM wrapper classes that have been imported to be callable from .NET.

Other Classes

Other classes that do not fit in the above listed categories are included in this folder.

Objects Tab

When an object is instantiated through a constructor or returned by a method and stored in a variable, the object is listed in the **Loaded Components** pane on the **Objects** tab. Instance members and methods that can then be invoked by the user are displayed.

The **Objects** tab shows objects stored in variables that have global scope and those that have a local scope of the currently selected test case.

Objects

By default, unless changed in the **Options** dialog box, only objects that differ from the basic types (for example, strings and integers) are listed here as objects. Instance methods and members are displayed to allow for easy invokation.

Members and properties have `get/set` accessors to enable the getting and setting of values.

Local test case objects are included in the `Local Test Case Objects` folder. Global test case objects are included in the `Global Test Case Objects` folder.

Variables

By default variables that store simple datatypes such as strings and integers are listed in the `Global Random Variables` folder, not as objects but as simple variables.

To work with variables of simple data types like objects, select **Show Primitive Types as Objects** on the **Options** dialog box.

References Tab

The **References** tab offers a list of loaded files (for example, assemblies and WSDLs).

Code Tab

Code view displays generated code files for the current test scenario. The following code types are available:

Client Proxy Code

The **Client Proxy** tab displays generated WebClient proxy code. When loading a WSDL file to access a Web Service, a WebClient proxy class is created. Proxy classes are created on method and type information in WSDL files. When Web Service methods return complex types, additional classes are declared. These additional classes are used to represent the complex types in the method calls that take complex parameters. For detailed information about WebClient Proxy generation, consult the MSDN for details regarding `WSDL.EXE`, a tool that creates proxy classes in the same way.

Test Driver Code

The test driver **Test Code** tab displays code generated for the test driver. This code can be compiled to a standalone console application. The main method for the application contains the calls to the test cases that are defined in your current test scenario. The test case methods contain all the calls that you have defined in your scenario for the respective test case.

When you export a standalone console application (via the **Export** menu) and then run the application, you will see output information regarding successful/failed methods in the console window.

SilkPerformer .NET Framework Code

The SilkPerformer .NET Framework Test Code tab displays test code for SilkPerformer .NET Framework. For each test case in your current test scenario you will find a method with the `[Transaction]` attribute applied to it. These transaction methods contain all the calls that you have defined in your test scenario for the respective test case.

This code can either be exported to Visual Studio .NET as a SilkPerformer .NET Project, or it can be exported directly to SilkPerformer. In either case, results will be reported to SilkPerformer and can be viewed in the log files or virtual user output.

Native BDL Script

The **BDL Native** tab displays native BDL script for Web Service calls. When testing Web Services, .NET Explorer can create BDL scripts with `WebPagePost` API calls for each Web Service method call in the current test scenario. `WebPagePost` methods are called with posted HTTP traffic from .NET Explorer during the last execution of the current test scenario.

Exporting to a native BDL script allows you to test a Web Service by sending its HTTP traffic without using the .NET Framework to generate the HTTP traffic.

BDL for .NET Framework

Th **BDL FW** tab displays BDL script for SilkPerformer .NET Framework. When exporting a project as a SilkPerformer .NET Framework project, .NET Explorer uses the code in the SilkPerformer .NET Framework code pane, compiles the code to a .NET Assembly (DLL), and creates a BDL script that loads the DLL and calls the methods that have been marked with the `[Transaction]` attribute. The BDL script is shown here, but it is not shown in the final state that will be generated upon export. Only a skeletal BDL script is displayed.

Output Tab

The **Output** tab displays log output during test runs. The **Output** tab shows details about each call of the most recent animated run. Statistics regarding execution time and executed methods are also shown.

Example Animated Log Output

```
Test started: 08.10.2004 08:01:50
Test stopped: 08.10.2004 08:01:53
Method calls: 1
Success calls: 1
Error calls: 0
TestWebProxies.Service1.echoStringArray
Parameters : System.String[](string)
Return Value: System.String[](string)
```

Note: The **Output** tab can also be accessed by clicking **Results** on the Workflow Bar.

Setting Up .NET Explorer Projects

This section shows you how to set up projects and define tests with .NET Explorer. A sample Web service is used for demonstration purposes.

.NET Explorer requires that Microsoft .NET Framework runtime 2.0 be installed. Microsoft .NET Framework runtime is available on Microsoft's *.NET Framework Downloads* page.

Creating .NET Explorer Projects

Upon opening .NET Explorer you are prompted to either open an existing project or create a new project.

1. Click the **Create a new project** option button.
2. Enter a name for your project in the text box.
3. Click **OK**.

Your new project is created and the SilkPerformer .NET Explorer GUI displays.

Test Case Definition

Three approaches are available for defining tests with .NET Explorer:

- Use the **Load File Wizard** to define where the information for the object you wish to test is located. The wizard then displays the available classes, automatically instantiates the available objects, and calls methods on those objects. This approach is more involved, but it offers more control.
- Enter the location of the Web services' WSDL file directly into .NET Explorer's **WSDL/DLL** field and click **Load**. Information about the methods and objects is then displayed, however objects are not instantiated and methods are not called. This approach offers a quick, simple approach to analyzing remote service functionality.
- Import a COM wrapper using the **Load COM Type Libraries** dialog from either the **Model** menu or from the **COM Classes** context menu in the **Classes** menu tree.

For demonstration purposes this Help uses the **Load File Wizard** to instantiate the objects available on the sample Web service, which is available at *http://demo.borland.com/BorlandSampleService/ BorlandSampleService.asmx?WSDL*.

Using the Load File Wizard

1. Click **Start Here** on the workflow bar.

The **Load File Wizard** dialog box displays.

2. To test a Web service, you must provide a Web Services Description Language (WSDL) document. Click the **Web Services** option button and specify a path or URL to the WSDL document using the browse **(...)** button.

   📝 **Note:** To test .NET Remoting or other .NET classes, click the corresponding option button (**.NET Remoting** or **Other .NET Classes**) and then specify a .NET Assembly (DLL or EXE) that defines the class.

3. Click **Next**.
   The WSDL file now loads and finds out what is available on the Web service. A WSDL file is an XML description of the functionality that is offered by the Web service.

   📝 **Note:** The sample Web service offers a minimum of functionality. It simply echoes back some values.

4. Select the Web service class you wish to test from the **Class** list box. Normally there will be only one class, but it is possible that the loaded WSDL will define multiple Web services.

5. Using the **URL (Endpoint)** text box, you can change the endpoint of the service. The default endpoint is the one defined in the WSDL document. If required, you can change Web service proxy and authentication settings for the Web server by clicking browse **(...)** and entering information into the **Web Proxy Settings Wizard** dialog box.

6. Click **Next**.
   The **Method invocation** page displays.

7. Check the check boxes beside the methods that you want to have tested, for example echoFloat(Single) and echoString(String). Methods are called using the default parameters.

   📝 **Note:** To see the member functions of the base classes, click **Inherited**. This feature is generally only applicable to the testing of .NET Remoting and other .NET classes.

8. To adjust the sequence in which the methods are called, use the up and down arrow buttons. This way, you can move methods up and down the invocation list. The button between the up and down arrow buttons enables you to invert the selection of methods, meaning that all unmarked methods become marked, and vice versa.

   📝 **Note:** Login and logout methods must be in first and last positions, respectively.

9. Click **Next**.
   The **Select a test case page** displays.

10. Select the test case to which the testing methods should be added. You can select from the following:

    • **Init Test Case**: This is the first test case that is called in test runs. Select this test case if your method calls are related to initialization, logging in, or start-up.
    • **Existing Test Case**: You can add to an existing test case. Choose the test case from the list box to the right of this selection.
    • **New Test Case**: A new test case is created and the methods are added to the new test case. Using the text box to the right of this selection, give the new test case a unique name.
    • **End Test Case**: This is the last test case that is called in test runs. Select this test case if your method calls are related to clean-up, for example logging off.

11. Click **Finish**.

    📝 **Note:** A message displays if you have selected a method that takes parameters, letting you know that default parameters will be used for the method, which may result in an exception being thrown. Methods that throw exceptions will be excluded from your test scenario. If you do not want to proceed, click **No**, then go back, deselect the method in question, and add it later manually with a more meaningful parameter value.

The Web service has been loaded in the **Loaded Components** menu tree and the selected methods have been called and added to the selected test case.

Viewing and Comparing WSDL Files

WSDL files are stored in project files. For older projects, you must update Web Service Proxy classes to store WSDL documents. This is done by updating individual Web Service Proxy classes with a new WSDL file.

1. Select a Web Service Proxy class in the **Loaded Components** menu tree.
   A property called WSDL is displayed in the **Input Data Properties** field.
2. Select the WSDL parameter and click browse **(...)**.
   The **WSDL View** dialog box opens, which includes the original URL, the date the WSDL file was downloaded, and the contents of the WSDL file.
3. Enter a URL in the **Compare With** text box and click **Compare**.
   The content check functionality compares the string contents of both files.

You are informed whether or not there is a difference between the files' contents, but not what the specific difference is.

Defining Client Certificates for a Web Service

For Web services that require client certificates for authentication, client certificates must be defined in Web service properties.

1. Select a Web service class in the **Loaded Components** menu tree.
   A property called ClientCertificates is displayed in the **Input Data Properties** field.
2. Select the ClientCertificates parameter and click browse **(...)**.
   The **Client Certificate** dialog box opens, where certificates can be added or removed.
3. Click browse **(...)** next to the **File** text box to locate and select a certificate (.cer) file.
   The selected client certificate is listed in the **File** text box.
4. Add a certificate file to the list of certificates by clicking the **+** button. Multiple certificates can be defined. Certificates can be removed by clicking the **-** button.
5. Click **OK**.

Once client certificates have been added, the ClientCertificates property description is updated with the number of assigned certificates.

Generated Code

The following code is generated for the test framework and standalone code:

```
mWebProxy1 = new TestWebProxies.Service1();
mWebProxy1.ClientCertificates.Add(System.Security.Cryptography.X5
09Certificates.
X509Certificate.CreateFromCertFile("C:\\TestX509.cer"));
```

When exporting to a native BDL script, the following comment appears:

```
// You have to add the following certificate(s) to your profile!
// C:\TestX509.cer
```

Test Case Characteristics

Following is an overview of the characteristics of test cases as they relate to .NET Explorer:

- Each test scenario includes an `Init` test case and an `End` test case. Multiple `Main` test cases may also be defined. An `Init` test case is the first test case that is called during a test run. `Init` test cases often include method calls that are related to initialization, logging in, or start-up. An `End` test case is the last test case that is called during a test run. `End` test cases often include method calls that are related to clean-up, for example logging off.
- `Main` test cases are executed between `Init` and `End` test cases and may include any sort of method call.
- A local variable is only valid within a single test case and cannot be passed to another test case. Only global variables can be used to pass values between test cases.
- Each test case must have a unique name.
- Test case properties can be edited by selecting a test case in the **Test Scenario** menu tree and defining field values in the **Output Data Properties** field:

  - `StopAtError` is the attribute that defines the number of errors that can occur during an animated run. If this number is reached, the animated run aborts. Setting this attribute to `0` specifies that errors will be ignored.
  - `CallCount` is the attribute that defines the number of times that a test case will be called during an animated run.

- You can create, copy, and remove test cases by right-clicking a test case in the **Test Scenario** menu tree and selecting the respective menu item.

Setting Up Tests

This section explains how to set up and customize your tests with .NET Explorer. Topics include an introduction to GUI elements, customization of input parameters, definition of output value verifications, and generation of random variables. The section concludes with an explanation of how to use .NET Explorer to test .NET Remoting.

Customizing Input Parameters

When a method has input headers or parameters, its values are displayed in the **Input Data** pane with their default values. Select the method you want to invoke in the **Loaded Components** pane. Parameters are displayed with their default values in the **Input Data** pane, and you can configure the option settings through **Tools** > **Options**.

To change a value, select a parameter. If a parameter is a simple data type, which means a string, float, or integer, you can enter the value directly into the **InputValue** text box in the **Input Data Properties** pane. If the parameter is a complex type, which means an object or array, or you want to use a random value or a value from a stored variable, click **(...)** to open the **Input Value Wizard**.

1. Select an `InputValue` row in the **Input Data Properties** pane to display the **Input Value Wizard (...)** button.
2. Click **(...)**.
   The **Input Value Wizard** opens.
3. Define the input value for the parameter of the method.

   You can pass a null value, a value from a variable, or a constant value.

   - Click **Is Null** to pass a null value. This value is only valid for strings, objects, and arrays.
   - If you have already stored a variable that holds values of the parameter's type, click **Use Variable** and select the variable from the list box.

     If the parameter is an array, type the index into the **Index Array** text box.
   - If the parameter is not an object or array, you can specify a constant value such as a string, integer, or double value. Click the **Constant value** option button.
   - If you do not want to use a constant value and would rather use a random value for types such as string, integer, or float, you can create a new random value by clicking **New random variable**. If a random value of the parameter type has already been created, click **Random variable** and select the variable from the list box.
4. Click **OK**.

Once input parameters have been defined, you can invoke a method to have that method automatically added it to the selected test case in the **Test Scenario** pane. From there, corresponding output values can either be stored as variables or verifications can be set up for them.

Storing Output Values in Variables

Using the **Output Data** and **Output Data Properties** panes, you can store output values as variables for later use or you can define verifications for the values.

1. Invoke a method by clicking **Invoke** on the Workflow bar (or by right-clicking a method in the **Input Data** pane and selecting **Invoke/Add to Test Case**) and explore the output data.
2. Select an output value node in the **Output Data** pane.

    You can store such values in variables that can be used as input values for other method calls.
3. Select an `OutputValue` row in the **Output Data Properties** pane to display the **(...)** button. Click **(...)** to invoke the **Output Value Wizard** and store the value.

    > **Note:** The **Output Value Wizard** can also be launched by clicking **Customize** on the Workflow bar, or by right-clicking an output parameter in the **Output Data** pane.
4. In the **Output Value Wizard**, type a name for the output value variable in the **Variable name** text box.
5. Define the scope of the variable.

    The default is *Local*, which means that the variable can only be used within the test case within which it has been created. *Global* scope means that the variable can be used within any test case. Global scope enables you to pass information between test cases.
6. Click **OK**.

    Verifications can be executed against variables. The **Verification Wizard** dialog box offers two text boxes for this purpose: **Variable** and **Array Index** (which defines the array index when a variable is an array).

Defining Global Variables

Global variables can be shared between test cases. These vary from *local* variables, which are only available to specific test cases.

Global variables can be used as input parameters for function calls and can also be used to store return values. Global variables are configured with an initial value, which makes them useful when you need to have the same value used for multiple function calls. Each time a test run begins, global variables are initialized with the defined initial value. The initial value can be changed if the variable is used to store result values.

When exporting global variables to SilkPerformer, the global attribute can be retained as a SilkPerformer project attribute, enabling you to also configure the initial values of global variables.

> **Note:** Global variables of type `string` and `byte[]` can be initialized from data files (`.txt` files). This is useful when a lot of content must be passed to a method call. Rather than manually typing the content into the edit control, you can initialize a variable from a specified file. The **From File** text box is only enabled when the variable type is `System.String` or `System.Byte[]`.

1. Select the **Objects** tab. Right-click an object in the **Objects** tree menu and select **New Global Variable**. The **Global Variable** dialog box displays.
2. Type a name for the variable in the **Variable Name** text box.
3. Select a **Variable type**.

The following types are available:

- string
- integer
- boolean
- double
- single
- byte
- byte array
- date/time

4. Each variable has an initial value and a current value. When tests are run, all global variables are initialized to their initial values. Specify an intial value for the variable in the **Initialize Value** text box. You can also take the initial value from a file by clicking the **from File** option button and then specifying the file location by clicking **(...)**.

5. Click **OK**.

Defining Output Value Verifications

1. Select **Verification** in the **Output Data Properties** pane to display the **(...)** button.
2. Click **(...)** to open the **Verification Wizard**.

   - Select **No Verification** if you do not want the value to be verified.
   - Select **NULL verification** to verify that the value is either null, by checking the **Should be null** check box, or not null. This option verifies whether or not a value is present.
   - Select **Constant string** to verify that the value exactly matches a constant string.
   - Select **Reg. expression** to verify a string value.
   - Select **Range** if you are verifying integers or doubles. Then specify a range for the value using the **From** and **To** text boxes.
   - Select **Variable** to verify that the value exactly matches the value of the variable. If the variable is an array you can also define the **Array index** for the element that should be used for the verification.

3. Select the **Severity** of error that should be thrown if the verification fails. The following severity options are available:

Severity Option	Description
Success, no error	No error is thrown.
Informational	An informational message is logged.
Warning	A warning is thrown.
Error, simulation continues	An error is thrown, but the simulation continues.
Error, the active test case is aborted	An error is thrown, the simulation continues, however the active test case is aborted.
Error, the simulation is aborted	An error is thrown and the simulation is aborted.
Custom	A section is inserted into the .NET code where special error handling code can be inserted.

4. Click **OK**.

Adding Method Calls

1. Within the **Loaded Components** or **Method Call** pane, select the method to be added.
2. Invoke the method by clicking **Invoke**.
   The invoked method is added automatically to the currently selected test case.

Updating Method Calls

1. From the **Test Scenario** pane, select the method that is to be changed.
2. Make modifications to the input/output parameters of the method in the **Input Data** or **Output Data** panes.
   Updates are automatically accepted when you exit the **Input Data** or **Output Data** panes.

Complex Input/Output Data

.NET Explorer supports complex data types such as arrays and objects as both input data and output data. The **Output Value Wizard** assists you in using such data as variables.

Storing Array or Object Output Data as Variable

1. Select the complex output parameter in the **Output Data** pane.
2. Click **(...)** to open the **Output Value Wizard**.
3. Specify the variable's name and scope.

Taking Array Object Data From a Variable

1. Select a complex input parameter in the **Input Data** pane.
2. Click **(...)** to open the **Input Value Wizard**.
3. Define the input value.

Defining Each Member of a Variable Individually

1. Select individual members of an object in the **Input Data** pane.
2. Define input values in the **InputValue** text box.

   For example, you can specify a random value from a value list as one input value and specify a constant value as another input value.

Defining Array Elements

1. The default length for arrays is 1. To change the number of elements in an array, select the array node in the **Input Data** pane.
2. In the **Length** text box, enter the number of elements in the array. .NET Explorer than automatically adds or removes the required elements.
3. To specify an element's value, select the element in the array and specify the value in the **InputValue** text box.
4. Array members or an entire array can be assigned input parameters from a variable. To specify input parameters from a variable, click **(...)** to open the **Input Value Wizard**.
5. Click **Use variable** on the **Input Value Wizard** and select a preconfigured variable from the list box. All variables of the current parameter type are listed.
6. Click **OK**.

Defining a Column-Bound Variable

1. Right-click the **Global Random Variables** tree menu node (**Loaded Components** pane, **Objects** tab) and select **New Multi-Column Variable**.
2. On the **Multi Column Parameter Wizard**, select a file that contains multi-column data. The files that are displayed in the combo box control are the CSV files that are available in the `Data` directory.
3. When a CSV file uses a specific column separator, select the **Separator** so that .NET Explorer can accurately populate the grid displaying the contents of the file.
4. Specify a name for the **Handle name** variable. A file handle variable is a handle that is assigned a row in a multi-column file.
5. Select the column that should be bound to the new variable by clicking within one of the column's cells. The currently selected column is marked with an **(*)** at the end of the column name. **Handle name** and **Parameter name** are automatically generated when you change the file or select a different column if the user has not already changed the name manually.
6. Specify a **File name** for the new bound variable.
7. Click **Finish** to open the CSV file in the registered client application (for example, Microsoft Excel). Changing the file in the client application and re-selecting it in the combo box will reload the file with the revised content.

You will see the new file handle and the new column variable in the **Global Random Variables** tree menu node. To bind another variable to another column in the file, right-click another file handle element in the tree menu and select **New Multi-Column Variable**. The wizard will reappear with read-only file properties (**File name**, **Handle name**, and **Separator**).

These variables can now be used as input parameters for method calls. They will appear in the **Input Parameter** page variable list.

Defining How Column Variables are Bound

Before proceeding with this task you must have defined a column-bound variable.

1. Double-click a multi-column file handle in the **Global Random Variables** tree menu node (**Loaded Components** pane, **Objects** tab).

The **Input Data Properties** table allows you to change the sequence of row access and the amount of time that passes before each subsequent row is selected. When a new row is selected, the bound column variables are set to the values defined in the file.

2. Specify if rows should be selected randomly or sequentially by selecting an option from the `RowSelection` list box.

3. Click within the `InitTestCase` list box and select the test case in which each new row should be selected (for example, If you select a main test case that runs multiple times during each animated run, a new row will be accessed with each run).

Testing .NET Remoting

.NET Explorer supports both Client Activated Objects (CAO), which are remote objects in which clients create new instances of objects on remoting servers, and Server Activated Objects (SAO), or WellKnown remote objects, in which a server offers one instance of a special type (class) and clients access the same object on the remoting server.

Both CAOs and SAOs are accessed through URLs. For CAOs, the Activation URL is the URL where a new CAO can be created, for example **http://localhost**. For SAOs, the URL is the location where the SAO object can be accessed, for example **http://localhost/RemoteObject.rem**.

CAO classes can only be derived from the base class `MarshalByRefObject`. SAOs can be derived from `MarshalByRefObject` or by implementing an interface.

To create a CAO or activated object, right-click a `MarshalByRef` object and select **Create Remote Object**. You will be prompted for the **Activation URL** and for the variable name that will be associated with the created object.

To call methods on a SAO or WellKnown object, simply call the method. You will then be prompted for the Activation URL.

For full details regarding .NET Remoting, refer to the MSDN Online Library at *http://msdn.microsoft.com*.

Negative Testing

Negative testing is testing in which test methods are designed to throw exceptions. Such methods are only considered successfully executed when the anticipated exception type is thrown.

SilkPerformer offers an attribute that can be applied to test methods to indicate that a specific exception type is expected. If the specified exception is not thrown during execution, then the test method fails.

The **Handle Exception** dialog box opens whenever a method throws an exception. The dialog box enables you to specify what to do with the method:

- *Do not add this method to the test case*
- *Add to test case*

  The method is added. If the method throws the exception again during the animated run, it is considered an error.

- *Add to test case and expect this exception during test run*

  The method is added. If the method throws the exception again during the animated run, it is considered a success. If the exception is not thrown, it is considered an error.

Click **Details** to view detailed exception information. The **Explorer Exception** dialog box displays the thrown exception and all inner exceptions. The exception detail grid shows all properties of the exception object. Place your cursor over the value column to display a tool-tip that includes complete content. This is especially useful for the `StackTrace` property as it is a multi-line property.

Expected Exceptions

Each method call that is added to a test case has a property called `Expected Exceptions`. This property holds a list of exception types that are expected to be thrown during an animated run. If you selected **Add to test case and expect this exception during test run** on the **Handle Exception** dialog box, the thrown exception type will automatically be added to this list.

Click **(...)** in the **Output Data Properties** pane to display the **Expected Exceptions** dialog box. This dialog box enables you to add/remove expected exception types.

The **Possible Exceptions** list is populated with the system exceptions. You can enter an exception type that is not of the system types, though to proceed you must click **OK** on a confirmation dialog box informing you of potential impact.

The following code is generated for method calls when the exception `SoapException` is expected:

```
public void TestWebProxies_Service1__GetRandomNumber1() {
try {
Bdl.MeasureStart("Service1::GetRandomNumber");
            mWebProxy1.GetRandomNumber(5, 1);
            Bdl.MeasureStop("Service1::GetRandomNumber");
            Bdl.LogVerification(
             "Expected exception of the type(s):
System.Web.Services.Protocols.SoapException "+
             "has not been thrown!", Bdl.Severity.SEVERITY_ERROR);
} catch (System.Web.Services.Protocols.SoapException expected_0) {
             Bdl.MeasureStop("Service1::GetRandomNumber");
            Bdl.LogException(expected_0, Bdl.Severity.SEVERITY_SUCCESS);
} catch (SilkPerformer.StopException stopExcp) {
            Bdl.MeasureStop("Service1::GetRandomNumber");
            Bdl.LogException(stopExcp);
            throw stopExcp;
} catch (System.Exception e) {
            Bdl.MeasureStop("Service1::GetRandomNumber");
            Bdl.LogException(e);
}
}
```

If an exception is not thrown, a verification of severity `Error` will be logged. The log will contain the exception types that were expected but not thrown.

Animated Runs

Animated runs or *test runs* execute all test methods that are assigned to your test cases, with corresponding input and output data displayed in the **Input Data** and **Output Data** windows as they are submitted/received. Status symbols indicate the success/failure status of method calls after they have run.

Output of animated runs includes several logs and reports that help you to identify problems with tested components.

To configure settings for animated runs go to **Tools** > **Options** > **Options tab**. The following settings are available:

- Always run Init test case
- Always run End test case
- Run all test cases during animated run
- Invoke Delay (time delay between method calls, in ms)

Note: By default, there is a 2-second (2000 ms) delay between the invocation of method calls. This is because .NET Explorer is designed to test the functionality of components, but not to overtax them with excessive load. Invocation delay settings can be configured on the **Options** tab.

Executing an Animated Run

1. Click **Run Test**. The **Animated Run** dialog box displays.
2. Select the test cases that are to be executed during the test run (`InitTestCase`, `TestCase`, or `EndTestCase`). The default selections vary based on your option settings.

   Note: When test cases depend on each other for accessing global objects, ensure that you do not exclude test cases that define objects that are used by other test cases.
3. Check the **Do verifications** check box to specify that verifications that have been defined for output parameters should be performed.
4. Click **OK** to begin the animated run. All global variables are initialized to their initial values. All random variables are reset.

   If you have specified a different calling count for your test cases than the default (1), the test cases will be executed the number of times defined in this property.

   If the number of method calls reporting an error exceeds the number of allowed errors defined in the test case properties, the test case will be aborted.

   Note: Animated runs can be aborted manually by either hitting `Esc` on your keyboard or by selecting `Stop Run` from the **Run** menu. Animated runs are aborted automatically when a verification with severity `Error, the simulation is aborted` fails.

Viewing Status of Animated Runs

Depending on the number of times a test case is called (`CallCount` property) you can see which iteration is currently executing in the **Test Scenario** window. Each test case and test method has one of the following status icons:

- *OK* - Green checkmark
- *Verified* - Blue *V* icon
- *Warning* - Yellow triangle with exclamation mark
- *Error* - Red *X*

Methods that fail due to thrown exceptions or failed verifications display failed or warning method call icons next to them. If a method call fails you can view a description of the failure by holding your cursor over the method.

Analyzing Results

.NET Explorer enables developers and testers to explore the TCP/HTTP network traffic that is sent and received with each Web Service and .NET Remoting call.

Once you have completed a test run, you can view four files to assist you in analyzing test results:

- *User report* - View the user report generated during the last animated run. The report contains statistical information about each test case and each test method call.
- *TrueLog for Web Service calls* - View the TrueLog generated during the last animated run. The TrueLog contains detailed information about Web Service calls.
- *Error report* - View the error report generated during the last animated run. The report contains a description for every error that occurred.
- *Animated log* - This file contains information about every method that has been called with their parameters and errors that occurred.

Viewing Result Files

Before proceeding with this task you must execute a test.

1. Click **Results** on the Workflow bar to display the **Explore Results** dialog box.
2. Select the result type you want to generate.

   Alternately, result files can be selected from the .NET Explorer **Results** menu.

User Report

The user report is based on the SilkPerformer virtual user report. This report offers information regarding time elapsed for each test case and test-method call.

User reports are comprised of four sections:

- Test case summary
- Test case details
- Test method summary
- Errors

The **Test case summary** section offers information about the number of executed test cases and the number of errors that occurred during the test run.

The **Test case details** table gives you information about each test case:

- Number of executions per test case
- Execution time of each test case

The **Test method summary** table offers information about each test method that has been called. You receive information about execution times, number of errors that occurred, and when testing Web Services or .NET Remoting, the number of bytes that were transmitted.

The **Errors** table gives you information about each error that occurred during the test run. You receive information regarding when each error occurred, which test methods caused the errors, and short descriptions of the errors.

TrueLog for Web Service Calls

When testing Web Services, TrueLogs that contain all HTTP traffic for each tested Web Service method are generated.

For each Web Service call there is a TrueLog node that contains request and response HTTP header and body information. The TrueLog contains a transaction node for each test case and a Web document node for each Web Service / .NET Remoting request.

Error Report

The error report delivers an overview of all errors that occur during test runs. They include details regarding error time, in which test case and which test method each error occurred, severity types, and error messages.

Animated Log

The animated log on the **Output** tab displays real-time information about the methods that were called in the most recent test run. Additional information about passed parameters is also included. When exceptions are encountered, the animated log offers detailed stack traces for the exceptions.

The animated log can be accessed at any time by selecting the **Output** tab.

Analyzing Request/Response Network Traffic

.NET Explorer enables developers and testers to explore the TCP/HTTP network traffic that is sent and received with each Web service and .NET Remoting call. Such information can be helpful to developers in seeing how large objects are serialized in XML when they are sent. It is also helpful to testers in load-testing scenarios.

With Web service calls, which are generally XML over HTTP, the actual content of HTTP headers from the requests and responses is displayed.

1. Right-click an invoked Web service call or .NET Remoting method in the **Input Data**, **Output Data**, or **Test Scenario** pane and select **Show Traffic**. The **Request/Response data of method call** dialog box opens.
   Alternative: Select a method call and click **Net Traffic** on the Workflow bar.

The upper left pane of the dialog box shows you the request header information of the call. The upper-right pane shows the reponse header information of the call. The lower-left window shows the request body, which is a SOAP envelope in the case of Web service calls. The lower-right pane displays the response body, which is also a SOAP envelope in the case of Web service calls.

2. Select view tabs to view data in the **request body** and **response body** panes:

   - **Hex View** - Displays traffic data in binary (hexadecimal) view. This is useful when exploring the .NET Remoting binary protocol.
   - **XML View** - Displays SOAP traffic in an XML tree view. This view makes it easier to analyze the XML content of SOAP messages.
   - **Text** - Displays traffic data in a textual representation without formatting.

3. Click **OK**.

Configuring Option Settings

Several application setting options are available in .NET Explorer:

- Default values
- Code
- Verifications
- Animated runs
- Loaded file drop list
- Default proxy connection
- .NET runtime version

Default Value Settings

The **Default Values** options tab enables you to change the default input parameter values for method calls. You can set default values for the following value types:

- Boolean
- Double
- Integer
- String

Boolean can either be `True` or `False`. The other data type values must be selected from available variables, random variables, or constant values.

You can define default value settings for the current project, or you can define system-wide default values, which will be used by all .NET Explorer projects.

Note: Project-wide default value settings override system-wide default value settings.

Configuring Project-Wide Default Values

1. Navigate to **Tools** > **Options**.
   The **Default Values** tab of the **Options** dialog box opens by default.
2. Select a default value row to display that data type's browse (**...**) button.
3. For `Boolean`, select either `True` or `False` from the list box. For all other value types, click (**...**) to launch the **Input Value Wizard**.
4. Click **OK**.

Configuring System-Wide Default Values

If .NET Explorer is running, shut it down.

1. Browse to your .NET Explorer installation directory with Windows Explorer. Locate the file `netexplorer.xml` and open it with a text editor.
2. Locate the `<DefaultValue>` node within the `<config>` node. If it does not exist, you must create it (see the example below).
3. Edit or create the following nodes to set system-wide default values:

   - `<Integer>value</Integer>`: Where `value` must be an integer.
   - `<Boolean>value</Boolean>`: Where `value` must be **true** or **false**.
   - `<Double>value</Double>`: Where `value` must be of type double, for example 1.11, 12.5.
   - `<String>value</String>`: Where `value` must be a string.

4. Save the file and start .NET Explorer. The values you defined will be used as default values for all projects for which you have not defined project-wide default values.

```
<config>
  ...

  <DefaultValue>
    <Integer>111</Integer>
    <Boolean>true</Boolean>
    <Double>1.11</Double>
    <String>my string</String>
  </DefaultValue>
</config>
```

Application Settings

The Microsoft .NET Framework allows you to specify settings in an application-specific configuration file. The configuration file must have the same name as the application and carry the `.config` extension (for example, `myapplication.exe.config`). For ASP.NET Web applications, the extension is `web.config`.

The configuration file is an XML file with the following schema:

```
<configuration>
  <startup />
  <runtime />
  <appSettings>
    <add key="mykey" value="myvalue />
  </appSettings>
  <system.runtime.remoting />
  …
</configuration>
```

When such a file is present, the .NET runtime reads the settings and configures the internal runtime. With the key `appSettings` you can specify key/value pairs that can be read during runtime. A useful application for this is connection strings or other configuration properties. By changing the settings in the XML file, the behavior of the application can be adjusted without changing the code.

The **App Settings** options tab facilitates the testing of .NET components by enabling you to specify key/value pairs. The option page not only allows you to define settings, it also enables you to initialize settings from an existing `.config` file. This saves time when you already have a `.config` file that contains all settings and meaningful values.

You can browse (**...**) to and **Load** a `.config` file, in which case all `appSettings` are used to initialize the grid control, or you can add the entries manually. Changes take effect when you click **OK**.

When a project is exported as a standalone application, an `app.exe.config` file that includes the `appSettings` and the currently selected runtime version is exported.

When exporting to either a Visual Studio .NET project or a SilkPerformer .NET project, a `app.config` file with only the `appsettings` entries is created. The `app.config` file is read by the perfrun when a virtual user is executed. Perfrun takes those settings and uses them to write a `perfrun.exe.config` file, which contains other settings as well.

Code Settings

Multiple options are available for defining the behavior of code when working with .NET Explorer. These settings are available on the **Options** dialog box, **Code** tab.

.NET Code Generation

These project-level options define the behavior of .NET code generation in .NET Explorer:

- *Try-Catch block in every method* - A try-catch block is scripted in each test method. Caught exceptions are forwarded to SilkPerformer .NET Framework. It is recommended that you select this option.
- *Verifications* - Verifications will be scripted in test methods. It is recommended that you select this option to make use of verification settings.
- *Test logic in BDL* - Each test method call becomes a `TestMethod` in the SilkPerformer .NET Framework and therefore becomes a separate method call in the resulting BDL script. It is recommended that you not select this option because test logic should remain in test case methods in the .NET Code.
- *Language* - Select your preferred .NET language for code generation.

Source View

These user-level options define the behavior of the .NET Explorer source tree menu:

- *Show primitive types as objects* - Primitive (simple) types that are stored in global variables are normally displayed as variables, not as objects. When this option is selected, types such as int and string are displayed as objects and you are able to call the methods that are defined for those types, for example `String.Compare`.
- *Show base type information* - When this option is selected, members of the base types of displayed objects are displayed. Otherwise only the members that are declared by the object types themselves are displayed.
- *Use namespaces* - When selected, the full namespace names of loaded classes are displayed in the class tree.
- *Switch to object view when storing a variable*

Verification Settings

Setting options for verifications are available on the **Options** dialog box, **Options** tab.

- *Automatically define a verification for stored values* - When selected, default verifications are defined whenever you store an output value in a variable.
- *Ignore case (default value)* - This is the default value of the **Ignore Case** check box when verifying a string value in the **Verification Wizard**.
- *Check verifications during animated run* - This is the default value for the **Do Verifications** check box on the **Selecting Test Cases** dialog box.

Animated Run Settings

Setting options for animated runs are available on the **Options** dialog box, **Options** tab.

- *Always run init test case* - The init test case is checked by default on the **Selecting Test Cases** dialog box.
- *Always run end test case* - The end test case is checked by default on the **Selecting Test Cases** dialog box.
- *Run all test cases during animated run* - All test cases are checked by default on the **Selecting Test Cases** dialog box.
- *Invoke delay* - Defines the delay between the invocation of method calls in animated runs. The lower the setting, the more frequently methods are called and consequently more load is placed on execution servers.

Editing the Loaded File Drop List

1. Navigate to **Tools** > **Options**.
   The **Options** dialog box displays.
2. Click the **Layout** tab.
   The **Load file entries** text box lists all recently loaded WSDL and DLL files.
3. Select a file entry and click **Delete (X)** to remove it from the **WSDL/DLL** list box.
4. Click **Yes** on the confirmation dialog box.
5. Click **OK**.

Configuring Proxy Connection Settings

1. Navigate to **Tools** > **Options**.
   The **Options** dialog box displays.
2. Click the **Connection** tab.
3. Type the **Address** of the default proxy server.
4. Type the **Port** number of the default proxy server.
5. Type the **User** who has access to the default proxy server.
6. Type the user password into the **Password** text box.
7. Click **OK**.

Specifying .NET Runtime Version

1. Navigate to **Tools** > **Runtime Settings**.
   The **Runtime Configuration** dialog box displays.
2. Select one of the following options:

   - Click the **Use default runtime (as defined in the registry)** option button.
   - Click the **Use a specific runtime from the installed ones** option button and select a version listed in the text box.

3. Click **OK**.

Exporting Projects

.NET Explorer enables you to export projects to SilkPerformer and Visual Studio .NET.

Exporting to SilkPerformer from .NET Explorer is helpful for QA personnel who are proficient with SilkPerformer in facilitating analysis of .NET components and want to create SilkPerformer projects, test case specifications, and scripts from which load tests can be run.

Exporting to Visual Studio .NET from .NET Explorer is helpful for developers who want to use .NET Explorer to quickly generate basic test scripts that can be modified in Visual Studio .NET.

When a developer opens an exported .NET Explorer project in Visual Studio, the virtual user classes, test methods, and all code are generated automatically. From here a developer can add more methods, perform modifications, or execute Try Script runs to confirm the accuracy of scripts. Once a script is ready, it can be passed along to QA in BDL form for use in SilkPerformer.

The other approach available to developers is to export projects directly to SilkPerformer as .NET projects.

Exporting a .NET Explorer Project

1. Open the .NET Explorer **Export** menu and select the required export type:

    *   SilkPerformer Web project
    *   SilkPerformer .NET project

    The **Export Project** dialog box opens.
2. Type a **Project name**, **Project description**, and destination **Directory** for the exported file. By default, projects are saved to the SilkPerformer`Working\Projects` directory.
3. Optionally check the **Open exported project after export** check box.
4. Click **OK**.

If you will be calling a Web service on the Internet, be sure to set the correct proxy in your SilkPerformer project settings.

Exporting a Project to SilkPerformer

Two options are available for exporting projects to SilkPerformer:

SilkPerformer Web Projects

When testing Web Services, .NET Explorer enables you to export SilkPerformer Web projects with native BDL scripts (no .NET code is included) that include `WebPagePost` statements for each Web service call, enabling you to run tests in SilkPerformer to test the Web services while making use of the powerful SilkPerformer Web engine and features such as TrueLog, statistical reporting, modem simulation, and IP spoofing. All random variables, output variables, and verifications are included in exported scripts.

Note: .NET Framework is not required.

- SilkPerformer project file
- BDL Web script
- Random files

SilkPerformer .NET Projects

Your test scenario can be compiled into a .NET assembly that uses the SilkPerformer .NET Framework. SilkPerformer .NET projects are then created with the compiled assembly and a BDF file that contains the calls to the assembly. In other words, .NET test code is compiled into a .NET intermediate language and then executed through SilkPerformer. The value of this is that you can utilize SilkPerformer runtime and scripting to make Web calls even though you are running .NET code and not a Web script.

Note: Visual Studio .NET is not required.

With this intermediate code, when a test case contains Web service calls, the calls are routed over the SilkPerformer Web engine so that you can obtain detailed information about each Web service call and make use of features such as TrueLog, statistical reporting, modem simulation, and IP spoofing.

- SilkPerformer project file
- BDL .NET script
- Random files
- Referenced assemblies

Exporting a Project to Visual Studio .NET

Test scenarios can be exported to C# or VB.NET files and then added to newly created SilkPerformer .NET projects, which can subsequently be used in Visual Studio .NET to run tests. Further code customization in Visual Studio is also possible.

When you export a test scenario to a SilkPerformer Visual Studio .NET project, .NET Explorer creates a SilkPerformer .NET project and a corresponding Visual Studio .NET project. The test code that you define in your test scenario is added to your project and you can continue to work in Visual Studio .NET to make changes to the script. The exported test code makes use of the SilkPerformer .NET Framework.

Once you have finished modifying test code, you can either continue to work in SilkPerformer or you can run a test from within Visual Studio .NET.

When a test case contains Web service calls, the calls are routed over the SilkPerformer Web engine so that you can obtain detailed information about each Web service call and make use of features such as modem simulation and IP spoofing.

- SilkPerformer .NET project
- Visual Studio project (C# or VB)
- Source files (test driver and proxies for Web services)

Exporting a BDL Script

1. Open the .NET Explorer **Export** menu and select the required export type (`BDL Web script` or `BDL .NET Framework script`).

2. On the **Save as** dialog box, type a name and destination for the exported BDF file. By default, projects are saved to the SilkPerformer `Projects` directory at `...\Working\Projects\<project name>`.

Exporting a Standalone Console Application

.NET Explorer can compile test cases into standalone console applications that call all test case methods in their main methods.

Console applications can be used to either run test scenarios independently of .NET Explorer, or to generate network traffic that can be recorded by SilkPerformer. SilkPerformer can consequently generate Web BDL scripts based on the network traffic.

1. Open the .NET Explorer **Export** menu and select **Standalone Console Application**.

2. On the **Export Standalone Console Application to File** dialog box specify a name and destination for the exported EXE file. By default, projects are saved to the SilkPerformer project's `Projects` directory at `...\Working\Projects\`.

Recording a Console Application With SilkPerformer

1. Launch SilkPerformer and create a new Web project.

2. Create a new application profile for the console application.

3. Specify a name for the profile.

4. Specify the path to the executable.

5. Select **Custom Application** as the application type.

6. Select **Winsock**.

7. Begin recording the application.

After recording you have a Web BDL script that contains all the HTTP/TCP calls of the console application to a Web service or .NET Remoting component.

Command Line Execution

Overview

It makes sense to run some functions from the command line, particularly those that are run numerous times on a daily basis.

The following .NET Explorer functions can be called from the command line:

Launch .NET Explorer

As with most applications, .NET Explorer can be launched from the command line. If you installed SilkPerformer in the default directory, the command is `C:\Program Files\Silk\SilkPerformer 2010 R2\DotNET Explorer \netexplorer.exe`.

Execute animated runs for one or all test cases

You can launch .NET Explorer and have animated runs start immediately for one or all test cases in a project using a command line parameter:

- To execute all test cases in a project, enter for example `netexplorer.exe myproject.nef *` at the command line.
- To execute only the main test case in a project, enter for example `netexplorer.exe myproject.nef TMain` at the command line.

Write log files

When executing projects, result files (`log`, `html`, and `error`) are stored in project directories (`.nef` file directories). These files receive the same name as the `.nef` file, for example `myproject.err`, `myproject.log`, and `myproject.html`.

To define a different base name for these files, specify the log filename as an additional parameter in the command line, for example `netexplorer myproject.nef * mylogfilename`.

Usage

```
netexplorer netfile testcase | * [output]
```

Parameter	Description
netfile	.NET Explorer file (`.nef`)
testcase	The name of the test case to execute, or * to execute all
output	The directory to write reports to. If no path information is passed, the files are written to the project directory. The following result files are written: `.log`, `.html`, and `.err`

Index

RMI samples, Java 10