



Silk Performer 21.0

.NET Framework Help

Micro Focus
The Lawn
22-30 Old Bath Road
Newbury, Berkshire RG14 1QN
UK
<http://www.microfocus.com>

© Copyright 1992-2020 Micro Focus or one of its affiliates.

MICRO FOCUS, the Micro Focus logo and Silk Performer are trademarks or registered trademarks of Micro Focus or one of its affiliates.

All other marks are the property of their respective owners.

2020-10-21

Contents

Tools and Samples	5
Introduction	5
Provided Tools	6
Silk Performer .NET Explorer	6
Silk Performer Visual Studio Extension	6
Silk Performer Java Explorer	6
Silk Performer Workbench	7
Sample Applications for testing Java and .NET	7
Public Web Services	7
.NET Message Sample	8
.NET Explorer Remoting Sample	8
Java RMI Samples	8
Sample Test Projects	9
.NET Sample Projects	9
Java Sample Projects	10
Silk Performer .NET Framework	11
Testing .NET Components	11
The .NET Framework Approach	11
The .NET Explorer Approach	11
Understanding the .NET Framework Platform	11
Working with Silk Performer .NET Framework	11
Silk Performer .NET Framework Overview	12
Intermediate Code	13
Silk Performer Helper Classes	13
Silk Performer Visual Studio Extension	13
Load Testing .NET Components	15
Setting Up Silk Performer .NET Projects	15
Creating a Web Service Client Proxy	16
Instantiating Client Proxy Objects	17
Try Script Runs From Microsoft Visual Studio	17
Executing a Try Script Run	17
Web Service Calls	18
Routing Web Service Calls	19
Dependencies	19
Adding Dependencies	19
Configuring .NET Add-In Option Settings	20
Continuing Your Work in Silk Performer	20
Custom Attributes	20
Attributes for Unit Test Standards	22
Negative Testing	22
Custom Attributes Code Sample	23
Generated BDF Script Example	23
Testing .NET Services	25
Development Workflow	25
Writing a .NET Test Driver	25
Creating a .NET Project	25
Defining a Virtual User in .NET	26
Defining a Transaction in .NET	26
Defining Additional Test Methods	27
Passing Data Between BDL and .NET	28

Calling BDL Functions from .NET	30
Random Variables	32
Exception Handling	32
Debugging	33
Configuration Files	33
BDL Code Generation Engine	34
Virtual User	34
Transactions - Virtual User Classes	35
Test Methods	35
Test Attributes	36
Methods with Parameters	36
BDL Parameters	38
Intelligent Parameter Passing	38
Options	39
Testing Your .NET Test Driver	41
Preparations	41
Try Script Runs	43
Exploring Results	43
Running Tests in Silk Performer	44
Testing Web Services With Microsoft Visual Studio	44
Creating a Web Service Client Proxy	44
Instantiating a Client Proxy Object	44
Calling a Web Service Method	45
Routing Web-Service Traffic	45
Exploring Results in Visual Studio	46
Testing with .NET Explorer	46
Available BDL Functions for .NET Interoperability	46
DotNetLoadObject Function	46
DotNetFreeObject Function	47
DotNetCallMethod Function	48
DotNetSetString Function	49
DotNetSetFloat Function	50
DotNetSetBool Function	51
DotNetSetInt Function	52
DotNetSetObject Function	53
DotNetGetString Function	54
DotNetGetFloat Function	55
DotNetGetBool Function	55
DotNetGetInt Function	56
DotNetGetObject Function	57

Tools and Samples

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

Introduction

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 Silk Performer Service Oriented Architecture (SOA) Edition.

Silk Performer SOA Edition Licensing

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

What You Can Test With Silk Performer SOA Edition

With Silk Performer 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, Silk Performer SOA Edition can test components under concurrent access by up to five virtual users, thereby emulating realistic server conditions. With a full Silk Performer license, the number of virtual users can be scaled even higher. In addition to testing the functionality of remote components, Silk Performer SOA Edition also verifies the performance and interoperability of components.

Silk Performer 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

Silk Performer 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 Silk Performer for testing Java and .NET.

Silk Performer .NET Explorer

Silk Performer .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 Silk Performer for immediate reuse in concurrency and load testing, and to Microsoft Visual Studio for further customization.

Silk Performer Visual Studio Extension

The Silk Performer Visual Studio extension allows you to implement test drivers in Microsoft Visual Studio that are compatible with Silk Performer. Such test drivers can be augmented with Silk Performer features that facilitate test organization, verification, performance measurement, test data generation, and reporting.

Tests created with the extension can be run either within Microsoft Visual Studio, with full access to Silk Performer's functionality, or within Silk Performer, for concurrency and load testing scenarios.

The extension offers the following features:

- Writing test code in any of the main .NET languages (C# or VB.NET).
- Testing Web services / .NET Remoting objects and redirecting HTTP traffic over the Silk Performer 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>

Silk Performer Java Explorer

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

Test scenarios created with Java Explorer can be exported to Silk Performer 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>

Silk Performer Workbench

Remote component tests that are developed and executed using Java Explorer or .NET Explorer can be executed within Silk Performer Workbench. Silk Performer 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 Silk Performer by creating Silk Performer Java Framework and .NET Framework projects. While Java Explorer and .NET Explorer serve as test-beds for functional test scenarios, Silk Performer can be used to run the same test scripts in more complex scenarios for concurrency and load testing.


In the same way that Silk Performer is integrated with Java Explorer and .NET Explorer, Silk Performer is also integrated with Silk Performer's Visual Studio .NET Add-On. Test clients created in Microsoft Visual Studio using Silk Performer's Visual Studio .NET Add-On functionality can easily be exported to Silk Performer 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, Silk Performer offers features that assist Java developers, such as syntax highlighting for Java and the ability to run the Java compiler from Silk Performer Workbench.

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

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

In addition to creating test clients visually and manually, Silk Performer 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 Silk Performer and create a new project of application type `Web Services/XML/SOAP`.

Sample Applications for testing Java and .NET

The sample applications provided with Silk Performer enable you to experiment with Silk Performer'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 Silk Performer SOA Edition: Go to **Start > Programs > Silk > Silk Performer SOA Edition 21.0 > Sample Applications > .NET Framework Samples** .

If you have Silk Performer Enterprise Edition: Go to **Start > Programs > Silk > Silk Performer 21.0 > 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 Silk Performer SOA Edition: Go to **Start > Programs > Silk > Silk Performer SOA Edition 21.0 > Sample Applications > .NET Explorer Samples > .NET Explorer Remoting Sample** .

If you have Silk Performer Enterprise Edition: Go to **Start > Programs > Silk > Silk Performer 21.0 > Sample Applications > .NET Explorer Samples > .NET Explorer Remoting Sample** .

DLL reference for .NET Explorer: `<public user documents>\Silk Performer 21.0\SampleApps\DOTNET\RemotingSamples\RemotingLib\bin\debug\RemotingLib.dll`.

Java RMI Samples

Two Java RMI sample applications are included:

- A simple RMI sample application that is used in conjunction with the sample Java Framework project (`<public user documents>\Silk Performer 21.0\Samples\JavaFramework\RMI`).

To start the ServiceHello RMI Server, go to: **Start > Programs > Silk > Silk Performer 21.0 > Sample Applications > Java Samples > RMI Sample - SayHello**.

- 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 > Silk Performer 21.0 > Sample Applications > Java Samples > Product Manager**. This sample can be used with the sample test project that is available at `<public user documents>\Silk Performer 21.0\SampleApps\RMILdap`.

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 <public user documents>\Silk Performer 21.0\SampleApps\Java.

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: <public user documents>\Silk Performer 21.0\SampleApps\Java\Lib\sampleRmi.jar.

RMI over IIOP

A simple example server can be found at: <public user documents>\Silk Performer 21.0\SampleApps\Java.

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: <public user documents>\Silk Performer 21.0\SampleApps\Java\Lib\sampleRmiOverIiop.jar.

Sample Test Projects

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

.NET Sample Projects

.NET Remoting

This sample project implements a simple .NET Remoting client using the Silk Performer .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 Silk Performer .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 Silk Performer 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 Silk Performer 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 `<public user documents>\Silk Performer 21.0\SampleApps\RMIldap\Readme.html`.

The Java RMI server can be found at: `<public user documents>\Silk Performer 21.0\SampleApps\RMIldap`.

RMI

This sample project implements a Java RMI client using the Silk Performer 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 `<public user documents>\Silk Performer 21.0\SampleApps\RMIldap\Readme.html`.

To access the Java RMI server:

If you have Silk Performer SOA Edition: Go to **Start > Programs > Silk > Silk Performer SOA Edition 21.0 > Sample Applications > Java Samples > RMI Sample - SayHello** .

If you have Silk Performer Enterprise Edition: Go to **Start > Programs > Silk > Silk Performer 21.0 > Sample Applications > Java Samples > RMI Sample - SayHello**.

Silk Performer .NET Framework

Silk Performer'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 Silk Performer—there is no need for staff to learn new tools. Silk Performer's .NET Framework thereby encourages efficiency and tighter integration between QA and development. The Silk Performer .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

Silk Performer's Visual Studio .NET Add-On provides functionality to developers working in .NET-enabled languages for generating Silk Performer 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 (Silk Performer'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 Silk Performer's Visual Studio .NET Add-On, developers can generate Silk Performer 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 Silk Performer.

The .NET Explorer Approach

.NET Explorer is a GUI-driven tool that is well suited to QA personnel who are proficient with Silk Performer in facilitating analysis of .NET components and thereby creating Silk Performer 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 Silk Performer .NET Framework

The Silk Performer .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 Silk Performer and an add-on for Microsoft Visual Studio.

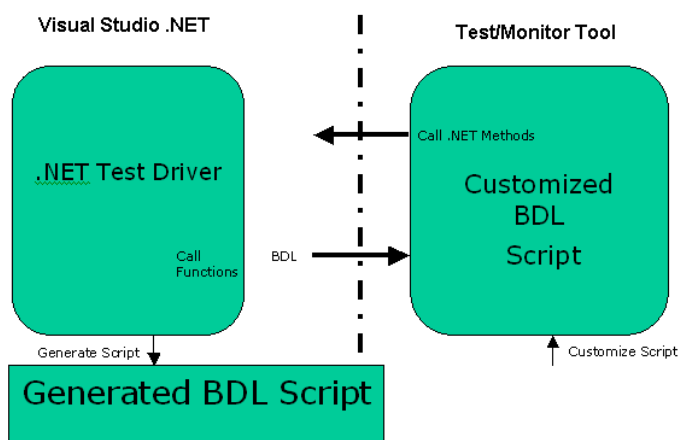
 **Note:** For additional details regarding the available BDL API functions, refer to the *Benchmark Description Language (BDL) Reference*.

The framework allows you to either code your BDL calls to .NET objects manually in Silk Performer 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 Silk Performer 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 Silk Performer BDL script.

Silk Performer .NET Framework Overview



The Silk Performer .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 Silk Performer 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 Silk Performer 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.

Silk Performer Helper Classes

.NET helper classes serve as an interface between Silk Performer’s BDL language and the .NET language. Although Silk Performer 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 Silk Performer. 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 Silk Performer or scheduled in load tests using Silk Central.

Silk Performer Visual Studio Extension

The Silk Performer Visual Studio extension allows you to implement test drivers in Microsoft Visual Studio that are compatible with Silk Performer. Such test drivers can be augmented with Silk Performer features that facilitate test organization, verification, performance measurement, test data generation, and reporting.

Tests created with the extension can be run either within Microsoft Visual Studio, with full access to Silk Performer’s functionality, or within Silk Performer, for concurrency and load testing scenarios.

The extension offers the following features:

- Writing test code in any of the main .NET languages (C# or VB.NET).
- Testing Web services / .NET Remoting objects and redirecting HTTP traffic over the Silk Performer 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>

Installing the Visual Studio Extension

By default, the Silk Performer Visual Studio Extension is not installed with the main Silk Performer installer. To create and run Silk Performer .NET Framework projects in Visual Studio, you need to install the extension:

1. In the Silk Performer installation directory, open `Templates\DotNet`.
2. Execute the file `SpVsExtension.vsix`.

Starting the Visual Studio Extension

Perform one of the following steps to start the Visual Studio extension:

- Click **Start > All Programs > Microsoft Visual Studio > Microsoft Visual Studio** and create a new Silk Performer Visual Studio project.
- Click **Start > All Programs > Silk > Silk Performer 21.0 > Silk Performer 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 Silk Performer Visual Studio .NET Add-On for the testing of .NET components and Web services.

Setting Up Silk Performer .NET Projects

1. Click **Start here** on the Silk Performer 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 **Name** text box, enter a name for your project.
3. Enter an optional project description in **Description**.
4. From the **Type** menu tree, select **.NET > .NET Framework using Visual Studio .NET Add-On** and click **Next**. The **Workflow - Model Script** dialog box opens.
5. Select the .NET Language (**C#** or **VB.NET**) icon for the language you prefer and click **OK**. The Microsoft Visual Studio **Silk Performer Project Wizard** opens.
6. Enter the name of the .NET Testclass in the **Name of testclass** text box. In the **Silk Performer Project** text box, enter the name of the project that you created earlier in Silk Performer.
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 Silk Performer 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 Silk Performer;

namespace SPPProject1
{
    [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 Silk Performer 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 Silk Performer 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 Silk Performer 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 Silk Performer 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 Silk Performer Web engine, enabling TrueLog output. You can turn off redirection or enable it for specific Web Service client proxy classes via the Silk Performer **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 Silk Performer **Results** menu.

Web Service Calls

The Silk Performer .NET Framework can route Web traffic generated by .NET components over the Silk Performer Web engine. This means that the Silk Performer 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 Silk Performer Web engine features such as modem simulation, IP multiplexing, network statistics, and TrueLog.

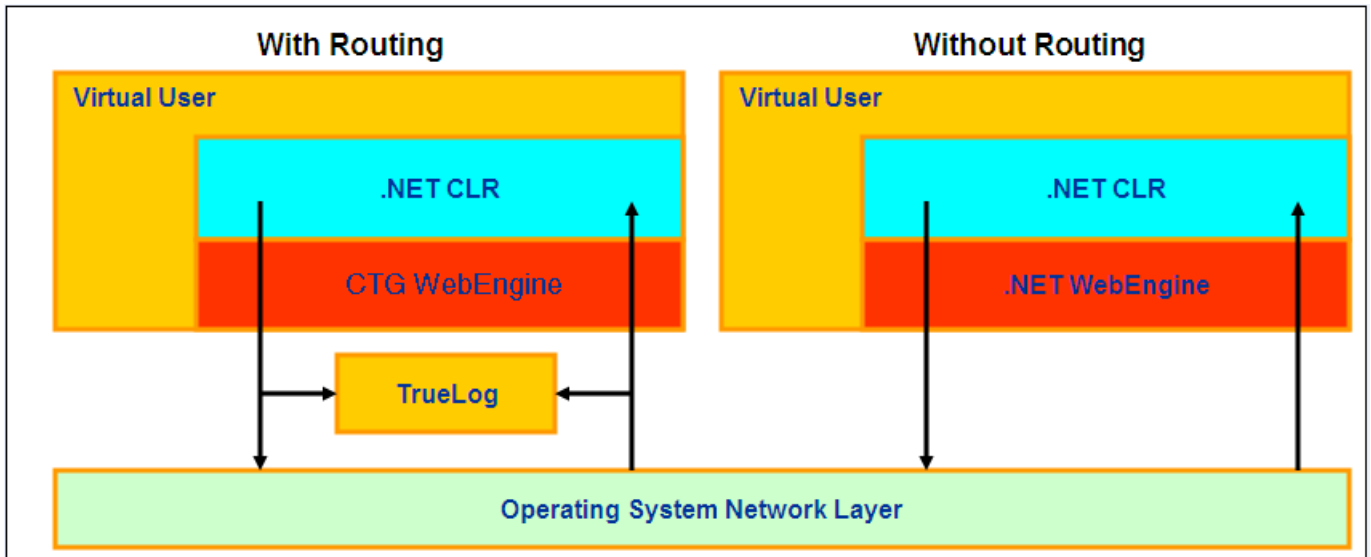
By default, all network traffic is routed over the Web engine. You can however enable routing only for specific Web Service client proxy classes. To enable this feature only for specific Web Service proxy classes, change the base class of a proxy class from `SoapHttpClientProtocol` to `SilkPerformer.SPSoapHttpClientProtocol`.

This base class exchange allows the Silk Performer .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 Silk Performer 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 Silk Performer, the HTTP traffic responsible for Web Service calls is routed over the Silk Performer Web engine. Network traffic and statistics are then written to the TrueLog. Modem simulation and IP multiplexing are also available.



Routing Web Service Calls

1. Open the **Web Services** dialog box (select **Web settings** from the Silk Performer menu).
2. Select the Web Service proxy classes that should be instrumented by Silk Performer.
These are the classes that will be routed over the Silk Performer 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 Silk Performer menu).

The files you specify will be added to your Silk Performer 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 Silk Performer 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 Silk Performer 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 Silk Performer 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, <code>MeasureStart</code> and <code>MeasureStop</code> statements are scripted around each <code>DotNetCallMethod</code> 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 Silk Performer

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

In Silk Performer, you can run tests with multiple users distributed over multiple agents. Take advantage of the Silk Performer 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 Silk Performer's project attribute wizard.
VirtualUserInitialize	Method		This method is called for classes that are loaded via DotNetUnitTestLoadObject
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

Attribute Class	Applicable to	Parameters	Description
TestIgnore	Method		Methods that have this attribute applied to them will not be called via DotNetUnitTestMethod
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 Silk Performer 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.

Silk Performer 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	<ul style="list-style-type: none"> - Exception type - Log text if the anticipated exception is not thrown (<i>optional</i>) 	<p>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.</p>

Custom Attributes Code Sample

C# Test Code Sample

```

using System;
using SilkPerformer;

namespace SPPProject1
{
    [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("\\SPPProject1\\bin\\release\\
\SPPProject1.dll", "SPPProject1.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;

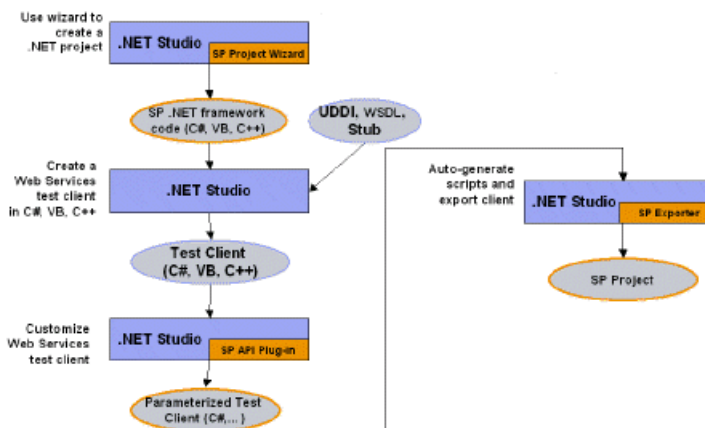
transaction VUser_TEnd
begin
    MeasureStart("TEnd");
    DotNetCallMethod(hVUser, "TEnd");
    MeasureStop("TEnd");
    DotNetFreeObject(hVUser);
end VUser_TEnd;

```


Testing .NET Services

This section offers an overview of the Silk Performer .NET Framework and the Silk Performer .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 Silk Performer and the .NET Framework.

Development Workflow



Writing a .NET Test Driver

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

Creating a .NET Project

Silk Performer 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# or VB.NET, 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 Silk Performer or by creating a new Silk Performer project in Visual Studio .NET. In either case you start with a new Silk Performer .NET project in Microsoft Visual Studio .NET with the sample test code file open.

`PerfDotNetFW.dll` is a .NET assembly that ships with Silk Performer. It implements all classes, custom attributes, and enums that can be used to define meta information for automatic Silk Performer 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	BDL Script
<pre>[VirtualUser("Vuser1")] public class MyTestUser1 { ... } [VirtualUser("Vuser2")] public class MyTestUser2 { ... }</pre>	<pre>dcluser user Vuser1 ... user Vuser2</pre>

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	BDL Script
<pre>[VirtualUser("Vuser1")] public class MyTestUser1 { [Transaction(Etranstype.TRANSTYPE_INI T)] public void TInit()</pre>	<pre>dcluser user Vuser1 transactions TInit : begin; TMain : 1; TMain2 : 5; TEnd : end;</pre>

C# Code	BDL Script
<pre> { [Transaction(Etranstype.TRANSTYPE_MAI N)] public void TMain() { } [Transaction(Etranstype.TRANSTYPE_MAI N, 5)] public void TMain2() { } [Transaction(Etranstype.TRANSTYPE_END)] public void TEnd() { } } </pre>	<pre> var hVuser1 : number; dcltrans transaction TInit begin hVuser1:= DotNetLoadObject("../MyTestUser1"); 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; </pre>

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	BDL Script
<pre> [VirtualUser("Vuser1")] [Transaction(Etranstype. </pre>	<pre> dcluser transaction TMain </pre>

C# Code	BDL Script
<pre>TRANSTYPE_MAIN)] public void TMain() { } [TestMethod] public void Method1() { } [TestMethod] public void Method2() { }</pre>	<pre>begin DotNetCallMethod(hVuser1 ,"TMain"); DotNetCallMethod(hVuser1 ,"Method1"); DotNetCallMethod(hVuser1 ,"Method2"); end;</pre>
<p>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.</p>	

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

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	BDL Script
<pre>[Transaction(Etranstype. TRANSTYPE_MAIN)] public string TMain(string s, int n) { return s + n.ToString(); }</pre>	<pre>dcltrans transaction Tmain var sReturn : string; begin DotNetSetString(hVuser1, "stringvalue"); DotNetSetInt(hVuser1, 123); DotNetCallMethod(hVuser1 ,"TMain"); DotNetGetString(hVuser1, sReturn, sizeof(sReturn)); end;</pre>

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

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 Silk Performer are implemented in `PerfDotNetFW.DLL`, which is a .NET assembly that comes with Silk Performer. The methods are static methods in the `SilkPerformer.Bdl` class. As Silk Performer is an imported namespace you can call the methods with `Bdl.<MethodName>.PerfDotNetFW.dll` is referenced by default when you create a Silk Performer .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
<ul style="list-style-type: none">• <code>AttributeSet</code>• <code>AttributeGet</code>	Setting and getting attribute values.
<ul style="list-style-type: none">• <code>MeasureStart</code>• <code>MeasureStop</code>• <code>MeasurePause</code>	Measure functions.

Function	Description
<ul style="list-style-type: none"> MeasureResume MeasureInc MeasureIncFloat MeasureGet MeasureSetBound MeasureTimeseries MeasureOnOff 	
<ul style="list-style-type: none"> 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: <ul style="list-style-type: none"> .ERR .LOG .RPT
<ul style="list-style-type: none"> 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_IIOPI*, *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 Silk Performer.

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 Silk Performer .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 Silk Performer. 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 Silk Performer, 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 Silk Performer 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 Silk Performer, you can have an `app.config` file in your project directory to specify your own configuration file settings. Silk Performer checks for this configuration file and merges the content into an automatically generated `perfrun.exe.config` file. To make the settings in the `app.config` file available to all agent machines, you need to add this file to the **Data Files** folder in the **Project** menu tree.

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

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

For backward compatibility, when Silk Performer 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 Silk Performer generates those nodes.

With Silk Performer you can provide a fully-configured `app.config` file with all possible configuration sections. Silk Performer 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 Silk Performer .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 Silk Performer does not support this feature.

Random Variables - Virtual User Classes

The Silk Performer 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	BDL Script
<pre>[Transaction(Etranstype. TRANSTYPE_MAIN)] public void TMain1() { }</pre>	<pre>dcltrans transaction TMain1 begin DotNetCallMethod(hVuser1 , "TMain1");</pre>
<pre>[TestMethod] public void TestMeth1() { }</pre>	<pre>DotNetCallMethod(hVuser1 , "TestMeth1"); end; transaction TMain2 begin</pre>
<pre>[Transaction(Etranstype. TRANSTYPE_MAIN)] public void TMain2() { }</pre>	<pre>DotNetCallMethod(hVuser1 , "TMain2");</pre>
<pre>[TestMethod] public void TestMeth2() { }</pre>	<pre>DotNetCallMethod(hVuser1 , "TestMeth2");</pre>
<pre>[TestMethod] public void TestMeth3() { }</pre>	<pre>DotNetCallMethod(hVuser1 , "TestMeth3"); end;</pre>

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
<pre>[Transaction(Etranstype. TRANSTYPE_MAIN)] [TestAttribute("Attr", "Value1")] public void TMain1() { } [TestMethod] [TestAttribute("Attr2", "Value2")] [TestAttribute("Attr3", "Value3")] public void TestMeth1() { }</pre>	<pre>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;</pre>

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. Silk Performer does not support arrays.

Methods with Parameters	
C# Code	BDL Script
<pre>[Transaction(Etranstype. TRANSTYPE_MAIN)] public object TMain1(string s, int n) { }</pre>	<pre>dcltrans transaction TMain1 var hReturn : number; begin DotNetSetString("stringv alue"); DotNetSetInt(123); DotNetCallMethod(hVuser1 , "TMain1"); DotNetGetObject(hReturn) ; end;</pre>

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

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 Silk Performer 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
<pre>[Transaction(Etranstype. TRANSTYPE_MAIN)] [return:BdlParameter("sName")] public string TMain() { return "Andi"; } [TestMethod] public void Hello(string sName) { Bdl.Print("Hello " + sName); }</pre>	<pre>dcltrans transaction Tmain var sName : string; begin DotNetCallMethod(hVuser1 , "TMain"); DotNetGetString(hVuser1, sName, sizeof(sName)); DotNetSetString(hVuser1, sName); DotNetCallMethod(hVuser1 , "Hello"); end;</pre>

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.

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

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
<pre>[Transaction(Etranstype. TRANSTYPE_MAIN)] public void TMain() { } [TestMethod] public string Test1(int nParam) { }</pre>	<pre>dclfunc function Vuser_Tmain(hObject: number) begin DotNetCallMethod(hObject , "TMain"); end; function Vuser_Test1(hObject: number;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;</pre>

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 Silk Performer. 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 Silk Performer 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 Silk Performer project. You can do this through the **Add Dependencies** dialog box of the Silk Performer 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 Silk Performer dialog boxes. Open the dialog boxes through the **Profile Settings** and **System Settings** menu entries of the Silk Performer menu.

The following profile settings are specific to .NET:

- **Traffic redirection**

Option	Description
--------	-------------

Route HTTP .NET through Silk Performer web engine	If this option is enabled, all HTTP/HTTPS network traffic that is generated by .NET components is routed over the Silk Performer 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.
--	---

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 users 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.
---	---

- **Framework version**

Option	Description
--------	-------------

Version	Select the Microsoft .NET Framework version for your script replay. Silk Performer automatically populates this list with the framework version that you have installed on your computer.
----------------	---

Web Settings

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

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

If you are testing Web services through a generated Web-service proxy class, you can route the generated network traffic over the Silk Performer 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 Silk Performer 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 Silk Performer 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 *Silk Performer Help*.

Try Script Runs

You can launch a Try Script run from the Silk Performer 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 Silk Performer

Once you have completed the implementation of the .NET test driver, you can run tests from within Silk Performer. To do this, open Silk Performer from the Silk Performer menu. The project file is opened in Silk Performer 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 Silk Performer project along to other employees who are more familiar with Silk Performer. 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 Silk Performer .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 Silk Performer .NET Framework can route Web traffic generated by .NET components through the Silk Performer 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 Silk Performer 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 Silk Performer .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 Silk Performer 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 **Response** tab of the node in TrueLog Explorer for the *Response SOAP Envelope*, or click the **Request** 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 Silk Performer .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 Silk Performer 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 Silk Performer 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)

Parameter	Description
sTimer	<p>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.</p> <p>If you haven't loaded the assembly where sTypeName is implemented you can specify the assembly file here and it will be loaded. Assemblies are normally loaded during DotNetLoadObject.</p> <p>The basic .NET assembly (mscorlib) is always loaded - so you can access all static methods of the basic classes.</p> <p>(optional)</p> <p>If defined – a custom timer will be generated to measure the execution time of the method call.</p>

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
<code>hObject</code>	Handle to a .NET Object to set the parameter for the next <code>DotNetCallMethod</code> call or <code>DOTNET_CONSTRUCTOR</code> to set the parameter for the next <code>DotNetLoadObject</code> call.
<code>bParam</code>	Boolean value that should be passed as parameter to the next <code>DotNetCallMethod</code> 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 <code>DotNetCallMethod</code> call or <code>DOTNET_CONSTRUCTOR</code> to set the parameter for the next <code>DotNetLoadObject</code> call.
nParam	Integer value that should be passed as parameter to the next <code>DotNetCallMethod</code> 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
<code>hObject</code>	Handle to a .NET Object to set the parameter for the next <code>DotNetCallMethod</code> call or <code>DOTNET_CONSTRUCTOR</code> to set the parameter for the next <code>DotNetLoadObject</code> call.
<code>hParam</code>	.NET Object handle that should be passed as parameter to the next <code>DotNetCallMethod</code> 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
<code>hObject</code>	Handle to a .NET Object.
<code>sReturn</code>	String buffer that will receive the return value of the last <code>DotNetCallMethod</code> call.
<code>nRetLen</code>	Size of the string buffer passed in <code>sReturn</code> (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. The returned object handle has to be freed with DotNetFreeObject.

Include file

DotNetAPI.bdh

Syntax

```
DotNetGetObject( in hObject : number,
                 in nParamIx : number optional ): number;
```

Return value

- object handle if successful
- 0 otherwise

Parameter	Description
hObject	Handle to a .NET Object
nParamIx	Optional: If specified, gets the parameter index, else gets the last return value.

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");

    hReturn := DotNetGetObject( hObject );

    DotNetFreeObject(hObject);
    DotNetFreeObject(hObject2);
    DotNetFreeObject(hReturn);
  end TMain;

```

Index

.NET

- available BDL functions 46
 - BDL code generation engine 34
 - calling BDL functions 30
 - configuration files 33
 - creating projects 15, 25
 - debugging 33
 - declaring attributes 28
 - defining input parameters 29
 - defining transactions 26
 - defining virtual users 26
 - development workflow 25
 - exception handling 32
 - helper classes 13
 - profile settings 41, 42
 - random variables 32
 - system settings 41
 - test methods 35
 - virtual user 34
 - writing test drivers 25
- .NET code
- intermediate code 13
- .NET components
- testing 11, 15
- .NET Explorer
- .NET message sample 8
 - .NET Remoting sample 8
 - generating .NET test drivers 46
 - overview 5, 6
- .NET framework
- overview 11
 - testing .NET components 11
 - testing Web services 11
- .NET Framework
- .NET message sample 8
 - .NET Remoting sample 8
 - custom attributes
 - code sample 23
 - generated BDF script example 23
 - negative testing 22
 - unit test standards 22
 - dependencies
 - adding 19
 - ExtensionMicrosoft Visual Studio 6, 13
 - instantiating client proxy objects 17
 - option settings 20
 - overview 5, 11, 12
 - routing Web Service calls 19
 - running tests 20
 - Try Script runs 17
 - understanding 11
 - version 41
 - Web Service calls 18
 - Web service client proxies, creating 16
- .NET projects
- creating 25
- .NET Remoting
- sample project 9

- .NET Remoting objects
 - sample 8
- .NET services
 - testing 25
- .NET test driver
 - data files 41
 - running tests 44
- .NET Test Driver
 - preparations for testing 41
 - testing 41
- .NET test drivers
 - passing data to BDL script 28
 - writing 25
- .NET testing
 - provided tools 6

B

- BDL functions
- .NET 46
 - calling from .NET 30
 - constant values 31
 - overview 30
- BDL scripts
- passing data to .NET test drivers 28
- BdlParameter
- Code Generation Engine 38

C

- calling
- Web service methods 45
- client proxies
- instantiating objects 44
- client proxy objects
- instantiating 44
- Code Generation Engine
- BdlParameter 38
 - input parameters 36
 - passing parameters 38
 - return values 36
 - test attributes 36
- configuration files
- .NET 33
- creating
- .NET projects 15
 - Web service client proxies 44

D

- debugging
- .NET 33
- declaring attributes
- .NET 28
- DotNetCallMethod 48
- DotNetFreeObject 47
- DotNetGetBool 55
- DotNetGetFloat 55

- DotNetGetInt 56
- DotNetGetObject 57
- DotNetGetString 54
- DotNetLoadObject 46
- DotNetSetBool 51
- DotNetSetFloat 50
- DotNetSetInt 52
- DotNetSetObject 53
- DotNetSetString 49

E

- exception handling
 - .NET 32
- exploring
 - results 43
- exploring results
 - Visual Studio 46
- extension
 - Visual Studio 14

H

- helper classes
 - .NET 13

I

- input parameters
 - defining 29
- intermediate code
 - .NET code 13

J

- Java Explorer
 - overview 5, 6
 - RMI sample 8
- Java Framework
 - overview 5
 - RMI sample 8
 - sample project 10
- Java testing
 - provided tools 6
- JDBC test client 10

L

- launching
 - Try Script runs 43

M

- Microsoft Visual Studio, Extension
 - overview 6, 13
- MS Visual Studio, Add-In 5, 7

O

- options

- Generating BDH for .NET Method Calls 40
- generating BDL functions for .NET method calls 40
- generating project attributes 41
- Generating Timers for DotNetCallMethod 39
- tests 42
- Visual Studio .NET 39

P

- parameters
 - intelligent passing 38
- profile settings
 - .NET 41

R

- random variables
 - virtual user classes 34
- results
 - exploring 43
- RMI sample project 10
- RMI samples, Java 8
- routing
 - Web service traffic 45
- running tests
 - .NET 44

S

- SOA Edition
 - overview 5
- system settings
 - .NET 41, 42

T

- test attributes
 - Code Generation Engine 36
- test drivers
 - writing for .NET 25
- test methods
 - .NET 35
 - defining 27
- testing
 - .NET components 15
 - .NET services 25
- testing .NET components
 - .NET framework 11
- testing Web services
 - .NET framework 11
 - Visual Studio 44
- tests
 - options 42
- transactions
 - defining in .NET 26
 - virtual user classes 35
- Try Script runs
 - .NET add-on prerequisites 43
 - launching 43

V

- virtual user
 - .NET 34
- virtual user classes
 - random variables 34
 - transactions 35
- virtual users
 - defining in .NET 26
- Visual Studio
 - exploring results 46
 - extension 14
 - testing Web services 44
- Visual Studio .NET
 - options 39
- Visual Studio Extension

- installing 14

W

- Web service methods
 - calling 45
- Web services
 - creating client proxies 44
 - routing traffic 45
- Web Services
 - .NET message sample 8
 - .NET Remoting sample 8
 - publicly accessible demonstration servers 7
 - sample project 9