Testing Framework version 1.0 Component Specification

1.Design

This component simplifies the process of testing server-driven applications by enabling the application to be built, deployed, tested, and shut down entirely within a NAnt build script. It provides analogous functionality for database servers, web servers (with a help of NUnitAsp tests), and the extensibility to provide the same support for other types of servers.

For a web application that uses an IIS ASP.NET server and SQL Server 2000 database, the test framework provides a simple way to specify how to configure and initialize both of these servers. Testing the application is then simply a matter of calling the framework, which initializes the servers, runs a test suite, and cleans up after itself.

This component provides the support for an IIS ASP.NET server using the appropriate Windows service. The support for database servers should be provided by the other component.

NUnit extension

The test framework should include a NUnit extension. But current implementation doesn't provide any NUnit extensions, because for the web-server they were provided by the NUnitAsp. The NUnitAsp provides a powerful framework of classes, which consists of one NUnit TestCase class and multiple classes for testing particular parts of the ASP pages, wrapping around all these classes is useless, because using another back-end will bring a completely different framework, which will be impossible to wrap.

So the design has decided that the NUnit extensions should be provided by the particular back-ends. It also seems that such extensions will be needed only for application servers.

NAnt integration

The component provides NAnt task to enable tests defined in the test framework to be driven by an NAnt build script. This task contains all of the configuration information necessary to perform its functions. There is no external configuration. Ant task provided by the component is **serverapptest.** The documentation for this task is provided with the corresponding class: *ServerAppTestTask.*

Server Initialization / Clean-up

The test framework provides a mechanism for starting a server if necessary, and initializing it with the proper data. For a SQL database server, this means starting the database server, creating the necessary tables and possibly filling them with test data. For an IIS ASP.NET server, this means deploying the web application and starting the IIS server. It is possible to initialize several servers for one NAnt server application testing task.

If the particular server was started and (or) initialized by the test framework, it will be stopped and (or) cleaned-up after running all the tests.

An element representing server in the NAnt build file has two boolean attributes for controlling server initialization. These attributes are “startserver” and “initserver”. The “startserver” attribute specifies if the server should be started by the framework. The “initserver” attribute specifies if the server should be initialized with the data by the framework.

Pluggable testing back-ends

The new type of server can be added or the existing type of server may be implemented, using different testing back-end. These actions do not require any changes to the existing framework. The names of pluggable implementations are specified in the NAnt build file.

1.1Design Patterns

The Strategy Pattern. The Server class is an abstract class representing the server. It contains the methods for starting/stopping server, etc. The derived classes can be easily plugged in to the ServerAppTestTask, to provide different implementations of the server control methods.

The Factory Method Pattern. The ServerElement class uses this pattern for creating appropriate instances of classes derived from Server class.

1.2Industry Standards

None

1.3Required Algorithms

None were required, but some complicated pieces of code are provided here.

1.3.1 Executing RunServerAppTest Ant task

```
// Startup/Initialize the servers
foreach (ServerElement element in this.servers) {
      Server server = element.GetServerInstance();
      if (element.StartServer)
                  server.StartUp();
}

// Note, if any of the servers fails to start (ServerException is caught),
// the servers that were started should be stopped
// The order of stopping the servers is reverse
// to that one of starting the servers
// All the exception caught while processing should be wrapped in ServerTaskException
// Note, that the complete list of caught exceptions should be supplied


// Startup/Initialize the servers
foreach (ServerElement element in this.servers) {
      Server server = element.GetServerInstance();
      if (element.InitServer)
                  server.InitData();
}
// Note, if any of the servers fails to initialize (ServerException is caught),
// the servers that were started should be stopped,
// the servers that were initialized should be deinitialized
// The order of stopping/deinitializing the servers is reverse
// to that one of starting/initializing the servers
// All the exception caught while processing should be wrapped in ServerTaskException
// Note, that the complete list of caught exceptions should be supplied



// Run the tests, just class the method of superclass
super.ExecuteTask();

// Deinitialize/shutdown the servers
for (int i = this.servers.GetLength() - 1; i >= 0; --i) {
      Server server = this.servers[i].GetServerInstance();
      if (this.servers[i].InitServer)
                  server.CleanUpData();
      if (this.servers[i].StartServer)
                  server.ShutDown();
}

// Note, if any of the servers fails to deinitialize/stop (ServerException is caught),
// the servers the remaining servers should be stopped, but the exception will be thrown.
// All the exception caught while processing should be wrapped in ServerTaskException
// Note, that the complete list of caught exceptions should be supplied
```

1.3.2 Start the server, the default web application server implementation

```
string serviceName;

// Get the server service name
if (Options[SERVICE_NAME] == null, or  empty string) {
    serviceName = DEFAULT_SERVICE_NAME;
} else {
    serviceName = Options[SERVICE_NAME];
}

// Create an appropriate ServiceController
ServiceController serviceController = new ServiceController(serviceName);

// Get timeout
int timeout;
if (Options[TIMEOUT] == null, or  empty string) {
    timeout = DEFAULT_TIMEOUT;
} else {
    timeout = Int32.Parse(Options[TIMEOUT]);
}

// Check if the service was is running
if (serviceController.Status == ServiceContollerStatus.Running) {
    // If it is running, the ServerException should be thrown
    // If the server should not be started,
     // the "startserver" attribute of the corresponding XML element should be "false"
}


// Start the service
if (serviceController.Status == ServiceContollerStatus.Paused) {
    ServiceController.Continue();
} else {
    ServiceController.Start();
}

// Wait for the service to reach the desired status, or the timeout expired
service.WaitForStatus(ServiceControllerStatus.Running, TimeSpan.ForMilliseconds(timeout));
// Note, that the TimeoutException should be caught and wrapped into the ServerException
```

1.3.2 Stop the server, the default web application server implementation

```
// Allmost analogous to the start server algorithm

string serviceName;

// Get the server service name
if (Options[SERVICE_NAME] == null, or  empty string) {
    serviceName = DEFAULT_SERVICE_NAME;
} else {
    serviceName = Options[SERVICE_NAME];
}

// Create an appropriate ServiceController
ServiceController serviceController = new ServiceController(serviceName);

// Get timeout
int timeout;
if (Options[TIMEOUT] == null, or  empty string) {
    timeout = DEFAULT_TIMEOUT;
} else {
    timeout = Int32.Parse(Options[TIMEOUT]);
}

// Stop the service
if (serviceController.CanStop) {
    ServiceController.Stop();
} else {
    // throw ServerException here
}

// Wait for the service to reach the desired status, or the timeout expired
service.WaitForStatus(ServiceControllerStatus.Stopped, TimeSpan.ForMilliseconds(timeout));
```

// Note, that the TimeoutException should be caught and wrapped into the ServerException

1.4 Component Class Overview

Class ServerAppTestTask

This class derives from *NUnit2Task*. This class represents a NAnt task that extends the optional NUnit2 task to provide support for testing using this Component.
This is the main “worker” NAnt task f this component.

Class ServerElement

This class is derived from the Element class. This class represents "server" XML element to be used in the NAnt build file. This element should be used inside the "**serverapptest**" NAnt task. That task is represented by the *ServerAppTestTask* class.

Class Server

This class is an abstract base class for all the classes representing a server to be used when running tests. It stores some attributes passed from the *ServerElement* class.
The appropriate properties are provided for accessing the attributes.

Class ApplicationServer

This class is derived from Server class and is the base class
for all classes representing an application server to be used while running tests.

Class DataServer

This class is derived from Server class and is the base class for all classes representing data server to be used while running tests.
This class is a base for another component which will provide testing ability for database servers.

Class DefaultWebApplicationServer

This class is derived from ApplicationServer class. It represents web application server to be used while running tests. It supports the Microsoft IIS server.
Really it should support almost any service-based servers.

1.5 Component Exception Definitions

Exception ServerException

This exception is derived from ApplicationException. This exception is thrown by the classes derived from Server class, when any error happen while processing.
It is also thrown by the ServerElement class, when it fails to create a corresponding class derived from Server.

Exception ServerTaskException

This exception is derived from BuildException. It is thrown by the ServerAppTestTask when any exceptions are caught when executing the task.

1.6 Thread Safety

This component is not thread-safe, because all of its classes are mutable. But there is no need for it to be thread-safe, because it will be used only in the NAnt builds, and so it will be NAnt framework's responsibility not to access class instances by the multiple threads, or to synchronize the access. They are most likely to be used by the single thread.

2. Environment Requirements

2.1 Environment

- Windows 2000/XP/2003
- C# 1.1
- IIS 6.0

2.2 TopCoder Software Components

None used.

The Configuration Manager component was recommended, but wasn't used because all the necessary configuration can be specified in the NAnt build file.

2.3Third Party Components

- NAnt 0.84: http://nant.sourceforge.net
- NUnit 2.2.5: http://nunit.sourceforge.net
- NUnitAsp 1.5.1: http://nunitasp.sourceforge.net

3.Installation and Configuration

3.1Package Names

TopCoder.Test.Framework

3.2Configuration Parameters

None required

3.3Dependencies Configuration

None required

4.Usage Notes

4.1Required steps to test the component

- Extract the component distribution.
- Follow Dependencies Configuration.
- Execute "ant test" within the directory that the distribution was extracted to.

4.2Required steps to use the component

None, see demo section

4.3Demo

```
// Sample unit test, using NUnitAsp,
// for more detailed tutorial visit http://nunitasp.sourceforge.net

[TestFixture]
public class SampleTest : WebFormTestCase
{
        // This method performs initialization before every test
        protected override void SetUp()
        {
                // Open the page to test
                Browser.GetPage("http://localhost/some_path/page_to_test.aspx");
                // Do something other here
        }

        // This method performs clean-up after every test
        protected override void SetUp()
        {
                // Do something here
        }

        // Test something
        [Test]
        public void TestLayout()
        {
                // Do something here,
                // then use the AssertXXX methods to chech the results
        }
}



<!-- Sample NAnt build file fragment -->

<!-- The build file should also contain definitions of the targets, compiling the application, etc.
-->

<!-- Load the NAnt task-->
<loadtasks assembly="TestFramework.Dll" />


<!-- This NAnt task starts the servers, runs the tests, and then stops the servers -->
<serverapptest printsummary="yes" failureproperty="tests.failed" todir="${test.reports.dir}">


        <!-- This element describes the application server using IIS server -->
        <!-- By default the server is started and initialized with data (app is deployed) -->

        <server class="DefaultWebApplicationServer">
                <!-- The name of server service -->
                <option name="service_name" value = "w3svc">
                <!-- The directory to deploy application to-->
                <option name="deployment_dir" value = "${deployment_dir}"/>
                <!-- The timeout is set to one minute in milliseconds-->
                <option name="timeout" value = "60000"/>

                <sourcefiles>
                        <!-- This should be a fileset specifing the files to deploy-->
                </sourcefiles>
        </server>

        <!-- This element describes the database server (It's not provided by this component) -->
        <!-- The server is not started but is initialized with data -->
        <server class="SomeDataBaseServer" startserver="no" initsever="yes">
                <!-- Specifying the port to connect to -->
                <option name = "port" value = "some_port_number"/>
                <!-- Specifying the address of macine running the server -->
                <option name = "address" value = "some_server_address">
                <!-- Specifying the SQL script used to init the test data -->
                <option name = "sql_init_script" value="${sql_scripts_dir}/init.sql">
                <!-- Specifying the SQL script used to clean up the test data -->
                <option name = "sql_cleanup_script" value="${sql_scripts_dir}/cleanup.sql">
                <!-- Specifyinf the credentials used to authentificate to the server -->
                <credentials username="topcoder" password="some_password" />
        </server>

        <formatter type="xml"/>
        <test assemblyname="NameOfAssemblyContainingTests.Dll" />

</serverapptest>
```

5.Future Enhancements

- Implement support for other types of servers, using different back-ends.