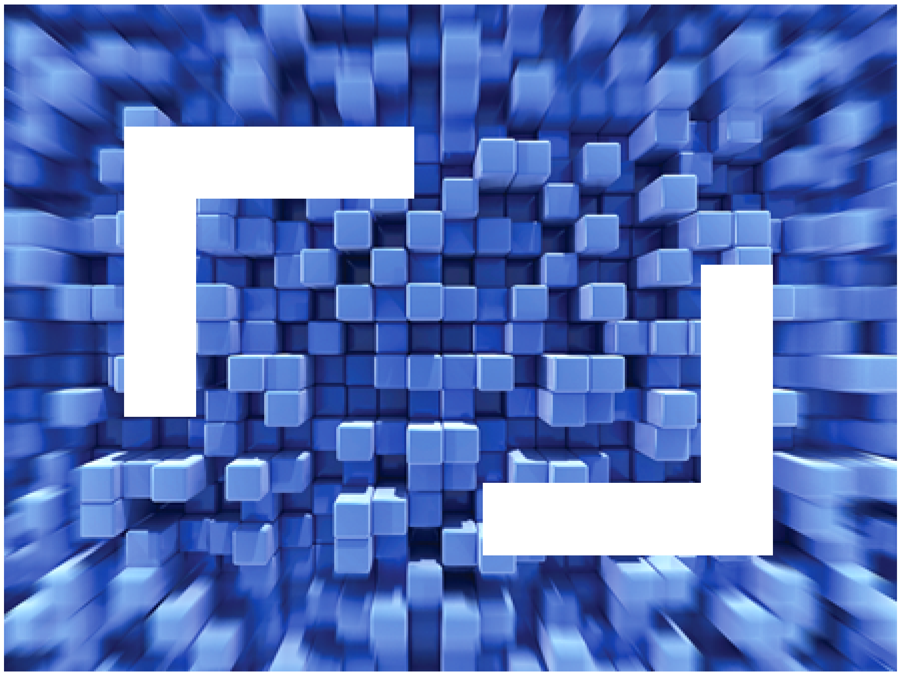


SilkPerformer 2011

.NET Framework Developer Guide

Borland®
(A MICRO FOCUS COMPANY)

MICRO FOCUS®

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

Contents

Tools and Samples

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

Introduction to SilkPerformer SOA Edition

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

SilkPerformer SOA Edition Licensing

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

What You Can Test With SilkPerformer SOA Edition

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

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

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

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

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

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

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

Provided Tools

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

SilkPerformer .NET Explorer

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

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

SilkPerformer Visual Studio .NET Add-On

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

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

The Add-On offers the following features:

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

.NET Resources

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

SilkPerformerJava Explorer

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

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

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

Java Resources

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

SilkPerformer Workbench

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

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

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

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

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

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

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

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

Sample Applications for testing Java and .NET

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

Sample applications for the following component models are provided:

- Web Services
- .NET Remoting
- Java RMI

Public Web Services

Several Web Services are hosted on publicly accessible demonstration servers:

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

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

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

.NET Message Sample

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

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

To access the .NET Message Sample:

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

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

.NET Explorer Remoting Sample

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

To access the .NET Explorer Remoting Sample:

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

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

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

Java RMI Samples

Four Java RMI sample applications are included:

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

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

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

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

Java Remote Method Protocol (JRMP)

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

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

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

Host: `localhost`

Port: `1099`

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

RMI over IIOP

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

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

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

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

Server: `Sun J2EE Server`

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

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

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

Sample Test Projects

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

.NET Sample Projects

.NET Remoting

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

Web Services

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

Java Sample Projects

JDBC

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

RMI/IIOP

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

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

RMI

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

To access the Java RMI server:

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

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

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. Microsoft Visual Studio offers wizards that allow you to specify the URLs of Web services. Microsoft Visual Studio 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 Microsoft Visual Studio 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 Microsoft Visual Studio.

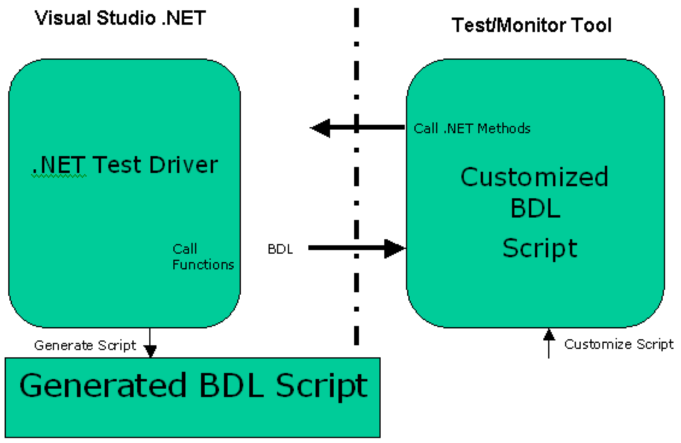
Note: For additional details regarding the available BDL API functions, refer to the *Benchmark Description Language (BDL) Reference*.

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 Microsoft Visual Studio through the Add-On. All user output and generated output files, like TrueLogs, logs, output, and others, can be viewed from within Microsoft Visual Studio.

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.

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.

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 2008 and 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** > **Silk** > **SilkPerformer2011** > **SilkPerformer Workbench** and create a new project with the application type **.NET** > **.NET Framework using Visual Studio .NET Add-On**.

Load Testing .NET Components

This section explains how to use the SilkPerformer Visual Studio .NET Add-On for the testing of .NET components and Web services.

Setting Up SilkPerformer .NET Projects

1. Click **Start here** on the SilkPerformer workflow bar.

   Note: If another project is already open, choose **File** > **New Project** from the menu bar and confirm that you want to close your currently open project.

   The **Workflow - Outline Project** dialog box opens.
2. In the **Project** text box, enter a name for your project.
3. In the **Project description** text box, enter a description for your project.

   This description is for your own project management purposes only.
4. From the **Application** menu tree, choose **.NET** > **.NET Framework using Visual Studio .NET Add-On** and click **OK**.
   The **Workflow - Model Script** dialog box opens.
5. Select your preferred .NET Language (**C#**, **VB.NET**, or **Managed C++**) and click **OK**.
   Microsoft Visual Studio's opens.
6. Enter the name of the .NET Testclass in the **Name of testclass** text box. In the text box, enter the name of the project that you created earlier in SilkPerformer.
7. Click **Finish**.

The following in the files and code are generated in Microsoft Visual Studio:

- Each generated Testclass becomes a VirtualUser in the BDL script.
- The first transaction becomes the `Init` transaction in the BDL script.
- Files that are generated by the Wizard (code files and SilkPerformer project/BDL scripts) are listed on the **Solution Explorer** page.
- Handler/clean-up code can be inserted in the `stopException` method.
- Custom code for exception handling can be inserted in the `testException` method.
- `ETransactionType.TRANSTYPE_MAIN` becomes the `Main` transaction in the BDL script.
- `ETransactionType.TRANSTYPE_END` becomes the `End` transaction in the BDL script.

Sample Skeleton Code Generated by the Project Wizard (C#)

```
using System;
using SilkPerformer;

namespace SPProject1
{
  [VirtualUser("VUser")]
  public class VUser
  {
    public VUser()
    {
    }

    [Transaction(ETransactionType.TRANSTYPE_INIT)]
    public void TInit()
    {
      /* You can add multiple TestAttribute attributes to each
function defining parameters that can be accessed through
Bdl.AttributeGet


      Example of testcode: (Access bdl function through the
static functions of the Bdl class Bdl.MeasureStart(...);
      ...
      Bdl.MeasureStop(...);
      */
    }

    [Transaction(ETransactionType.TRANSTYPE_MAIN)]
    public void TMain()
    {
    }

    [Transaction(ETransactionType.TRANSTYPE_END)]
    public void TEnd()
    {
    }
  }
}
```

As you can see from the skeleton example above, there is a custom attribute called `VirtualUser` that can be applied to classes. This causes the Add-On's BDL Generation Engine to generate a virtual user definition. You can implement multiple classes that have the `VirtualUser` attribute applied. The `VirtualUser` attribute takes the name virtual user as a parameter.

The BDL Generation Engine then parses the methods of the Virtual User class for methods that have a `Transaction` attribute applied to them. The `Transaction` attribute takes as a first parameter the transaction type (`Init`, `Main` or `End`). You can only have one `Init` and one `End` transaction, but multiple `Main` transactions.

The `Main` transaction type takes a second parameter that indicates the number of times that the transaction is to be called during load tests (default: 1).

Creating a Web Service Client Proxy

Microsoft Visual Studio includes a wizard that generates a Web Service client proxy that you can use to call Web Service methods. The wizard is launched via **Project** > **Add Web Reference**.

1. Type the URL of your Web Service into the top text box, for example, *http://demo.borland.com/BorlandSampleService/BorlandSampleService.asmx?WSDL*, and click **Enter.**
   The **Add Web Reference** button is enabled when the wizard loads the WSDL document from the specified URL.
2. Click **Add Web Reference**.
   The wizard generates a proxy class in a namespace that is the reverse of the name of the Web server that hosts the service (for example, `demo.host.com` becomes `com.host.demo`).
3. Explore objects to see which classes have been generated. Each Web Service, and all complex data types used by the Web Service methods, are represented as classes.

   In the generated proxy code, the proxy class takes its name from the Web Service. The namespace of the class is the reverse of the name of the Web server that hosts the service. All files that are generated by the **Add Web Reference** wizard are displayed on the **Solution Explorer** tab.

   Note: The **Show All Files** option must be activated to display all generated files.

Instantiating Client Proxy Objects

To instantiate a client proxy object you can declare a variable of the client proxy class as a public member variable of the .NET test driver. The variable should be instantiated either in the constructor or in the `Init` transaction. The first part of the namespace where the proxy class is generated is the name of your project, as this is the default namespace.

1. Once you have instantiated a proxy class object, make calls to the service by inserting Web Service invocation code into a main transaction. Call the Web Service methods using simple parameters.
2. Use `MeasureStart` and `MeasureStop` to measure the time required for the methods to execute.
3. Print the result of the `echoString` method.

You can also call a Web Service method that takes an object as a parameter. To do this, instantiate the object, set the member values, and pass the object to the Web Service.

Note: You can catch exceptions and log them in the TrueLog.

Try Script Runs From Microsoft Visual Studio

Once you have implemented your .NET test code, you can execute a Try Script run from Microsoft Visual Studio by calling **Run** > **Try Script** from the SilkPerformer menu. Try Script runs are trial test runs that you can use to evaluate if your tests have been set up correctly.

The steps that are then performed by the Add-In are as follows:

- The .NET code is compiled into a .NET assembly.
- A BDF script is generated based on the meta information of the custom attributes and the settings in the **Options** dialog box.
- The most recent BDF script is overwritten if there have been changes to the meta data of your assembly (for example, changed custom attributes, method order, or generation options).
- If the meta data has changed, but you have altered the latest BDF file manually, you will be prompted to confirm that you want to have the file overwritten. This detection is achieved by comparing the last modified date of the BDF file with the timestamp scripted in the BDF file.
- If you have multiple virtual user classes (classes that have the `VirtualUser` attribute applied) you will be prompted to specify which of the users is to be started.

Executing a Try Script Run

1. Select **Run** > **Try Script** from the SilkPerformer menu.

   Note: If you are accessing a Web Service on the Internet, ensure that you have configured proxy settings for the active profile.
2. If you have multiple virtual user classes, select the virtual user that you want to execute from the **Select Virtual User** dialog box.

   Note: If there are multiple test classes, you must select the test class that you want to execute.
3. Click **Run** to begin the test.

   Note: If the **Automatic Start when running a Try Script** option has been selected in SilkPerformer options, TrueLog Explorer will launch showing the TrueLog that was generated by the test.

Virtual user return-value output can be viewed in the **Virtual User** output tool window within Microsoft Visual Studio via the `Bdl.Print` method. The output window can be docked to other windows. Test controller output is displayed in a separate pane of the output tool window.

Note: `WebDotNetRequest` entries are Web Service calls that are routed over the SilkPerformer Web engine.

TrueLog Explorer launches automatically during Try Script runs. Each Web Service call has a node in the displayed TrueLog. The nodes in the main transaction represent the SOAP HTTP traffic that was responsible for the Web Service calls. By default, all HTTP traffic is redirected over the SilkPerformer Web engine, enabling TrueLog output. You can turn off redirection or enable it for specific Web Service client proxy classes via the SilkPerformer **Web Settings** dialog box.

4. Using the TrueLog Explorer XML control, explore the SOAP envelope that was returned by each Web Service call.

Once the test is complete you can explore other result files (log, output, report, and error) by selecting them from the SilkPerformer **Results** menu.

Web Service Calls

The SilkPerformer .NET Framework can route Web traffic generated by .NET components over the SilkPerformer Web engine. This means that the SilkPerformer Web engine executes the actual Web requests, allowing you to see exactly what is sent over the wire. This enables you to make use of SilkPerformer Web engine features such as modem simulation, IP multiplexing, network statistics, and TrueLog.

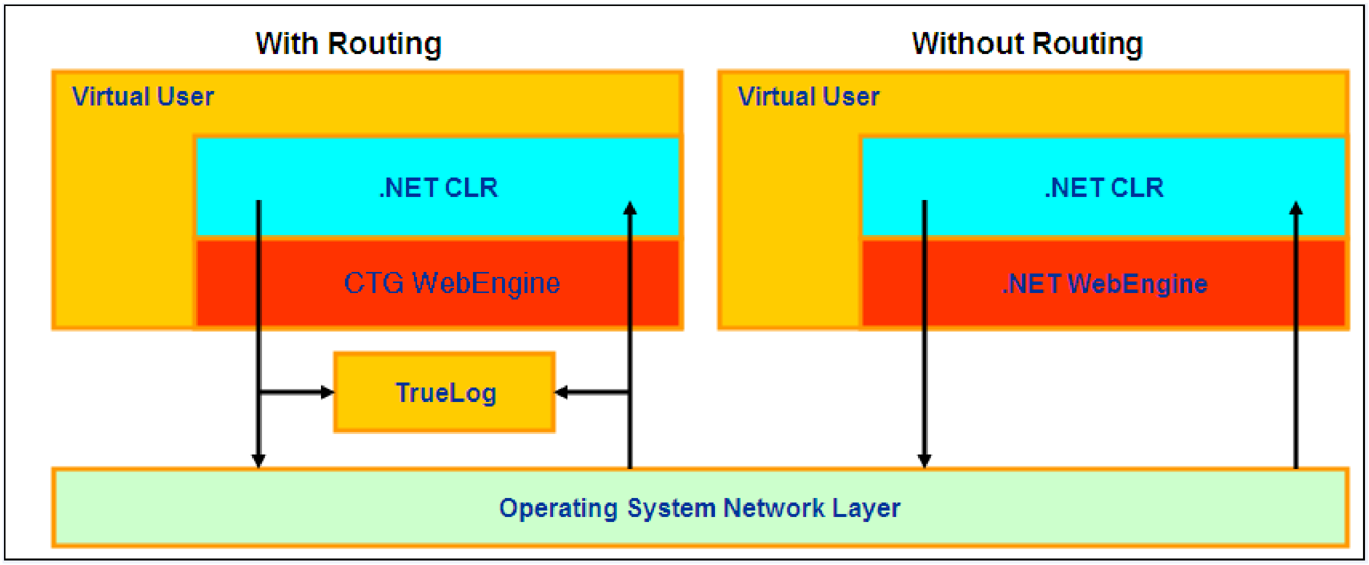
By default, all network traffic is routed over the Web engine. You can however enable routing only for specific Web Service client proxy classes. To enable this feature only for specific Web Service proxy classes, change the base class of a proxy class from `SoapHttpClientProtocol` to `SilkPerformer.SPSoapHttpClientProtocol`.

This base class exchange allows the SilkPerformer .NET Framework to generate more detailed statistical information for each Web Service call. It is recommended that you enable this feature for all of your Web Service proxy classes. This can be done using Visual Studio’s **Web Service** dialog box, which is accessible via the SilkPerformer menu.

When this feature is disabled, the .NET HTTP classes process all requests.

For each Web Service call, a node is created in the TrueLog with the SOAP envelope that was passed to the Web Service and returned to the client.

When all or some classes are instrumented by SilkPerformer, the HTTP traffic responsible for Web Service calls is routed over the SilkPerformer Web engine. Network traffic and statistics are then written to the TrueLog. Modem simulation and IP multiplexing are also available.



Microsoft Web Service Enhancement SDK

A special method of the SilkPerformer Helper Class is required when using the Microsoft Web Service Enhancement (WSE) SDK to call secure Web Services.

Microsoft WSE SDK uses multiple threads to fulfill SOAP requests. When Web traffic routing is enabled, these threads make use of the SilkPerformer Web Engine. For synchronization purposes, it i required that you call the `Bdl.SetVUserContext()` method in your test code before you make the first Web Service request. This needs to be the first call in the `TInit` transaction.

Routing Web Service Calls

1. Open the **Web Services** dialog box (select **Web settings** from the SilkPerformer menu).
2. Select the Web Service proxy classes that should be instrumented by SilkPerformer.

   These are the classes that will be routed over the SilkPerformer Web engine.

   Select **Instrument all HTTP/HTTPS traffic** to have all calls routed or select specific proxy classes for routing.

Dependencies

You can specify the files upon which your .NET code depends using the **Add Dependencies** dialog box (select **Add Dependencies** from the SilkPerformer menu).

The files you specify will be added to your SilkPerformer project's data files section. This ensures that those files will be available on agents when you run tests that use multiple agents. To get the path to a file you have added to the data files section, use the `GetDataFilePath` function of the BDL object. This function returns the absolute path to the file. If you run a Try Script on `localhost`, the path will be to your original file. If you run a test it will return the path in the agent's data directory.

Adding Dependencies

1. Select **Add Dependencies** from the SilkPerformer menu to open the **Add Dependencies** dialog box.
2. Click **Add file** to browse to and select a file that you want to add.

   To remove a selected file, click **Remove**.

3. Click **OK** to accept the dependent file list.

   Note: All files in the SilkPerformer project's data files section will be copied to the agent that executes the test. To get the full path to a file, use the `Bdl.GetDataFilePath` function with the filename as a parameter. This function ensures that you receive the correct path to your file, regardless of whether or not the file was executed locally or remotely.

Configuring .NET Add-In Option Settings

1. From the SilkPerformer menu, select **Options** to open the **Options** dialog box.
2. Check the **Automatic Start when running a Try Script** check box to have TrueLog Explorer launch automatically and display the TrueLog of the current Try Script.
3. In the **Virtual User Output** group box, define which types of information you want to have displayed in the **Virtual User Output** window.
   - **Errors**
   - **Transactions**
   - **Functions**
   - **Information**
   - **User Data**
   - **All Errors of all Users**
4. In the **BDL Script Generation** group box, specify BDF script-generation settings.

Option	Description
DotNetCallMethod	When checked, `MeasureStart` and `MeasureStop` statements are scripted around each `DotNetCallMethod` call.
Generate BDH for .NET Method Calls	When checked, a BDH file that contains BDL functions for each .NET call is generated. This makes the main BDF file slim as it only includes the BDL function calls in the transactions.
Generate BDL functions for .NET Methods	When checked, a BDL function is scripted for each .NET call. The transactions then call the functions.

5. Click **OK** to confirm the settings.

Continuing Your Work in SilkPerformer

Once you have finished implementing your .NET test driver you can continue running tests with SilkPerformer. You can open your .NET project in SilkPerformer by selecting the*Open in SilkPerformer* command from the *SilkPerformer* menu.

In SilkPerformer, you can run tests with multiple users distributed over multiple agents. Take advantage of the SilkPerformer Web engine features (modem simulation and IP-address multiplexing) by testing how Web Service calls perform when they are called over a slow modem and how the Web server performs when numerous users make simultaneous service calls.

Custom Attributes

A custom attribute called `VirtualUser` can be applied to classes. This attribute instructs the Add-In's BDL generation engine to generate a virtual user definition. You can implement multiple classes that have the `VirtualUser` attribute applied to them. The `VirtualUser` attribute takes the name `virtual user` as a parameter.

Note: When a BDF file is modified manually, you are prompted to specify whether or not you want to have the file overwritten.

The BDL generation engine parses the methods of the `VirtualUser` class for methods that have a `Transaction` attribute applied to them. The `Transaction` attribute takes the transaction type, `Init`, `Main` or `End`, as a first parameter. You can only have one `Init` and one `End` transaction, but multiple `Main` transactions are allowed.

The `Main` transaction type takes a second parameter that indicates the number of times that the transaction is to be called during a test (the default is `1`).

Following are the available custom attributes and what the BDL generation engine scripts for them.

Attribute Class	Applicable to	Parameters	Description
VirtualUser	Class	Name of the Virtual User Group (optional) IsUnitTest	Defines a Virtual User Group. If you specify true, DotNetUnitTest methods will be scripted instead of the standard DotNet methods (e.g., DotNetUnitTestLoadObject).
Transaction	Method	Type (Init, Main, End) If type is Main the number of transaction iterations (optional) Name	Defines a Transaction for the Virtual User Group. The transaction implementation will call the method of the .NET Object. The first script call in the Init transaction is a DotNetLoadObject loading the Object The last script call in the end transaction is a DotNetFreeObject. Optionally you can define a name that should be used in the generated BDL script for this transaction. By default, the transaction name in BDL is created by combining the VUser name and the method name.
TestMethod	Method		This will script a call to the method in the current transaction. The current transaction is the previous method with a Transaction attribute. So a method with this attribute that has no prior method with a Transaction attribute makes no sense.
TestAttribute	Method	Attribute Name Attribute Value (optional) Description	This can be applied multiple times to a method that has either a Transaction or TestMethod attribute. An AttributeSetString function will be scripted prior to the DotNetCallMethod that calls this method. AttributeSetString will set an attribute with the passed name and value. This is a way how parameters can be passed from the script to the .NET function. The .NET function can read the attributes with Bdl.AttributeGet. Its meant that people (QA) who will receive the finished script only have to change the value passed to the AttributeSetString to customize the script. So there is no need for them to change the .NET Code. Allows you to define a description for the project attribute. The description can be seen in SilkPerformer's project attribute wizard.
VirtualUserInitialize	Method		This method is called for classes that are loaded via DotNetUnitTestLoadObject

Attribute Class	Applicable to	Parameters	Description
VirtualUserCleanup	Method		This method is called for classes that are freed via DotNetUnitTestFreeObject
TestCleanup	Method		This method is called after a method is called via DotNetUnitTestCallMethod
TestInitialize	Method		This method is called before a method is called via DotNetUnitTestCallMethod
TestIgnore	Method		Methods that have this attribute applied to them will not be called via DotNetUnitTestCallMethod
TestException	Method	Type of exception Additional log message	Normally, methods that throw exceptions are considered failed. If you want a method to throw an exception, you can use the TestException attribute to tell SilkPerformer that this method is supposed to throw an exception.

Attributes for Unit Test Standards

Unit-testing frameworks such as NUnit and Microsoft Unit Test Framework introduce attributes for methods that are to be called before and after test methods. These methods are called `Setup/Initialize` and `TearDown/Cleanup`.

To comply with these standards, four attributes are offered:

Attribute Class	Applicable to	Parameters	Description
VirtualUserInitialize	method		This method is called before a normal test method/transaction is called. It can be used for the global initialization of variables that all test methods use. Only one method with this attribute is allowed per virtual user.
VirtualUserCleanup	method		This method is called after each test method is called. It can be used for global clean-up. Only one method with this attribute is allowed per virtual user.
TestInitialize	method		This method is called before each test method/ transaction. It can be used to initialize variables that are utilized by the subsequent test method.
TestCleanup	method		This method is called after each test method/ transaction. It can be used for clean-up after a test method call

Negative Testing

Negative testing is testing in which test methods are designed to throw exceptions. Such methods should only be considered successful when a specific 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 has failed.

Attribute Class	Applicable to	Parameters	Descriptions
TestException	method	- Exception type - Log text if the anticipated exception is not thrown *(optional)*	Test method/transactions can be declared with one or more `TestException` attributes. During execution, the runtime checks to see if the defined exception type was thrown. If the anticipated exception type was thrown, then the method call is considered successful. If not, the method call is considered a failure and exception details are written to the log file.

Custom Attributes Code Sample

C# Test Code Sample

```
using System;
using SilkPerformer;

namespace SPProject1
{
  [VirtualUser("VUser")]
  public class VUser
  {
    public VUser()
{
    }
    [Transaction(ETransactionType.TRANSTYPE_INIT)]
    public void TInit()
    {
    }
    [Transaction(ETransactionType.TRANSTYPE_MAIN)]
    public void TMain()
    {
    }
    [TestMethod]
    [TestAttribute("Attr1", "DefaultValue1")]
    public void TestMethod1()
    {
      string sAttrValue = Bdl.AttributeGet("Attr1");
      Bdl.Print(sAttrValue);
    }
    [Transaction(ETransactionType.TRANSTYPE_END)]
    public void TEnd()
    {
    }
  }
}
```

Generated BDF Script Example

```
benchmark DOTNETBenchmarkName

use "dotnetapi.bdh"

dcluser
 user

    VUser
 transactions
   VUser_TInit : begin;
   VUser_TMain : 1;
   VUser_TEnd  : end;
var
 hVUser : number;

dcltrans
 transaction VUser_TInit
 begin
   hVUser:= DotNetLoadObject("\\SPProject1\\bin\\release\
\SPProject1.dll", "SPProject1.VUser");
   MeasureStart("TInit");
   DotNetCallMethod(hVUser, "TInit");
   MeasureStop("TInit");
 end VUser_TInit;

 transaction VUser_TMain
 begin
   MeasureStart("TMain");
   DotNetCallMethod(hVUser, "TMain");
   MeasureStop("TMain");
   AttributeSetString("Attr1", "DefaultValue1");
   MeasureStart("TestMethod1");
   DotNetCallMethod(hVUser, "TestMethod1");
   MeasureStop("TestMethod1");
 end VUser_TMain;

 transaction VUser_TEnd
 begin
   MeasureStart("TEnd");
   DotNetCallMethod(hVUser, "TEnd");
   MeasureStop("TEnd");
   DotNetFreeObject(hVUser);
 end VUser_TEnd;
```

Testing SOAP Web Services

This section explains the basics of SOAP based Web services and details how you can test them.

Simple Object Access Protocol (SOAP)

Simple Object Access Protocol (SOAP) is a lightweight XML-based protocol that is used for the exchange of information in decentralized, distributed application environments. You can transmit SOAP messages in any way that the applications require, as long as both the client and the server use the same method. The current specificationdescribes only a single transport protocol binding, which is HTTP.

SOAP perfectly fits into the world of Internet applications and promises to improve Internet inter-operability for application services in the future. In essence, SOAP packages method calls into XML strings and delivers them to component instances through HTTP.

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

Microsoft is one of the greatest advocates of SOAP and has incorporated SOAP as a standard interface in the .NET architecture.

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

Sample Call

```
POST /StockQuote HTTP/1.1
Host: www.stockquoteserver.com
Content-Type: text/xml;
charset="utf-8"
Content-Length: nnnn
SOAPAction: "Some-URI"
<?xml version="1.0"?>
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/
encoding/">
  <SOAP-ENV:Body>
    <m:GetLastTradePrice xmlns:m="Some-URI">
      <symbol>DIS</symbol>
    </m:GetLastTradePrice>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

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

The additional HTTP header `SOAPAction` is mandatory for HTTP based SOAP messages, and you can use it to indicate the intent of a SOAP HTTP request. The value is a URI that identifies the intent. The content of a `SOAPAction` header field can be used by servers, for example firewalls, to appropriately filter SOAP request messages in HTTP. An empty string ("") as the header-field value indicates that the intent of the

SOAP message is provided by the HTTP Request-URI. No value means that there is no indication on the intent of the message.

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

The programm that sends this message only needs to understand how to frame a request in a SOAP-complient XML message and how to send it through HTTP. In the following example, the program knows how to format a request for a stock price. The HTTP server that receives the message knows that it is a SOAP message because it recognizes the HTTP header `SOAPAction`. The server then processes the message.

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

Sample Response

```
HTTP/1.1 200 OK
Content-Type: text/xml;
charset="utf-8"
Content-Length: nnnn
<?xml version="1.0"?>
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/
encoding/"/>
  <SOAP-ENV:Body>
    <m:GetLastTradePriceResponse xmlns:m="Some-URI">
      <Price>34.5</Price>
    </m:GetLastTradePriceResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

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

XML headers enclose the actual SOAP payloads. The XML element `GetLastTradePriceResponse` contains a response to the request for a trading price. The child element is `Price`, which indicates the value that is returned to the request.

Testing SOAP over HTTP-Based Web Services

You can use one of the following three SilkPerformer options to test SOAP over HTTP-based Web services:

1. Record and replay the HTTP traffic.
2. Use .NET Explorer in combination with the SilkPerformer .NET Framework. For additional information, refer to the *.NET Explorer Help*.
3. Use Java Explorer in combination with the SilkPerformer Java Framework. For additional information, refer to the *Java Explorer Help* and the *Java Framework Help*.

Your customer environment and the prerequisites determine which one of these approaches is best suited to your needs.

Recording and Replaying HTTP Traffic

Recording the SOAP protocol over HTTP is as straightforward as recording any Web application that runs in a browser. The application that you record is the application that executes the SOAP Web-service calls. This can either be a client application or a part of the Web application itself.

Creating a New XML/SOAP Project

When you want to record and replay HTTP traffic to test SOAP over HTTP-based Web services, you first need to create a new SilkPerformer project of the **Web Services** > **XML/SOAP** type.

1. In SilkPerformer, click **File** > **New Project**.
   The **Workflow - Outline Project** dialog box opens.
2. Type a name for the project in the **Project** text box.

   For example, `testSOAP`.
3. From the **Application** tree select **Web Services** > **XML/SOAP**.
   This application type automatically configures its profile settings so that `SOAPAction` HTTP-headers, that are used by SOAP-based applications when calling Web services, are to be recovered.

Creating the Application Profile

When you want to record and replay HTTP traffic to test SOAP over HTTP-based Web services, you need to create an application profile for the client application that you want to record.

1. In SilkPerformer, click **Settings** > **System**.
   The **System Settings** dialog box opens.
2. In the **System** group box, click **Recorder**.
   The **Application Profiles** page opens.
3. Click **Add** to add a new application profile to the list.
   The **Application Profile** dialog box opens.
4. Type a name for the application profile in the **Application profile** text box.

   For example `Internet Explorer`.
5. Click **Browse ...** next to the **Application** path text box and select the path to the application executable.

   For example `C:\Program Files\Internet Explorer\Explorer.exe`.
6. Define the **Working directory**.

   For example `C:\Program Files\Internet Explorer`.
7. Define the **Program arguments**.

   For example **about:blank**.
8. Select the application type from the **Application Type** list box.

   For example `MS Internet Explorer`.
9. In the Protocol selection area, check the check box that corresponds to the protocol that you want to use.
   For example, check the **Web** check box.
10. Click **OK**.

Recording a Script

Record a script with your created application profile.

1. Select **Record** > **Recorder**.
2. Interact with your client application.
   The recorder records all SOAP requests that are executed over HTTP/HTTPS.

3. When you are finished, close the application and click **Stop** on the recorder.
   The **Save As** dialog box opens.
4. Select the desired location for the recorded script and click **Save**.

Script Customization

Each SOAP request that is recorded includes a `WebHeaderAdd` and a `WebUrlPostBin` API call.

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

Sample SOAP Request

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

Replaying a Script

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

Select how you want to replay your script. The following options are available:

- Start a Try Script run.
- Replay the script as part of a baseline identification.
- Replay the script in a test.

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

SilkPerformer .NET Explorer

This section offers a brief overview of how you can use .NET Explorer to test Web services with SilkPerformer. .NET Explorer is a tool that allows you to create test cases through a point and click interface. .NET Explorer provides support for the following .NET technologies:

- SOAP Web services
- .NET Remoting
- .NET Components (other classes)

You can use the .NET Explorer to create test scenarios. You can then use the test scenarios to run component testing in .NET Explorer or you can export the test scenarios to SilkPerformer for testing.

You can also use .NET Explorer to generate test drivers, in either C# or VB.NET, that contain all test logic defined for test scenarios. You can export the test drivers to Visual Studio .NET where you can make further customizations and where you can execute TryScript runs directly in Visual Studio .NET. Once you have completed the customization, you can execute tests with SilkPerformer.

.NET Explorer requires permanent projects. This was not the case with previous versions as it was possible to save current projects at any time. When you launch .NET Explorer you are prompted to either create a new project or open an existing project by choosing from a recent file list or by browsing for a .NET Explorer project file, of type `.NEF`.

For additional details on .NET Explorer, refer to the .NET Explorer Help.

Requirements

The only requirement for testing a SOAP-based Web service is a description of the exposed methods of the Web service. You can find such descriptions in Web Service Description Language (WSDL) files, which are normally generated when you browse a Web-service end point or you specify special `HTTP GET` parameters for retrieving such files.

In the case of an ASP.NET Web service, appending `?WSDL` to the URL end point of the Web service will return the WSDL file for the Web service. Other SOAP-stack implementations may use the same approach or offer WSDL files under separate URLs.

Loading a WSDL File

.NET Explorer offers the **Load File Wizard** that guides you through the steps required to load a WSDL file and invoke Web-service methods. To activate the wizard in .NET Explorer, click **Start Here** on the toolbar.

You can also load WSDL files, or .NET assemblies that contain .NET Remoting or other .NET classes, through the address bar of your browser. Specify the URL or path to the WSDL file and click **Load**.

Microsoft .NET Framework offers classes that load WSDL files and generate client proxies for Web services that are defined in WSDL files. .NET Explorer uses this functionality to generate C# or VB.NET code for Web-service proxies and compile the code into temporary .NET assemblies that are displayed as **Web Service Proxies** in **.NET Explorer** > **Loaded Components** > **References**.

As .NET Explorer uses Microsoft .NET Framework to generate proxies, .NET Explorer shares the drawbacks and limitations of Microsoft .NET Framework. The most significant problem when generating proxies is that not all SOAP stack implementations produced by other vendors comply with the W3C standard. This can lead to problems when you attempt to load WSDL files. You can avoid these problems by manually editing the WSDL files so that they are recognized by Microsoft .NET Framework. To edit the WSDL files you must be familiar to WSDL. For additional information, refer to *http://www.w3.org*.

.NET Explorer shows a Web-service proxy class, derived from `System.Web.Services.Protocols.SoapHttpClientProtocol`, in the **References** menu tree below the **Web Service Proxies** tree node. Normally, when you write C# or VB.NET code, you must instantiate an instance of the proxy class and call methods on the proxy. .NET Explorer eliminates the need for this by automatically instantiating an instance of the proxy class. You cannot create an instance of a proxy class by calling the constructor. .NET Explorer treats Web services like static objects offering static methods.

If the methods of a Web service take complex objects as parameters, then the classes of those parameters are defined in the WSDL file and loaded by .NET Explorer. Such classes are not Web-service proxy classes. They are simple classes with members and are listed under the **Other Classes** tree node in the **Class** menu tree.

You can set several connection-related properties by double-clicking a proxy class in the menu tree and opening the connection wizard in the **Input Data Properties** pane. These properties are set to the corresponding properties of the *internal* proxy instance.

When you export a project to either Visual Studio .NET or directly to SilkPerformer as a .NET project, the base class of the Web-service proxy is replaced by `SilkPerformer.SPSoapHttpClientProtocol`. The reason for this is that by exchanging the base class, the SilkPerformer .NET Framework is able to generate more detailed timers for Web-service calls that are routed through the SilkPerformer Web engine. If you don't want this behavior you can export a project to Visual Studio .NET and either change the base class back manually or use the **Web Service** dialog box from the SilkPerformer menu and deselect the option for routing the proxy class.

You can now either load the WSDL file of the Web service you want to test or select a WSDL file from the list box.

Calling a Web Service

After you have successfully loaded a WSDL file you can explore the methods that the Web service offers by expanding the tree node of the Web-service proxy class.

By clicking one of the methods you will see the required input parameters and input header information for the Web-service call. You can customize your input data by either exchanging the default static values for the primitive types or by using global, local, or rendom variables. For additional information, refer to the *.NET Explorer Help* and the *SilkPerformer Help*.

If a method is called for the first time on a Web service, the internal instance of the proxy class is instantiated. There is a property on the proxy class that holds a cookie container. This property is initialized with a new cookie container so that it can call Web services that handle cookies.

.NET Explorer then sets all the defined values for the SOAP headers to the corresponding member fields of the proxy class. Then a parameter list with all the values that are defined for the input parameters is created. Using this list, the method on the Web-service proxy object is invoked.

Microsoft .NET Framework includes a hooking mechanism that allows .NET Explorer to capture traffic that is passed between the .NET Explorer client and the Web server. You can view the trafic in the **Show Traffic** dialog box after the method call. You can invoke the dialog box on each Web service and each .NET Remoting call.

This feature is also used to generate BDL Web scripts with a `WebPagePost` for each Web-service call captured in the traffic moving from the client to the server.

The returned values and SOAP header information are displayed when the method calls return successfully. When exceptions occur, the exception text is displayed in a message box. Currently you cannot add method calls to test scenarios that throw errors because .NET Explorer requires information about the returned values.

Final Steps

After calling a Web service you can store the returned values to variables and define verifications for those values.

Once you have finished defining your test scenario you can either remain in .NET Explorer, and use your test scenario for functional testing, or you can export the project to Visual Studio .NET or SilkPerformer to further customize the generated code and run regression tests.

You can only export to BDL Web projects when your test scenarios contain only Web-service calls, because only `WebPagePost` statements are generated for each Web-service call. If you have calls other than Web-service calls, for example calls to other .NET objects, those calls will not be included in Web scripts and therefore your exported scripts may not behave as defined in .NET Explorer, as some method calls will be missing.

Only export to BDL Web projects if you have only Web-service calls and if you only wish to test the SOAP stack of your server, as there is no .NET client SOAP stack involved when executing scripts in

SilkPerformer. The SilkPerformer Web engine posts only those SOAP envelopes that have been used in .NET Explorer.

If you also wish to test the .NET client side, you should export your project to a SilkPerformer .NET project. This type of project will compile generated .NET test code into a .NET assembly that can be called from a BDL script, which will also be generated by .NET Explorer.

If you wish to make further customizations to .NET code generated by .NET Explorer you can export your project to Visual Studio .NET. If you export your project you can alter generated test code and run a TryScript within Visual Studio .NET. If you are finished with customizations you can export the project to SilkPerformer and proceed with testing.

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. Microsoft Visual Studio offers wizards that allow you to specify the URLs of Web services. Microsoft Visual Studio can also create Web-service client proxies to invoke Web-service methods.

Creating a New SilkPerformer .NET Project

To create a new SilkPerformer .NET project, you can either create a new project of type **Web Services** > **.NET Explorer** in SilkPerformer or you can use one of the SilkPerformer .NET project wizards in Visual Studio .NET.

With both approaches you can choose one of the following three implementation languages:

- C#
- VB.NET
- C++

If you choose one of the .NET project wizards in Visual Studio .NET, the result is a new project with a template class that defines three methods, which are the init, main, and end transactions of your SilkPerformer virtual user.

To develop a .NET test driver in another language, create an empty project using the language of your choice and perform the following steps:

1. Add a reference to `perfdotnetfw.dll`.

   This DLL is located in the SilkPerformer installation directory.
2. Add a new class to your project.
3. Add the `VirtualUser` custom attribute to your class.
4. Add public member functions to your class to serve as your user transactions.
5. Add the Transaction attribute to the functions you have created and pass the corresponding transaction type.

   - init
   - main
   - end

Creating a Web Service Client Proxy

Visual Studio .NET has a wizard that generates a Web-service-client proxy that allows you to call Web-service methods.

You can start the wizard in **Project** > **Add Web Reference**.

1. To start the wizard, click **Project** > **Add Web Reference**.
2. In the corresponding text box, type the URL of your Web service and press **Enter**.
   For example, *http://demo.borland.com/BorlandSampleService/BorlandSampleService.asmx?WSDL*.
3. If the wizard can load the WSDL document from the URL, click **Add Reference**.
   The wizard generates a proxy class in a namespace, which is the reverse of the name of the Web server that hosts the service.

Explore projects to see which classes are generated. Each web service, and all complex data types used by the Web-service methods, are represented as classes. So in the example URL above, there is `Service1`, which is a Web service, and `User`, which is a complex parameter.

Instantiating a Client Proxy Object

You can declare a variable of a client proxy class as a public member of the .NET test driver to instantiate a client-proxy object. The variable should be instantiated either in the constructor or in the `init` transaction. The first part of the namespace where the class is generated, which is the default namespace, is the name of your project.

Example

If you have created a project with the name *DotNetProject* you would use the following variable declaration:

```
[VirtualUser("Vuser")]
public class Vuser
{
  public DotNetProject.com.borland.demo.Service1 mService;
  [Transaction(Etranstype.TRANSTYPE_INIT)]
  public void TInit()
  {
    mService = new DotNetProject.com.borland.demo.Service1();
  }
}
```

Calling a Web Service Method

All methods that are exposed by Web services are also available in proxy objects. The methods that are shared by proxy objects use the same names as their corresponding WSDLs. Web-service method calls should be placed in `main` transactions.

Example

```
[Transaction(Etranstype.TRANSTYPE_MAIN)]
public void TMain()
{
  string sReturn = mService.echoString("Test");
  Bdl.Print(sReturn);
}
```

To customize your Web-service calls from a generated BDL script, you must allow the exchange of data between BDL and .NET with usage of attributes or method parameters.

Example

```
[Transaction(Etranstype.TRANSTYPE_MAIN)]
[TestAttribute("EchoInput", "Test")]
public void TMain()
{
  string sReturn =
mService.echoString(Bdl.AttributeGet("EchoInput"));
  Bdl.Print(sReturn);
}
```

or

```
[Transaction(Etranstype.TRANSTYPE_MAIN)]
public void TMain(string sEcho)
{
  string sReturn = mService.echoString(sEcho);
  Bdl.Print(sReturn);
}
```

Routing Web-Service Traffic

The SilkPerformer .NET Framework can route Web traffic generated by .NET components through the SilkPerformer Web engine. This means that the Web engine executes the actual Web requests, enabling you to see exactly what is sent over the wire. You can also use features of the SilkPerformer Web engine, like modem simulation, IP multiplexing, network statistics, TrueLog, and others.

By default all network traffic is routed through the Web engine. You can switch the routing off and only enable it for specific Web-service client-proxy classes. To switch the routing on for specific Web-service client-proxy classes, you need to change the base class of the proxy classes from `SoapHttpClientProtocol` to `SilkPerformer.SPSoapHttpClientProtocol`. Changing the base class allows the SilkPerformer .NET Framework to generate more detailed statistical information for each Web-service call. We recommend that you enable this feature for all your Web-service proxy classes. You can enable this feature by using the **Web Service** dialog box in Microsoft Visual Studio, which is accessible through the SilkPerformer menu.

For each Web-service call a node is created in the TrueLog with the SOAP envelope that was passed to the Web service and returned to the client.

If detailed statistical information for Web-service calls is disabled, the .NET HTTP classes process all requests.

Exploring Results

When Web-service-traffic routing is enabled, a TrueLog node is logged for each Web-service call that is executed by the .NET test driver.

In the overview report of the Web-service method that is called, you will find statistical information.

External References

1. Session, Roger

SOAP. An overview of the Simple Object Access Protocol, March 2000

http://www.objectwatch.com/issue_25.htm

2. W3C

Simple Object Access Protocol (SOAP) 1.1, December 2000

http://www.w3.org/TR/SOAP/

3. UN/CEFANT, OASIS

Enabling Electronic Business with ebXML, December 2000

http://www.ebxml.org/white_papers/whitepaper.htm

4. Geyer, Carol

ebXML Integrates SOAP Into Messaging Services Specification, March 2001

http://www.ebxml.org/news/pr_20010222

5. Open Financial Exchange

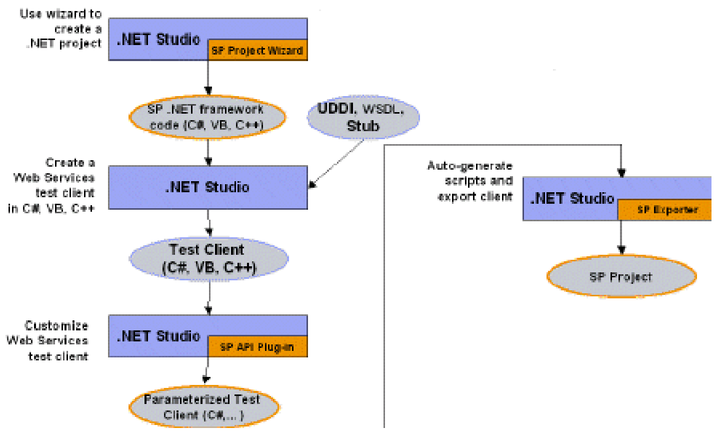
Open Financial Exchange Specification 2.0, April 2000

http://www.ofx.net/ofx/noreg.asp

Testing .NET Services

This section offers an overview of the SilkPerformer .NET Framework and the SilkPerformer .NET Add-On for Microsoft Visual Studio .NET. It also serves as an in-depth demonstration of how to execute .NET methods with BDL, how to write complete test drivers in .NET, and how to test drivers in Visual Studio .NET. It explains how to write test code in .NET and customize test code in BDL. Because SOAP (Web services) has become widely accepted, this chapter also explores how Web services can easily be tested using SilkPerformer and the .NET Framework.

Development Workflow


Use wizard to create a .NET project
.NET Studio
SP Project Wizard
SP .NET framework code (C#, VB, C++)
UDDI, WSDL, Stub
Create a Web Services test client in C#, VB, C++
.NET Studio
Auto-generate scripts and export client
.NET Studio
SP Exporter
SP Project
Test Client (C#, VB, C++)
Customize Web Services test client
.NET Studio
SP API Plug-in
Parameterized Test Client (C#,...)

Writing a .NET Test Driver

This section describes how you can write a .NET test driver.

Creating a .NET Project

SilkPerformer ships with an add-on and a project wizard for Microsoft Visual Studio .NET. Microsoft Visual Studio .NET must be installed on your machine prior to the installation of the add-on. The project wizard creates a .NET project in one of the supported .NET languages, which are C#, VB.NET, or Managed C++, and generates a sample test driver into which you only have to add your test code into the test methods. You can insert calls to .NET components, Web services and BDL functions, like `MeasureStart`, `MeasureStop`, and others.

This wizard can be invoked either by creating a new .NET project with SilkPerformer or by creating a new SilkPerformer project in Visual Studio .NET. In either case you start with a new SilkPerformer .NET project in Microsoft Visual Studio .NET with the sample test code file open.

`PerfDotNetFW.dll` is a .NET assembly that ships with SilkPerformer. It implements all classes, custom attributes, and enums that can be used to define meta information for automatic SilkPerformer BDL script generation and calls BDL functions from within the .NET test driver. `PerfDotNetFW.dll` is discussed in great detail later in this chapter. The assembly is automatically referenced by the generated project.

Defining a Virtual User in .NET

A virtual user in .NET is a public .NET class with the `SilkPerformer.VirtualUser` attribute applied. This attribute tells the add-on to generate a virtual user definition in the BDL script.

The `VirtualUser` attribute has one parameter - the name of the virtual user that is to be generated in the BDL script when running a try script.

You can have multiple `VirtualUser` classes in your .NET assembly but the names of the virtual users must be unique.

Defining a Virtual User in .NET

C# Code

```
[VirtualUser("Vuser1")]
public class MyTestUser1
{
  ...
}
[VirtualUser("Vuser2")]
public class MyTestUser2
{
  ...
}
```

BDL Script

```
dcluser
  user
    Vuser1
    ...
  user
    Vuser2
```

Defining a Transaction in .NET

A transaction in .NET is a public (non virtual) .NET method of your virtual user class that has the `SilkPerformer.Transaction` attribute applied to it. There are three types of transactions, `init`, `main`, and `end`. There can only be one `init` and one `end` transaction for each virtual user class, but there can be multiple `main` transaction methods.

The transaction type is passed as the first parameter. `ETransactionType` is an enum that defines the possible types.

`SilkPerformer.ETransactionType`:

- TRANSTYPE_INIT
- TRANSTYPE_MAIN
- TRANSTYPE_END

Transactions of type `main` have an optional second parameter that defines the number of transaction calls - the default value is 1.

Example

C# Code

```
[VirtualUser("Vuser1")]
public class MyTestUser1
{
  [Transaction(

Etranstype.TRANSTYPE_INI
T)]
  public void TInit()
```

BDL Script

```
dcluser
  user
    Vuser1
  transactions
    TInit : begin;
    TMain : 1;
    TMain2 : 5;
    TEnd : end;
```

C# Code

```
  {
  }
  [Transaction(

Etranstype.TRANSTYPE_MAI
N)]
  public void TMain()
  {
  }
  [Transaction(

Etranstype.TRANSTYPE_MAI
N, 5)]
  public void TMain2()
  {
  }
  [Transaction(

Etranstype.TRANSTYPE_END
)]
  public void TEnd()
  {
  }
}
```

BDL Script

```

var hVuser1 : number;

dcltrans
  transaction TInit
  begin
    hVuser1:=

DotNetLoadObject("..","M
yTestUser1");

DotNetCallMethod(hVuser1
,"TInit");
  end;

  transaction TMain
  begin

DotNetCallMethod(hVuser1
,"TMain");
  end;
  transaction TMain2
  begin

DotNetCallMethod(hVuser1
,"TMain2");
  end;

  transaction TEnd
  begin

DotNetCallMethod(hVuser1
,"TEnd");

DotNetFreeObject(hVuser1
);
  end;
```

The add-on script generator scripts an `init` and an `end` transaction even if there are no corresponding .NET methods. These are used to load the .NET object in the `init` transaction and free it in the `end` transaction.

As you can see from the sample above, the transactions contain a `DotNetCallMethod` to call the .NET test driver method.

Defining Additional Test Methods

A test method in .NET is a public (non virtual) method that has the `SilkPerformer.TestMethod` attribute applied to it. Each call to a test method is scripted as a `DotNetCallMethod` function in the current transaction. It's possible to have multiple test methods in a single transaction.

Defining a Test Method

C# Code

```
[VirtualUser("Vuser1")]
[Transaction(Etranstype.
```

BDL Script

```
dcluser
  transaction TMain
```

C# Code	BDL Script

`TRANSTYPE_MAIN)]`
`  public void TMain()`
`  {`
`  }`
 	
`  [TestMethod]`
`  public void Method1()`
`  {`
`  }`
 	
`  [TestMethod]`
`  public void Method2()`
`  {`
`  }`
	`  begin`
	
	`DotNetCallMethod(hVuser1`
	`,"TMain");`
	
	`DotNetCallMethod(hVuser1`
	`,"Method1");`
	
	`DotNetCallMethod(hVuser1`
	`,"Method2");`
	`  end;`

The two test methods, `Method1` and `Method2`, will be called in the current transaction. The current transaction is the transaction method that is declared previous to these test methods in the .NET code.

Passing Data Between BDL and .NET

There are two means of exchanging values between the generated BDL script and the .NET test driver:

- Attributes
- Parameters

Declaring Attributes

A method can have multiple `SilkPerformer.TestAttribute` attributes applied to it. Attributes are not scripted, but created as project attributes. This makes it easier for engineers to customize BDL scripts to change values for different attributes, as all attributes can be found in the project attributes dialog box. Therefore the .NET test driver can access attributes with `Bdl.AttributeGet`.

The `TestAttribute` attribute has two parameters. The first parameter is the name of the attribute and the second is the default value of the project attribute.

A comment is scripted prior to the function call to indicate what specific attributes the function call requires.

```
// Requires attribute "Attrib1" with the default value: "Value1"
DotNetCallMethod(hVuser1, "TMain");
```

Declaring Attributes

C# Code	BDL Script

`[Transaction(Etranstype.`
`TRANSTYPE_MAIN)]`
`[TestAttribute("Attrib1"`
`,"Value1")]`
`public void TMain()`
`{`
`  string s =`
`Bdl.AttributeGet("Attrib`
`1");`
`}`
	`dcltrans`
	`  transaction TMain`
	`  begin`
	`    // Requires`
	`attribute "Attrib1"`
	`    // with the default`
	`value: "Value1"`
	
	`DotNetCallMethod(hVuser1`
	`,"TMain");`
	`  end;`

By customizing the attribute value in the project attributes dialog box, you can customize the runtime behavior of the .NET test driver without changing the .NET code.

Defining Parameters

If a method has input parameters and a return value, the Code Generation Engine appropriately creates the BDL calls for passing those parameters in and getting the return parameter. Therefore a method can have any combination of parameters and return values that can be accessed by the DotNet API functions (`DotNetGetXX`).

Defining Parameters

C# Code

```
[Transaction(Etranstype.
TRANSTYPE_MAIN)]
public string
TMain(string s, int n)
{
  return s +
n.ToString();
}
```

BDL Script

```
dcltrans
  transaction Tmain
  var
    sReturn : string;
  begin

DotNetSetString(hVuser1,
"stringvalue");

DotNetSetInt(hVuser1,
123);

DotNetCallMethod(hVuser1
,"TMain");

DotNetGetString(hVuser1,
 sReturn,
sizeof(sReturn));
  end;
```

As you can see in the above example, `DotNetSetXX` functions are scripted for each method parameter. The values are default values that are suggested by the Add-On. The Code Generation Engine also scripts a `DotNetGetXX` function for the return value of the method. The following .NET parameter types are supported:

- String
- Byte
- SByte
- UIntPtr
- UInt16
- UInt32
- UInt64
- Int16
- Int32
- Int64
- IntPtr
- Decimal
- Double
- Single
- Boolean
- Object

Return Parameter

By default the return parameter is stored in a variable with the name `xReturn`, or `sReturn` for strings. You can give the variable a meaningful name by applying the `SilkPerformer.BdlParameter` attribute to your return type and passing the variable name as the first parameter (`sConcatParam` in the following example).

Example for Return Parameter

C# Code	BDL Script
`[Transaction(Etranstype.TRANSTYPE_MAIN)] [return:BdlParameter("sConcatParam")] public string TMain(string s, int n) { return s + n.ToString(); }`	`dcltrans transaction Tmain var sConcatParam : string; begin DotNetSetString(hVuser1, "stringvalue"); DotNetSetInt(hVuser1, 123); DotNetCallMethod(hVuser1,"TMain"); DotNetGetString(hVuser1, sConcatParam, sizeof(sConcatParam)); end;`

The ability to define a different name for the return variable is necessary for the Code Generation Engine to generate BDL code that passes values between function calls.

Calling BDL Functions from .NET

Most of the functions exposed by the `kernel.bdh` of SilkPerformer are implemented in `PerfDotNetFW.DLL`, which is a .NET assembly that comes with SilkPerformer. The methods are static methods in the `SilkPerformer.Bdl` class. As SilkPerformer is an imported namespace you can call the methods with `Bdl.<MethodName>`. `PerfDotNetFW.dll` is referenced by default when you create a SilkPerformer .NET project.

Primary BDL Functions

The following table lists the primary functions that are implemented by the `SilkPerformer.Bdl` class. For a complete list of all functions, open the class in Microsoft Visual Studio .NET.

Function	Description
• `AttributeSet` • `AttributeGet`	Setting and getting attribute values.
• `MeasureStart` • `MeasureStop` • `MeasurePause`	Measure functions.

Function	Description
• `MeasureResume` • `MeasureInc` • `MeasureIncFloat` • `MeasureGet` • `MeasureSetBound` • `MeasureTimeseries` • `MeasureOnOff`	
• `GetUser` • `GetUserId` • `GetUserIdOnAgent` • `GetUserGroup` • `GetProfile` • `GetAgentId` • `GetRuntimes` • `GetAgent` • `GetController` • `GetProject` • `GetLoadTest` • `GetMemUsage` • `GetBdfFileName` • `GetBuildNo`	Information about the current test.
`Print`	Printing messages to the virtual user output.
`RepMessage`	Write a message to the following files: • .ERR • .LOG • .RPT
• `WriteErr` • `WriteLog` • `WriteData` • `WriteWrt` • `Write` • `Writeln`	Write data to output files.
`GetDataFilePath`	Gets the absolute path to a file in the data files section.
`RndUniN`, and others	Random functions.

Constant Values

Some functions in the BDL take constant values as parameters. These constant values are defined as public enums in the SilkPerformer.Bdl namespace. The following list lists some of the defined enums:

- `PrintDisplay` (*OPT_DISPLAY_ERRORS*, *OPT_DISPLAY_TRANSACTIONS*, ...)
- `PrintColor` (*TEXT_GRAY*, *TEXT_BLACK*, ...)
- `Severity` (*SEVERITY_SUCCESS*, *SEVERITY_INFORMATIONAL*, ...)

- `MeasureKind` (*MEASURE_KIND_SUM*, *MEASURE_KIND_COUNT*, ...)
- `MeasureUsage` (*MEASURE_USAGE_TIMER*, ...)
- `MeasureClass` (*MEASURE_IIOP*, *MEASURE_TIMER*, ...)
- `ThinkTimeOption` (*OPT_THINKTIME_RANDOMWAIT*, ...)

Example

```
[Transaction(EtransactionType.TRANSTYPE_MAIN)]
[TestAttribute("Attribute1", "TestValue")]
public void Tmain()
{
  string sValue = Bdl.AttributeGet("Attribute1");
  Bdl.MeasureStart("My Testmeasure");
  string sReturn = SomeMethod(sValue);
  Bdl.Print(sReturn);
  Bdl.MeasureStop("My Testmeasure");
}
```

Random Variables

You can define random variables for your test user that can be accessed within .NET test code. There is a wizard in Visual Studio .NET that helps you define these random variables. The wizard has the same look & feel as the random variable wizard of SilkPerformer.

Depending on the type of random variable you choose, the wizard adds the appropriate random custom attribute to the virtual user class. You can use the following random custom attributes:

- `RndBin`
- `RndExpF`
- `RndFile`
- `RndInd`
- `RndPerN`
- `RndSno`
- `RndStr`
- `RndStream`
- `RndUniF`
- `RndUniN`

The first parameter is the name of the random variable. This is followed by the parameters that must be defined in BDL to define the random variable. For additional information, refer to the *Benchmark Description Language (BDL) Reference* or check Visual Studio .NET's Code Completion.

You can access random variables with the following three new functions that are declared in the BDL class as static methods:

- `GetRandomFloat`
- `GetRandomNumber`
- `GetRandomString`

Those methods take the name of the random variable as an input parameter and then return random values.

Exception Handling

The goal in exception handling is to catch all exceptions in your test code. That is why after creating a project, the default code has try/catch blocks in each test method.

The SilkPerformer .NET Framework throws one exception, the `SilkPerformer.StopException`. The framework throws this exception when a run is aborted or stopped by the user. You can utilize this exception for clean up.

Other exceptions are handled normally. For detailed exception information in your TrueLogs, use `Bdl.LogException` to log exceptions, including message and stack trace, to TrueLog. This is particularly useful when running tests while TrueLog On Error is activate, because you can see which exceptions are thrown and you get a complete stack trace.

Debugging

Running TryScripts in debugging mode is not directly supported from within Visual Studio .NET. We recommend an approach to enforce debugging through a work-around.

Place a `System.Diagnostics.Debug.Assert` (`false`) statement in the constructor of your test driver or at any other position in your code. Compile your code and initiate a TryScript run from SilkPerformer. You can also initiate a TryScript run from Visual Studio .NET, however we do not recommend this approach, as a new instance of Microsoft Visual Studio .NET is required for debugging and the new instance will not obtain information regarding it's impact on the instance that is running the TryScript run.

If you subsequently initiate a TryScript run from SilkPerformer, a debug assertion dialog box opens, allowing you to debug the code.

Configuration Files

Microsoft .NET Framework allows to store a configuration file in the directory of the .NET executable or ASP.NET application that contains runtime-specific configurations. These configurations are loaded when the application is launched. For .NET executables the configuration file needs to be named with both the `.exe` file extension and the `.config` file extension. For example `myprogramm.exe.config`.

When running a test, the executable that hosts your .NET test driver code is `perfrun.exe`. Therefore it's possible to have a `perfrun.exe.config` that contains configuration settings that are loaded at startup. However this approach is not possible because SilkPerformer generates the `perfrun.exe.config` file automatically before starting a test and would therefore overwrite such a configuration file. The `perfrun.exe.config` file contains settings based on the profile settings of the project profile.

With SilkPerformer, you can have an `app.config` file in your project directory to specify your own configuration file settings. SilkPerformer checks for this configuration file and merges the content into an automatically generated `perfrun.exe.config` file.

A `perfrun.exe.config` file has the following structure:

```
<configuration>
  <system.net>
    ...
  </system.net>
  <runtime>
    ...
  </runtime>
</configuration>
```

For backward compatiblity, when SilkPerformer locates an `app.config` file, the content is added after the `runtime` tag. That means that your `app.config` file can contain any configuration nodes that are allowed below the root configuration node, except the *system.net* and runtime nodes, because SilkPerformer generates those nodes.

With SilkPerformer you can provide a fully-configured `app.config` file with all possible configuration sections. SilkPerformer adds the necessary entries for web-traffic routing in the generated `perfrun.exe.config` file.

Configuring .NET Remoting Components

If you wish to configure .NET Remoting components, you need an `app.config` file such as the following:

```
<system.runtime.remoting>
  <application>
    <channels>
      <channel ref="http" port="2000" />
    </channels>
    <client url="http://remoteserver:2000">
      <activated type="RemoteDll.RemoteClass1, RemoteDll" />
      <activated type="RemoteDll.RemoteClass2, RemoteDll" />
    </client>
  </application>
</system.runtime.remoting>
```

BDL Code Generation Engine

The SilkPerformer .NET Add-On has a BDL Code Generation Engine that generates a BDL script based on the meta data information of the compiled .NET assembly (.NET test driver). The engine is invoked when a TryScript run is started in Microsoft Visual Studio .NET. The compiled .NET assembly is scanned for classes that have the `VirtualUser` attribute applied. Those classes are then scanned for methods that have either a `Transaction` or `TestMethod` attribute applied to them. The sequence of methods is important because calls to test methods, which are methods with the `TestMethod` attribute, are scripted in the transaction, which is a method with the `Transaction` attribute, that is declared prior to the test method.

Virtual User

The Virtual User name is used as a prefix for all transactions and methods scripted in BDL. This is necessary to prevent method-name duplication, since the same method name may exist in two different .NET classes.

The engine checks for duplicate Virtual User names. If the assembly contains more than one Virtual User class, the names passed as parameters to the `VirtualUser` attribute must be unique. If there are duplicate names, an error is thrown and shown in the task list.

Note: You should avoid using multiple methods with the same name in one .NET class. .NET allows using multiple methods with the same name if the methods have different parameters, but the Code Generation Engine of SilkPerformer does not support this feature.

Random Variables - Virtual User Classes

The SilkPerformer code-generation engine also checks the virtual user class of all defined random-variable custom attributes. For each random variable, the corresponding random-variable definition is declared in the BDL script.

Currently these variables are of no real use, neither on the BDL side or the .NET side. If you use one of the `Bdl.GetRandom` functions, the definition of the random variable is read from the metadata information of the .NET code, not from the BDL file. That means that if you change the random variable definition in BDL it will not have any effect on the .NET code as it still must have the definition from the custom attributes. This behavior will be adjusted in future versions so that you will be able to change settings in the BDL script and have the executing .NET code receive those random values based on the BDL settings.

Transactions - Virtual User Classes

The Code Generation Engine checks all methods that have the transaction attribute applied. There are three types of transactions, `init`, `main`, and `end`. A Virtual User class can only have one `init` and one `end` transaction. Even if there is no `init` or `end` transaction within the .NET code, transactions are still scripted because they are required for loading and freeing the .NET objects.

The first call in the `init` transaction in BDL is a `DotNetLoadObject` method call. After this call the actual call to the .NET method is made. Thereafter all methods with the `TestMethod` attribute that are defined between this transaction method and the next are called.

The last call in the `end` transaction in BDL is a `DotNetFreeObject` method call.

The `main` transactions call the actual transaction method in .NET and then the appropriate test method calls.

Test Methods

When the Code Generation Engine scans the .NET assembly and finds a method with a `TestMethod` attribute, the engine scripts a call to the method in the current transaction. The current transaction is the transaction method that was declared before the test method. As a result, declaring a test method without a transaction method results in an error. *Declaration* means that the transaction method has been declared previously in the code.

Test Method

C# Code

```
[Transaction(Etranstype.
TRANSTYPE_MAIN)]
public void TMain1()
{
}

[TestMethod]
public void TestMeth1()
{
}

[Transaction(Etranstype.
TRANSTYPE_MAIN)]
public void TMain2()
{
}

[TestMethod]
public void TestMeth2()
{
}

[TestMethod]
public void TestMeth3()
{
}
```

BDL Script

```
dcltrans
  transaction TMain1
  begin

DotNetCallMethod(hVuser1
, "TMain1");

DotNetCallMethod(hVuser1
, "TestMeth1");
  end;
  transaction TMain2
  begin

DotNetCallMethod(hVuser1
, "TMain2");

DotNetCallMethod(hVuser1
, "TestMeth2");

DotNetCallMethod(hVuser1
, "TestMeth3");
  end;
```

Erroneous Test Method Declaration

The following test method declaration would cause an error with the Code Generation Engine as there is no current transaction for the `TestMeth1` method:

```
[TestMethod]
public void TestMeth1()
{}

[Transaction(Etranstype.TRANSTYPE_MAIN)]
public void TMain1()
{}
```

Test Attributes

When the Code Generation Engine processes a transaction or test method it checks whether the method has `TestAttribute` attributes applied to it. A project attribute is generated for each test attribute. The name and default values are used to generate project attributes with the same names and default values. To make it easier for engineers who customize BDL script, comments are scripted before the `DotNetCallMethod`.

```
// Requires attribute "Attr1" with the default value: "Value1"
DotNetCallMethod(hVuser1, "TMain1");
// Requires attribute "Attr2" with the default value: "Value2"
// Requires attribute "Attr3" with the default value: "Value3"
DotNetCallMethod(hVuser1, "TestMeth1");
```

Test Attributes

C# Code	BDL Script

`[Transaction(Etranstype.TRANSTYPE_MAIN)]`
`[TestAttribute("Attr", "Value1")]`
`public void TMain1()`
`{`
`}`
 	
`[TestMethod]`
`[TestAttribute("Attr2", "Value2")]`
`[TestAttribute("Attr3", "Value3")]`
`public void TestMeth1()`
`{`
`}`
	`dcltrans`
	`  transaction TMain1`
	`  begin`
	`    // Requires attribute "Attr1"`
	`    // with the default value: "Value1"`
	
	`DotNetCallMethod(hVuser1, "TMain1");`
	`    // Requires attribute "Attr2"`
	`    // with the default value: "Value2"`
	`    // Requires attribute "Attr3"`
	`    // with the default value: "Value3"`
	
	`DotNetCallMethod(hVuser1, "TestMeth1");`
	`  end;`

Methods with Parameters

If a method, whether it is a transaction or a test method, has input parameters or a return value, the engine scripts `DotNetSetXX` functions to pass the input parameters and a `DotNetGetXX` function to retrieve the return value.

The following `DotNetGetXX` and `DotNetSetXX` functions are available:

- `DotNetSetString`
- `DotNetSetInt`
- `DotNetSetFloat`
- `DotNetSetBoolean`
- `DotNetSetObject`
- `DotNetGetString`
- `DotNetGetInt`
- `DotNetGetFloat`
- `DotNetGetBoolean`
- `DotNetGetObject`

Therefore you can exchange strings, integers, floats, Booleans, and objects. Objects are object handles to other .NET objects.

The Code Generation Engine scripts a `DotNetSetXX` function for each parameter in the same sequence as the parameter definition. If there is a return value, it scripts the corresponding `DotNetGetXX` function.

The Code Generation Engine creates appropriate values for input parameters such as `123` for the int in the example below. SilkPerformer does not support arrays.

Methods with Parameters

C# Code	BDL Script
`[Transaction(Etranstype.TRANSTYPE_MAIN)] public object TMain1(string s, int n) { }`	`dcltrans transaction TMain1 var hReturn : number; begin DotNetSetString("stringvalue"); DotNetSetInt(123); DotNetCallMethod(hVuser1, "TMain1"); DotNetGetObject(hReturn); end;`

```
[Transaction(Etranstype.
TRANSTYPE_MAIN)]
public object
TMain1(string s, int n)
{
}
```

```
dcltrans
  transaction TMain1
  var
    hReturn : number;
  begin

DotNetSetString("stringv
alue");
    DotNetSetInt(123);

DotNetCallMethod(hVuser1
, "TMain1");

DotNetGetObject(hReturn)
;
  end;
```

The following .NET data types are supported:

- `String`
- `Byte`
- `SByte`
- `UIntPtr`
- `UInt16`
- `UInt32`
- `UInt64`
- `Int16`
- `Int32`
- `Int64`
- `IntPtr`
- `Decimal`

- `Double`
- `Single`
- `Boolean`
- `Object`

BDL Parameters

If a method, whether it is a transaction or a test method, has a return parameter, the Code Generation Engine stores the value by default in a variable with the name *xResult*, where x depends on the return type. To change this behavior, apply a `BdlParameter` attribute to the return type of the method and pass the variable name as the first parameter.

BDL Parameters

C# Code	BDL Script

`[Transaction(Etranstype.`
`TRANSTYPE_MAIN)]`
`[return:BdlParameter("sC`
`oncatParam")]`
`public string`
`TMain(string s, int n)`
`{`
`  return s +`
`n.ToString();`
`}`
	`dcltrans`
	`  transaction Tmain`
	`  var`
	`    sConcatParam :`
	`string;`
	`  begin`
	
	`DotNetSetString(hVuser1,`
	` "stringvalue");`
	
	`DotNetSetInt(hVuser1,`
	`123);`
	
	`DotNetCallMethod(hVuser1`
	`, "TMain");`
	
	`DotNetGetString(hVuser1,`
	` sConcatParam,`
	
	`sizeof(sConcatParam));`
	`  end;`

Intelligent Parameter Passing

The Code Generation Engine checks methods for their return values and input parameters. If a method has an input parameter that has the same name as a previously returned value, the return value will be passed to the function. Normally return parameters do not have names assigned to them - but you can assign a name by applying a `BdlParameter` attribute.

As a result, you can pass values between method calls by giving the parameters and return values the same names. When you declare the parameters, ensure that you do not assign the same name to different parameter types. In SilkPerformer the Code Generation Engine does not check if the return value and input parameter types are the same.

Intelligent Parameter Passing

C# Code	BDL Script

`[Transaction(Etranstype.`
`TRANSTYPE_MAIN)]`
	`dcltrans`
	`  transaction Tmain`

C# Code	BDL Script
[return:BdlParameter("sN	var sName : string;
ame")]	begin
public string TMain()	
{	DotNetCallMethod(hVuser1
return "Andi";	, "TMain");
}	
[TestMethod]	DotNetGetString(hVuser1,
public void	sName, sizeof(sName));
Hello(string sName)	
{	DotNetSetString(hVuser1,
Bdl.Print("Hello " +	sName);
sName);	
}	DotNetCallMethod(hVuser1
	, "Hello");
	end;

Options

This section describes the options that you can set in the **Options** dialog box of the Visual Studio .NET add-on. These options change the BDL code generation.

Generating Timers for DotNetCallMethod

If this option is enabled, the engine scripts a `MeasureStart` and `MeasureStop` for each `DotNetCallMethod` that uses the name of the method. This gives you the time taken for the method to execute and the information becomes available in the overview report.

Generating Timers for DotNetCallMethod

C# Code	BDL Script
[Transaction(Etranstype.	dcltrans
TRANSTYPE_MAIN)]	transaction Tmain
public void TMain()	begin
{	
}	MeasureStart("TMain");
[TestMethod]	DotNetCallMethod(hVuser1
public void Test1()	, "TMain");
{	
}	MeasureStop("TMain");
	MeasureStart("Test1");
	DotNetCallMethod(hVuser1
	, "Test1");
	MeasureStop("Test1");
	end;

Generating BDL Functions for .NET Method Calls

If this option is enabled, the engine scripts a BDL function for each call to a .NET method. The transaction calls the generated function. This makes the transactions shorter and easy to read. Input parameters to the .NET method become input parameters for the BDL function. If the .NET method returns a value, the value will be the return value of the function.

BDL Functions for .NET Method Calls

C# Code	BDL Script

[Transaction(Etranstype.
TRANSTYPE_MAIN)]
public void TMain()
{
}
 	
[TestMethod]
public string Test1(int
nParam)
{
}
	dclfunc
	  function
	Vuser_Tmain(hObject:numb
	er)
	  begin
	
	DotNetCallMethod(hObject
	, "TMain");
	  end;
	
	  function
	Vuser_Test1(hObject:numb
	er;nParam:number):
	string;
	    var sReturn :
	string;
	  begin
	
	DotNetSetInt(hObject,
	nParam);
	
	DotNetCallMethod(hObject
	, "TMain");
	
	DotNetGetString(hObject,
	 sReturn,
	sizeof(sReturn));
	    Vuser_Test1 :=
	sReturn;
	  end;
	  dcltrans
	  transaction Tmain
	    var sReturn :
	string;
	  begin
	    Vuser_Tmain(hVuser);
	    SReturn :=
	Vuser_Test1(hVuser,
	"stringparam");
	  end;

Generating BDH for .NET Method Calls

If this option is enabled, the engine scripts a BDL function for each .NET method and stores them in a BDH file in the main BDF file. This option implies the **Generating BDL Functions for .NET Method Calls** option.

Generating Project Attributes

When this option is enabled (by default), project attributes are created for each `TestAttribute` custom attribute on a method and a comment is scripted before the `DotNetCallMethod` call to help engineers determine which attributes are required by which methods.

If the **Generating Project Attributes** option is disabled, you need to script `AttributeSetString` calls before the call to define the attributes and default values.

We recommend that you enable the option because you can then manage all your attributes using the project attributes dialog and you do not need to browse through the script.

Testing Your .NET Test Driver

You can test the .NET test driver within Visual Studio .NET without even opening SilkPerformer. The add-on can run a Try Script, making the results and output visible in Microsoft Visual Studio .NET. This feature enables .NET developers to concentrate on developing their .NET test drivers. They do not need SilkPerformer or BDL skills because script generation is done by the add-on. QA departments can take the final versions of .NET test drivers and customize the generated scripts to their needs.

Preparations

If you have created a .NET test driver, and you do not need any special data files or profile changes, you are ready to execute a TryScript run to test your .NET test driver.

Data Files

If you access any files in your .NET methods you should add those files to the data files section of the SilkPerformer project. You can do this through the **Add Dependencies** dialog box of the SilkPerformer menu. If you are running a test on a remote agent, the files will be copied to the data directory of the agent. To ensure you have the right file path to your files, use the `Bdl.GetDataFilePath` method. This method returns the path to each data file using the filenames as parameters.

Adding the `C:\myfiles\file1.txt` File to the Data Files

```
[Transaction(Etranstype.TRANSTYPE_MAIN)]
public void TMain()
{
  string sFilename = Bdl.GetDataFilePath("file1.txt");
  System.IO.FileStream fs = System.IO.File.Open(sFilename,
  System.IO.FileMode.Open);
  ...
  fs.Close();
}
```

The .NET test driver DLL is automatically added to the data files of the project so that the DLL is available to remote agents.

Profile and System Settings

You can edit your profile and system settings within Microsoft Visual Studio .NET using the familiar SilkPerformer dialog boxes. Open the dialog boxes through the **Profile Settings** and **System Settings** menu entries of the SilkPerformer menu.

The following profile settings are specific to .NET:

- **Traffic redirection**

Option	Description
Route HTTP .NET through SilkPerformer web engine	If this option is enabled, all HTTP/HTTPS network traffic that is generated by .NET components is routed over the SilkPerformer Web engine. This allows you to make use of Web engine features such as modem simulation, IP multiplexing, and others. This also creates TrueLog nodes for each Web request, along with statistical information about the traffic.

Option	Description
Routed Web Service proxy classes	This read-only class is filled by the .NET add-on with the Web service proxy classes you are using in your .NET test driver.

- **Application domain setting**

Option	Description
One application domain for each virtual user container process	If this option is checked, there is only one application domain for all virtual users in the container process. This option improves performance because there is less overhead for the .NET Common Language Runtime to administrate multiple application domains for each process. The disadvantage is that all objects that are loaded from all virtual sers in the container process share a single application domain and therefore may conflict with one another in terms of, for example, global variables/resources.
One application domain for each virtual user	If this option is checked, each virtual user has its own application domain. The objects loaded by each virtual user are isolated in their application domains from other objects in other application domains. The disadvantage of this is that there is additional overhead for the .NET Common Language Runtime to administer multiple application domains in one process - therefore there is some performance loss.

Web Settings

You can change the Web settings in the **Web Settings** dialog box from the SilkPerformer menu.

In this dialog box you can set the option **Route HTTP .NET through SilkPerformer web engine.**

If you are testing Web services through a generated Web-service proxy class, you can route the generated network traffic over the SilkPerformer Web engine to get information about sent and received data and use features such as modem simulation.

In the checklist of the dialog box you find all the Web Service proxy classes of your project. You can check or uncheck the proxy classes that should be routed over the Web engine. When you check **Route HTTP .NET through SilkPerformer Web engine**, all proxy classes are routed since all network traffic is routed. Alternately, you can route individual Web-service calls by selecting them from the list.

The list of Web-service proxy classes is written to your project file and you can view the list in the .NET options of the profile settings dialog box.

Testing Options

Select the **Options** dialog box from the SilkPerformer menu. The dialog box includes options that affect output during tests and behavior of the Code Generation Engine.

The options that are relevant for running tests are:

- **TrueLog Explorer**

Option	Description
Automatically Start when running a TryScript	If this option is checked, TrueLog Explorer will launch automatically when a TryScript is run.

- **Virtual User Output**

  These options define what output will be printed to the **Virtual User Output** window during TryScript runs. For additional information on the options, refer to *SilkPerformer Help.*

Try Script Runs

You can launch a Try Script run from the SilkPerformer menu or by pressing **F8**. This is only possible if you did not assign a different command to F8 when the add-on was installed.

Steps Performed by the Add-On before Try Script Runs

The .NET add-on performs the following steps before it begins with the execution of a Try Script run:

1. The add-on compiles the .NET Assembly.
   If the compilation fails the add-on does not execute the Try Script.
2. The Code Generation Engine generates the BDF/BDH files.

   The engine checks if it has to overwrite old BDF/BDH files. Old files are overwritten only if there has been one of the following changes made to the structure of the .NET test driver:

   - Changes to `VirtualUser`, `Transaction`, `TestMethod`, `TestAttribute`, or `BdlParameter` attributes.
   - Changes to the sequence of a method definition.
   - Changes to the parameter definition of the methods.
   - Changes to code generation options.

   If significant changes have been made, the engine checks whether the BDF has been changed manually since it was last generated. If it has been changed, you must specify whether the file can be overwritten. If you select **No**, the Try Script is not executed.
3. The add-on prompts you to execute the `VirtualUser`. If there is more than one `VirtualUser` class in the .NET test driver, you must choose which virtual user is to be executed.
4. A temporary project file is generated and configured to run the Try Script.
5. The **Virtual User Output** window is activated.
6. The test begins.
7. If the **Automatically Start TrueLog Explorer** option setting has been enabled, TrueLog Explorer will launch with the TrueLog of the active Try Script.
8. All virtual user output, contingent on options, is printed to the **Virtual User Output** window.

Exploring Results

When Web-service-traffic routing is enabled, a TrueLog node is logged for each Web-service call that is executed by the .NET test driver.

In the overview report of the Web-service method that is called, you will find statistical information.

Running Tests in SilkPerformer

SilkPerformerOnce you have completed the implementation of the .NET test driver, you can run tests from within SilkPerformer. To do this, open SilkPerformer from the SilkPerformer menu. The project file is opened in SilkPerformer with all the generated scripts and data files.

You can customize the .NET test driver by changing the input values of the .NET functions in the BDL script. The separation of .NET code and BDL code is a great benefit for companies with testing departments in which not all employees are familiar with .NET. One employee with .NET skills can implement .NET test drivers and pass a generated SilkPerformer project along to other employees who are more familiar with SilkPerformer. Those employees can then customize the BDL script by changing input parameters and configuring real tests.

Running Tests on Remote Agents

If you want to run a .NET test on remote agents you must make sure that the Microsoft .NET Framework is installed on each of the agents. You can download the Microsoft .NET Framework from *http://msdn.microsoft.com*.

Testing Web Services With Microsoft Visual Studio

The SilkPerformer .NET Framework and .NET add-on allow easy access to Web services from within .NET. Microsoft Visual Studio .NET has wizards that allow you to specify the URLs of Web services. It can also create Web-service client proxies to invoke the methods of Web services.

Creating a Web Service Client Proxy

Visual Studio .NET has a wizard that generates a Web-service-client proxy that allows you to call Web-service methods.

You can start the wizard in **Project** > **Add Web Reference**.

1. To start the wizard, click **Project** > **Add Web Reference**.
2. In the corresponding text box, type the URL of your Web service and press **Enter**.

   For example, *http://demo.borland.com/BorlandSampleService/BorlandSampleService.asmx?WSDL*.
3. If the wizard can load the WSDL document from the URL, click **Add Reference**.
   The wizard generates a proxy class in a namespace, which is the reverse of the name of the Web server that hosts the service.

Explore projects to see which classes are generated. Each web service, and all complex data types used by the Web-service methods, are represented as classes. So in the example URL above, there is `Service1`, which is a Web service, and `User`, which is a complex parameter.

Instantiating a Client Proxy Object

You can declare a variable of a client proxy class as a public member of the .NET test driver to instantiate a client-proxy object. The variable should be instantiated either in the constructor or in the `init` transaction. The first part of the namespace where the class is generated, which is the default namespace, is the name of your project.

Example

If you have created a project with the name *DotNetProject* you would use the following variable declaration:

```
[VirtualUser("Vuser")]
public class Vuser
{
  public DotNetProject.com.borland.demo.Service1 mService;
  [Transaction(Etranstype.TRANSTYPE_INIT)]
  public void TInit()
  {
    mService = new DotNetProject.com.borland.demo.Service1();
  }
}
```

Calling a Web Service Method

All methods that are exposed by Web services are also available in proxy objects. The methods that are shared by proxy objects use the same names as their corresponding WSDLs. Web-service method calls should be placed in `main` transactions.

Example

```
[Transaction(Etranstype.TRANSTYPE_MAIN)]
public void TMain()
{
  string sReturn = mService.echoString("Test");
  Bdl.Print(sReturn);
}
```

To customize your Web-service calls from a generated BDL script, you must allow the exchange of data between BDL and .NET with usage of attributes or method parameters.

Example

```
[Transaction(Etranstype.TRANSTYPE_MAIN)]
[TestAttribute("EchoInput", "Test")]
public void TMain()
{
  string sReturn =
mService.echoString(Bdl.AttributeGet("EchoInput"));
  Bdl.Print(sReturn);
}
```

or

```
[Transaction(Etranstype.TRANSTYPE_MAIN)]
public void TMain(string sEcho)
{
  string sReturn = mService.echoString(sEcho);
  Bdl.Print(sReturn);
}
```

Routing Web-Service Traffic

The SilkPerformer .NET Framework can route Web traffic generated by .NET components through the SilkPerformer Web engine. This means that the Web engine executes the actual Web requests, enabling you to see exactly what is sent over the wire. You can also use features of the SilkPerformer Web engine, like modem simulation, IP multiplexing, network statistics, TrueLog, and others.

By default all network traffic is routed through the Web engine. You can switch the routing off and only enable it for specific Web-service client-proxy classes. To switch the routing on for specific Web-service client-proxy classes, you need to change the base class of the proxy classes from `SoapHttpClientProtocol` to `SilkPerformer.SPSoapHttpClientProtocol`. Changing the base class allows the SilkPerformer .NET Framework to generate more detailed statistical information for each Web-service call. We recommend that you enable this feature for all your Web-service proxy classes. You can enable this feature by using the **Web Service** dialog box in Microsoft Visual Studio, which is accessible through the SilkPerformer menu.

For each Web-service call a node is created in the TrueLog with the SOAP envelope that was passed to the Web service and returned to the client.

If detailed statistical information for Web-service calls is disabled, the .NET HTTP classes process all requests.

Exploring Results in Visual Studio

A TrueLog node is logged for each Web-service call that is executed by the .NET test driver, if routing Web-service traffic is enabled. Click the **Rendered** tab of the node in TrueLog Explorer for the *Response SOAP Envelope*, or click the **Post Data** tab for the *Request SOAP Envelope*.

The overview report for each Web-service method that is called contains the statistical information, like the *round-trip time*, the *server busy time*, and others.

Testing with .NET Explorer

.NET Explorer is a tool that allows you to generate .NET test drivers through a point & click approach. .NET Explorer supports the following technologies:

- SOAP Web Services
- .NET Remoting
- .NET Components

For additional information, refer to the *.NET Explorer Help*.

Available BDL Functions for .NET Interoperability

This section describes the BDL functions that The SilkPerformer .NET Framework and .NET add-on allow easy access to Web services from within .NET. provides for .NET interoperability. For detailed descriptions of these functions and all other BDL API functions, refer to the *Benchmark Description Language (BDL) Reference*. BDL reference details are also available in SilkPerformer Help.

DotNetLoadObject Function

Action

Loads a .NET Assembly and creates an instance of a .NET type. A handle to this created object will be returned. When creating the object, the default constructor will be called. If you want to call a constructor that takes parameters you have to set these parameters with the DotNetSetXX functions prior to the call to `DotNetLoadObject`.

Include file

`DotNetAPI.bdh`

Syntax

```
DotNetLoadObject( in sAssemblyFile : string,
                  in sTypeName     : string,
                  in sTimer        : string optional ): number;
```

Return value

- object handle if successful
- 0 otherwise

Parameter	Description
`sAssemblyFile`	Path to the assembly file that contains the specified type. The path can be either absolute or relative. If the path is relative, it is either relative to the project or the data directory.
`sTypeName`	Full qualified type name (Namespace + TypeName) of the type that should be instantiated.
`sTimer`	(optional) If defined – a custom timer will be generated to measure the creation time of the object.

Example

```
dcltrans
  transaction TMain
  var
    hObject : number;
begin
  hObject := DotNetLoadObject("bin\\Release\\MyDotNet.dll",
"MyDotNet.TestClass");
  DotNetCallMethod(hObject,"TestMethod");
  DotNetFreeObject(hObject);
end TMain;
```

DotNetFreeObject Function

Action

Releases the object handle, so that the garbage collector will free the object.

Include file

`DotNetAPI.bdh`

Syntax

```
DotNetFreeObject( in hObject : number ): boolean;
```

Return value

- `true` if successful
- `false` otherwise

Parameter	Description
`hObject`	Handle to a .NET Object that will be released

Example

```
dcltrans
  transaction TMain
  var
    hObject : number;
begin
  hObject := DotNetLoadObject("bin\\Release\\MyDotNet.dll",
"MyDotNet.TestClass");
```

```
  DotNetCallMethod(hObject,"TestMethod");
  DotNetFreeObject(hObject);
end TMain;
```

DotNetCallMethod Function

Action

Calls a public method on the .NET object or a static method of a .NET type. If parameters have been set with the DotNetSetXX methods, these parameters will be passed in the order of the DotNetSetXX calls. If you want to call a static method you have to use the constant DOTNET_STATIC_METHOD as object handle for the `DotNetCallMethod` and the DotNetSetXX methods. Another way to pass parameters to and from the method is to use the AttributeXX functions. The .NET Method can use the perfDotNetFW.dll to call into the SilkPerformer runtime to get/set attributes (Bdl.AttributeGet, Bdl.AttributeSet).

If the function returns a value, this value can be retrieved with the DotNetGetXX methods. Again - if calling a static method - use DOTNET_STATIC_METHOD as object handle.

Include file

`DotNetAPI.bdh`

Syntax

```
DotNetCallMethod( in hObject     : number,
                  in sMethodName : string,
                  in sTypeName   : string optional,
                  in sAssembly   : string optional,
                  in sTimer      : string optional ): boolean;
```

Return value

- `true` if successful
- `false` otherwise

Parameter	Description
`hObject`	Handle to a .NET Object or DOTNET_STATIC_METHOD if you want to call a static method. In this case you have to at least specify the name of the .NET type (class) you want to call the method.
`sMethodName`	Method that should be called
`sTypeName`	(optional) If you want to call a static method this parameter specifies the .NET type (class) of the method.
`sAssembly`	(optional) If you want to call a static method and this parameter is omitted the type specified in `sTypeName` is searched in the currently loaded assemblies. If you haven't loaded the assembly where `sTypeName` is implemented you can specify the assembly file here

Parameter	Description
	and it will be loaded. Assemblies are normally loaded during `DotNetLoadObject`. The basic .NET assembly (mscorlib) is always loaded - so you can access all static methods of the basic classes.
`sTimer`	(optional) If defined – a custom timer will be generated to measure the execution time of the method call.

Example

```
dcltrans
  transaction TMain
  var
    hObject, nValue : number;
  begin
    // load an object and call a method on this instance
    hObject := DotNetLoadObject("C:\\MyDotNet\\Bin\\Release\
\MyDotNet.dll", "MyDotNet.TestClass");
    DotNetCallMethod(hObject,"TestMethod");
    DotNetFreeObject(hObject);
    // call a static method - no additional assembly needs to
be loaded because DateTime is defined in mscorlib
    DotNetSetInt(DOTNET_STATIC_METHOD, 2003);
    DotNetSetInt(DOTNET_STATIC_METHOD, 2);
    DotNetCallMethod(DOTNET_STATIC_METHOD, "DaysInMonth",
"System.DateTime");
    DotNetGetInt(DOTNET_STATIC_METHOD, nValue);
  end TMain;
```

DotNetSetString Function

Action

Sets a string parameter for the next `DotNetCallMethod` or `DotNetLoadObject` call in an internal parameter array. The parameters will be passed in the order they have been set to the internal array. After calling `DotNetCallMethod` or `DotNetLoadObject` the internal parameter array will be cleared.

Include file

`DotNetAPI.bdh`

Syntax

```
DotNetSetString( in hObject : number,
                 in sParam  : string );
```

Parameter	Description
`hObject`	Handle to a .NET Object to set the parameter for the next `DotNetCallMethod` call or DOTNET_CONSTRUCTOR to set the parameter for the next DotNetLoadObject call.
`sParam`	String value that should be passed as parameter to the next `DotNetCallMethod` on the passed .NET Object

Example

```
dcltrans
  transaction TMain
  var
    hObject, hObject2 : number;
    sReturnValue      : string;
  begin
    DotNetSetString(hObject, "ConstrValue1");

    hObject := DotNetLoadObject("bin\\Release\\MyDotNet.dll",
"MyDotNet.TestClass");
    hObject2 := DotNetLoadObject("bin\\Release\\MyDotNet.dll",
"MyDotNet.ParamClass");
    DotNetSetFloat(hObject, 1.23);
    DotNetSetInt(hObject, 123);
    DotNetSetBool(hObject, false);
    DotNetSetObject(hObject, hObject2);
    DotNetCallMethod(hObject,"TestMethod");
    DotNetGetString(hObject, sReturnValue);
    DotNetFreeObject(hObject);
    DotNetFreeObject(hObject2);
  end TMain;
```

DotNetSetFloat Function

Action

Sets a float parameter for the next `DotNetCallMethod` or `DotNetLoadObject` call in an internal parameter array. The parameters will be passed in the order they have been set to the internal array. After calling `DotNetCallMethod` or `DotNetLoadObject` the internal parameter array will be cleared.

Include file

`DotNetAPI.bdh`

Syntax

```
DotNetSetFloat( in hObject : number,
                in fParam  : float );
```

Parameter	Description
`hObject`	Handle to a .NET Object to set the parameter for the next `DotNetCallMethod` call or DOTNET_CONSTRUCTOR to set the parameter for the next DotNetLoadObject call.
`fParam`	Float value that should be passed as parameter to the next `DotNetCallMethod` on the passed .NET Object

Example

```
dcltrans
  transaction TMain
  var
    hObject, hObject2 : number;
    sReturnValue      : string;
  begin
    DotNetSetString(hObject, "ConstrValue1");
    hObject := DotNetLoadObject("bin\\Release\\MyDotNet.dll",
"MyDotNet.TestClass");
    hObject2 := DotNetLoadObject("bin\\Release\\MyDotNet.dll",
"MyDotNet.ParamClass");
    DotNetSetFloat(hObject, 1.23);
    DotNetSetInt(hObject, 123);
    DotNetSetBool(hObject, false);
    DotNetSetObject(hObject, hObject2);
    DotNetCallMethod(hObject,"TestMethod");
    DotNetGetString(hObject, sReturnValue);
    DotNetFreeObject(hObject);
    DotNetFreeObject(hObject2);
  end TMain;
```

DotNetSetBool Function

Action

Sets a boolean parameter for the next `DotNetCallMethod` or `DotNetLoadObject` call in an internal parameter array. The parameters will be passed in the order they have been set to the internal array. After calling `DotNetCallMethod` or `DotNetLoadObject` the internal parameter array will be cleared.

Include file

`DotNetAPI.bdh`

Syntax

```
DotNetSetBool( in hObject : number,
               in bParam  : boolean );
```

Parameter	Description
`hObject`	Handle to a .NET Object to set the parameter for the next `DotNetCallMethod` call or DOTNET_CONSTRUCTOR to set the parameter for the next `DotNetLoadObject` call.
`bParam`	Boolean value that should be passed as parameter to the next `DotNetCallMethod` on the passed .NET Object

Example

```
dcltrans
  transaction TMain
  var
    hObject, hObject2 : number;
    sReturnValue      : string;
  begin
    DotNetSetString(hObject, "ConstrValue1");

    hObject := DotNetLoadObject("bin\\Release\\MyDotNet.dll",
"MyDotNet.TestClass");
    hObject2 := DotNetLoadObject("bin\\Release\\MyDotNet.dll",
"MyDotNet.ParamClass");
    DotNetSetFloat(hObject, 1.23);
    DotNetSetInt(hObject, 123);
    DotNetSetBool(hObject, false);
    DotNetSetObject(hObject, hObject2);
    DotNetCallMethod(hObject,"TestMethod");
    DotNetGetString(hObject, sReturnValue);
    DotNetFreeObject(hObject);
    DotNetFreeObject(hObject2);
  end TMain;
```

DotNetSetInt Function

Action

Sets an integer parameter for the next `DotNetCallMethod` or `DotNetLoadObject` call in an internal parameter array. The parameters will be passed in the order they have been set to the internal array. After calling `DotNetCallMethod` or `DotNetLoadObject` the internal parameter array will be cleared.

Include file

`DotNetAPI.bdh`

Syntax

```
DotNetSetInt( in hObject : number,
              in nParam  : number );
```

Parameter	Description
`hObject`	Handle to a .NET Object to set the parameter for the next `DotNetCallMethod` call or DOTNET_CONSTRUCTOR to set the parameter for the next `DotNetLoadObject` call.
`nParam`	Integer value that should be passed as parameter to the next `DotNetCallMethod` on the passed .NET Object

Example

```
dcltrans
  transaction TMain
  var
    hObject, hObject2 : number;
    sReturnValue      : string;
  begin
    DotNetSetString(hObject, "ConstrValue1");

    hObject := DotNetLoadObject("bin\\Release\\MyDotNet.dll",
"MyDotNet.TestClass");
    hObject2 := DotNetLoadObject("bin\\Release\\MyDotNet.dll",
"MyDotNet.ParamClass");
    DotNetSetFloat(hObject, 1.23);
    DotNetSetInt(hObject, 123);
    DotNetSetBool(hObject, false);
    DotNetSetObject(hObject, hObject2);
    DotNetCallMethod(hObject,"TestMethod");
    DotNetGetString(hObject, sReturnValue);
    DotNetFreeObject(hObject);
    DotNetFreeObject(hObject2);
  end TMain;
```

DotNetSetObject Function

Action

Sets an object as parameter for the next `DotNetCallMethod` or `DotNetLoadObject` call in an internal parameter array. The parameters will be passed in the order they have been set to the internal array. After calling `DotNetCallMethod` or `DotNetLoadObject` the internal parameter array will be cleared.

Include file

`DotNetAPI.bdh`

Syntax

```
DotNetSetObject( in hObject : number,
                 in hParam  : number );
```

Parameter	Description
`hObject`	Handle to a .NET Object to set the parameter for the next `DotNetCallMethod` call or DOTNET_CONSTRUCTOR to set the parameter for the next DotNetLoadObject call.
`hParam`	.NET Object handle that should be passed as parameter to the next `DotNetCallMethod` on the passed .NET Object

Example

```
dcltrans
  transaction TMain
  var
    hObject, hObject2 : number;
    sReturnValue      : string;
  begin
    DotNetSetString(hObject, "ConstrValue1");

    hObject := DotNetLoadObject("bin\\Release\\MyDotNet.dll",
"MyDotNet.TestClass");
    hObject2 := DotNetLoadObject("bin\\Release\\MyDotNet.dll",
"MyDotNet.ParamClass");
    DotNetSetFloat(hObject, 1.23);
    DotNetSetInt(hObject, 123);
    DotNetSetBool(hObject, false);
    DotNetSetObject(hObject, hObject2);
    DotNetCallMethod(hObject,"TestMethod");
    DotNetGetString(hObject, sReturnValue);
    DotNetFreeObject(hObject);
    DotNetFreeObject(hObject2);
  end TMain;
```

DotNetGetString Function

Action

Gets the string return value of the last `DotNetCallMethod` call in `sReturn`. Parameter `sRetLen` defines the size of the `sReturn` string buffer.

Include file

`DotNetAPI.bdh`

Syntax

```
DotNetGetString( in    hObject : number,
                 inout sReturn : string,
                 in    nRetLen : number optional ) :boolean;
```

Return value

- `true` if successful
- `false` otherwise

Parameter	Description
`hObject`	Handle to a .NET Object.
`sReturn`	String buffer that will receive the return value of the last `DotNetCallMethod` call.
`nRetLen`	Size of the string buffer passed in `sReturn` (optional).

Example

```
dcltrans
  transaction TMain
  var
    hObject, hObject2 : number;
    sReturnValue : string;
  begin
    DotNetSetString(hObject, "ConstrValue1");

    hObject := DotNetLoadObject("bin\\Release\\MyDotNet.dll",
"MyDotNet.TestClass");
    hObject2 := DotNetLoadObject("bin\\Release\\MyDotNet.dll",
"MyDotNet.ParamClass");
    DotNetSetFloat(hObject, 1.23);
    DotNetSetInt(hObject, 123);
    DotNetSetBool(hObject, false);
    DotNetSetObject(hObject, hObject2);
    DotNetCallMethod(hObject,"TestMethod");
    DotNetGetString(hObject, sReturnValue);
    DotNetFreeObject(hObject);
    DotNetFreeObject(hObject2);
  end TMain;
```

DotNetGetFloat Function

Action

Gets the float return value of the last `DotNetCallMethod` call in `fReturn`.

Include file

`DotNetAPI.bdh`

Syntax

```
DotNetGetFloat( in    hObject : number,
                inout fReturn : float ): boolean;
```

Return value

- `true` if successful
- `false` otherwise

Parameter	Description
`hObject`	Handle to a .NET Object
`fReturn`	Float variable that will receive the return value of the last `DotNetCallMethod` call

Example

```
dcltrans
  transaction TMain
  var
    hObject, hObject2 : number;
    fReturn : float;
  begin
    DotNetSetString(hObject, "ConstrValue1");

    hObject := DotNetLoadObject("bin\\Release\\MyDotNet.dll",
"MyDotNet.TestClass");
    hObject2 := DotNetLoadObject("bin\\Release\\MyDotNet.dll",
"MyDotNet.ParamClass");
    DotNetSetFloat(hObject, 1.23);
    DotNetSetInt(hObject, 123);
    DotNetSetBool(hObject, false);
    DotNetSetObject(hObject, hObject2);
    DotNetCallMethod(hObject,"TestMethod");
    DotNetGetFloat(hObject, fReturn);
    DotNetFreeObject(hObject);
    DotNetFreeObject(hObject2);
  end TMain;
```

DotNetGetBool Function

Action

Gets the boolean return value of the last `DotNetCallMethod` call in `bReturn`.

Include file

`DotNetAPI.bdh`

Syntax

```
DotNetGetBool( in    hObject : number,
               inout bReturn : boolean ): boolean;
```

Return value

- `true` if successful
- `false` otherwise

Parameter	Description
`hObject`	Handle to a .NET Object
`bReturn`	Boolean variable that will receive the return value of the last `DotNetCallMethod` call

Example

```
dcltrans
  transaction TMain
  var
    hObject, hObject2 : number;
    bReturn           : boolean;
  begin
    DotNetSetString(hObject, "ConstrValue1");
    hObject := DotNetLoadObject("bin\\Release\\MyDotNet.dll",
"MyDotNet.TestClass");
    hObject2 := DotNetLoadObject("bin\\Release\\MyDotNet.dll",
"MyDotNet.ParamClass");
    DotNetSetFloat(hObject, 1.23);
    DotNetSetInt(hObject, 123);
    DotNetSetBool(hObject, false);
    DotNetSetObject(hObject, hObject2);
    DotNetCallMethod(hObject,"TestMethod");
    DotNetGetBool(hObject, bReturn);
    DotNetFreeObject(hObject);
    DotNetFreeObject(hObject2);
  end TMain;
```

DotNetGetInt Function

Action

Gets the integer return value of the last `DotNetCallMethod` call in `nReturn`.

Include file

`DotNetAPI.bdh`

Syntax

```
DotNetGetInt( in    hObject : number,
              inout nReturn : number ): boolean;
```

Return value

- `true` if successful
- `false` otherwise

Parameter	Description
`hObject`	Handle to a .NET Object
`nReturn`	Integer variable that will receive the return value of the last `DotNetCallMethod` call

Example

```
dcltrans
  transaction TMain
  var
    hObject, hObject2 : number;
    nReturn           : number;
  begin
    DotNetSetString(hObject, "ConstrValue1");

    hObject := DotNetLoadObject("bin\\Release\\MyDotNet.dll",
"MyDotNet.TestClass");
    hObject2 := DotNetLoadObject("bin\\Release\\MyDotNet.dll",
"MyDotNet.ParamClass");
    DotNetSetFloat(hObject, 1.23);
    DotNetSetInt(hObject, 123);
    DotNetSetBool(hObject, false);
    DotNetSetObject(hObject, hObject2);
    DotNetCallMethod(hObject,"TestMethod");
    DotNetGetInt(hObject, nReturn);
    DotNetFreeObject(hObject);
    DotNetFreeObject(hObject2);
  end TMain;
```

DotNetGetObject Function

Action

Gets the handle to the object returned by the last `DotNetCallMethod` call in `hReturn`. The returned object handle has to be freeded with DotNetFreeObject.

Include file

`DotNetAPI.bdh`

Syntax

```
DotNetGetObject( in    hObject : number,
                 inout hReturn : number ): number;
```

Return value

- object handle if successful
- 0 otherwise

Parameter	Description
`hObject`	Handle to a .NET Object
`hReturn`	Integer variable that will receive the returned .NET object handle of the last `DotNetCallMethod` call

Example

```
dcltrans
  transaction TMain
  var
    hObject, hObject2 : number;
    hReturn           : number;
  begin
    DotNetSetString(hObject, "ConstrValue1");

    hObject := DotNetLoadObject("bin\\Release\\MyDotNet.dll",
"MyDotNet.TestClass");
    hObject2 := DotNetLoadObject("bin\\Release\\MyDotNet.dll",
"MyDotNet.ParamClass");
    DotNetSetFloat(hObject, 1.23);
    DotNetSetInt(hObject, 123);
    DotNetSetBool(hObject, false);
    DotNetSetObject(hObject, hObject2);
    DotNetCallMethod(hObject,"TestMethod");
    DotNetGetObject(hObject, hReturn);
    DotNetFreeObject(hObject);
    DotNetFreeObject(hObject2);
    DotNetFreeObject(hReturn);
  end TMain;
```

Index