Microsoft

Enterprise Library Test Guide

patterns & practices

Microsoft®

Enterprise Library Test Guide

patterns & practices

ISBN 0-7356-2389-9

Contents

Introduction

Enterprise Library Test Guide provides guidance to testers and programmers who have extended or modified an Enterprise Library component and who want to apply functional testing techniques to their work. An Enterprise Library component can be an Enterprise Library application block or the Enterprise Library Core.

This guide works by example. It explains how each component was tested by Microsoft test teams. You can use the same techniques and adapt the templates and checklists that are included in the guide to suit your needs. The functional tests that the Microsoft test teams used to test the Enterprise Library are included with this documentation. You can alter them to test your own code.

Scope

This guide provides guidance for performing functional tests on Enterprise Library components. Functional tests include design reviews, code reviews, and automated testing. The guide also describes how to test the components to determine whether they follow globalization and security best practices in addition to meeting performance and scalability requirements.

Audience

This guidance is intended for software testers and test managers who must test Enterprise Library components. It is also for programmers who must extend or modify an existing Enterprise Library component.

To get the most benefit from this guidance, you should have an understanding of the following technologies:

- Microsoft Visual C#
- Microsoft .NET Framework 2.0
- Visual Studio Team System automated tests or NUnit

System Requirements

To run the automated tests that accompany the Enterprise Library, you need the following:

- Enterprise Library–January 2006
- Microsoft .NET Framework 2.0
- Microsoft Visual Studio 2005 Team Suite if you want to run the automated Visual Studio Team System tests
- NUnit 2.2 or later if you want to run automated NUnit tests
- Oracle 9i client if you want to test the Data Access Application Block with an Oracle database
- SQL Server 2005 Express Edition (SQL Server Express) if you want to test the Data Access Application Block with a SQL database
- Microsoft Windows Message Queuing if you want to test the Logging Application Block
- Authorization Manager (AzMan) if you want to test the Security Application Block. By default, AzMan is installed with Windows Server 2003. If you need to install it on a computer running Windows XP or Windows 2000, you can download the *Windows Server 2003 Administration Pack* on MSDN.

Note: The SQL Server Express and Oracle 9i database requirements may also matter if you are testing the Exception Handling Application Block or the Logging Application Block. The Exception Handling Application Block requires the Logging Application Block when it uses the logging handler. If you configure the Logging Application Block to use the database trace listener, it requires the Data Access Application Block. For more information, see the Enterprise Library documentation for the Logging Application Block and for the Data Access Application Block.

The automated tests assume that Enterprise Library is installed in the default location. If it is not in the default location, open the solution files. Ignore any errors about project files that cannot be found. Change the project references to the correct location.

Functional Testing

Functional testing involves the following:

- Design reviews
- Code reviews
- Automated tests

This section provides an overview of functional testing as it was used with the Enterprise Library. To see how a specific application block or the Enterprise Library Core was tested, refer to the relevant chapter. In general, there are four steps to functional testing:

1. Create the test cases for the design and the code.
2. Perform design reviews to verify that the design addresses all functional requirements.
3. Perform code reviews to verify that the code addresses all functional requirements.
4. Perform automated tests to validate that the application is functioning as it should and to simulate situations such as multiple users concurrently using the application.

Creating Test Cases

Design test cases specify the requirements that a design must fulfill for the application to perform the way it is expected to. Each test case should be satisfied by a specific part of the design. For example, if the application is meant to be extensible, there must be a specific way that the application can be extended, such as an interface or a base class.

When you write the design test cases, have a list of requirements and any design-related documents, such as architectural diagrams and class diagrams, available so you can incorporate this information into the test cases. The Microsoft test teams found it helpful to create tables that listed each design requirement and its priority. Table 1 is an example of some of the design cases for the Cryptography Application Block.

Table 1: Cryptography Application Block Design Test Cases

Priority	Design Test Case
High	Verify that the symmetric algorithm providers and the hash providers are extensible.
High	Verify that there is a consistent approach to creating symmetric algorithm providers and hash providers.
High	Verify that there is a facade that mediates between the client code and the application block's cryptographic functions such as encryption, decryption, and hashing.

After you write the design test cases, do the same for the code. The code must implement the design and, usually, it must conform to some set of guidelines and best practices. Try to include a variety of issues when you write the code test cases. Examples of these issues include the following:

- Performance
- Security
- Globalization and localization
- Exception management

Table 2 is an example of some of the test cases for the Cryptography Application Block.

Table 2: Cryptography Application Block Code Test Cases

Priority	Code Test Case
High	Verify that the **Cryptographer** facade exposes all public members as static and supports methods for encryption, decryption, and hashing.
High	Verify that the **Cryptographer** facade uses the **SymmetricCryptoProviderFactory** class and the **HashProviderFactory** class to create the cryptography providers.

Performing Design Reviews

After you write the design test cases, you can review the design to determine whether the test cases are satisfied. The Microsoft test team used checklists that had a column for the test cases, a column titled "Implemented?," and a column for the features that implement the design.

Table 3 is an example of a design checklist for the Cryptography Application Block.

Table 3: Cryptography Application Block Design Verification

Design Test Case	Implemented?	Feature that Implements Design
Verify that the symmetric algorithm providers and the hash providers are extensible.	Yes	The **ISymmetricCryptoProvider** interface allows users to implement or extend a configurable symmetric provider. The **IHashProvider** interface allows uses to implement or extend a hash provider.
Verify that there is a consistent approach to creating symmetric algorithm providers and hash providers.	Yes	The **SymmetricCryptoProviderFactory** class is the factory that creates the **SymmetricProvider** objects. The **HashProviderFactory** class is the factory that creates the **HashProvider** objects.
Verify that there is a facade that mediates between the client code and the application block's cryptographic functions such as encryption, decryption and hashing.	Yes	The **Cryptographer** class is a facade that acts as the interface between the client code and the application block.

Performing Code Reviews

Code reviews happen periodically throughout the development cycle. The process is similar to that of the design reviews. Each test case must be satisfied by a specific part of the implementation. The Microsoft test team used checklists for the code review that were similar to the ones used for the design review. Table 4 is an example of the code checklist for the Cryptography Application Block.

Table 4: Cryptography Application Block Code Verification

Code Test Case	Implemented?	Feature That is Implemented

Verify that the **Cryptographer** facade exposes all public members as static and supports methods for encryption, decryption, and hashing.	Yes	The **Cryptographer** class is a facade that exposes the **CreateHash** method to compute the hash value of plain text, the **CompareHash** method to compare plain text with a hash value, the **EncryptSymmetric** method to encrypt plain text, and the **DecryptSymmetric** method to decrypt a symmetrically encrypted secret. These methods are shown in the following code.
		`public static byte[] CreateHash(string hashInstance, byte[] plaintext) {}`
		
		`public static bool CompareHash(string hashInstance, string plaintext, string hashedText) {}`
		
		`public static string EncryptSymmetric(string symmetricInstance, string plaintext) {}`
		
		`public static string DecryptSymmetric(string symmetricInstance, string ciphertextBase64) {}`

Verify that the **Cryptographer** facade uses the **SymmetricCryptoProviderFactory** class and the **HashProviderFactory** class to create the cryptography providers.	Yes	The client code calls static methods on the **Cryptographer** class to create hashes, compare hashes, encrypt data, and decrypt data. Each static method instantiates a factory class and passes the configuration source to the factory class's constructor. The factory uses the configuration data to determine the type of provider to create. The following code demonstrates how the **Cryptographer.EncryptSymmetric** method calls the **SymmetricCryptoProviderFactory** class to create a symmetric provider. The process is similar for hash providers.
		`public static byte[] EncryptSymmetric(string symmetricInstance, byte[] plaintext)`
		`{`
		`…`
		`SymmetricCryptoProviderFactory factory = new SymmetricCryptoProviderFactory(ConfigurationSourceFactory.Create());`
		`…`
		`}`

Running the Automated Tests

Each Enterprise Library application block has two solution files associated with it that contain the automated tests. The solution file that ends with .VSTS.sln contains the Visual Studio Team System automated tests. The solution file that ends with .NUnit.sln contains the NUnit automated tests. For example, the Data Access Application Block uses the DAAB Functional Tests.VSTS.sln file and the DAAB Functional Tests.NUnit.sln file for its automated tests.

The automated tests for the Enterprise Library components require some preliminary setup actions, such as running scripts, before you can execute them. These preliminary actions are described later.

Setting Up the Enterprise Library Core Automated Tests

To set up the Enterprise Library Core automated tests, first compile the solution file that is in the Core Functional Tests folder. Next, run the Installer tool (installutil.exe) on the test assembly that is named ConfigurationCoreTests.dll and is located in the bin folder inside the Core Functional Tests folder. The Installer tool registers the performance counters and the WMI events. Open a command window and type the following.

```
installutil –I ConfigurationCoreTests.dll
```

There are no scripts for the Enterprise Library Core automated tests. The automated tests are in the Core Functional Tests folder.

Setting Up the Caching Application Block Automated Tests

To set up the Caching Application Block automated tests, you must run the script that creates a database backing store. To do this, click **Start**, point to **All Programs**, point to **Microsoft patterns & practices**, point to **Enterprise Library–January 2006**, point to **Application Blocks for .NET**, point to **Caching Application Block**, and then click **Create Database Backing Store**. The automated tests are in the Caching Functional Tests folder.

Once the database is created, run the "Install instrumentation" script. To do this, click **Start**, point to **All programs**, point to **Microsoft patterns & practices**, and then click **Install Instrumentation**. Once installed, you need to create a file called "FileforCaching,txt" in the \\Enterprise Library 2.0 Functional Tests\Caching Functional Tests\CachingCoreTests folder and create a file called "FileforCachinginDB.txt" in the \\Enterprise Library 2.0 Functional Tests\Caching Functional Tests\CachingDatabaseTests folder.

Setting Up the Cryptography Application Block Automated Tests

To set up the Cryptography Application Block automated tests, you must run the test script. To do this, run the CryptoTestScript.bat script that is located in the TestScript folder. This folder is in the Cryptography Functional Tests folder. The automated tests are also in this folder.

Setting Up the Data Access Application Block Automated Tests

To set up the Data Access Application Block automated tests, you must run the appropriate test script. If you are running test cases that use a SQL database as the backing store, run the SetupTestDB.bat script that is located in the TestScript folder. This folder is in the DAAB Functional Tests folder.

If you are running test cases that use an Oracle database, you must first create a database named "entlib," a user named "testuser," and a password that is "testuser." Next, perform the following procedure to create the database tables and stored procedures.

► **To run the scripts do the following**

1. Grant permissions to create session to testuser;
2. Grant permissions to create table to testuser;
3. Grant permissions to create procedure to testuser;

► **To create the Oracle tables and stored procedures**

1. Run the scripts in the Table folder to create the tables.
2. Run the scripts in the Data folder to create the data for the tables.
3. Run the SP scripts in the SP folder to create the stored procedures.

The automated tests, which you can use for both the SQL database and the Oracle database, are in the DAAB Functional Tests folder.

Setting Up the Logging Application Block Automated Tests

To set up the Logging Application Block automated tests, first create a private message queue named "EntLibTest" and a public message queue named "EntLibTest Public." Next, run the script that creates a logging database. To do this, click **Start**, point to **All Programs**, point to **Microsoft patterns & practices**, point to **Enterprise Library–January 2006**, point to **Application Blocks for .NET**, point to **Logging Application Block**, and then click **Create Logging Database**. The automated tests are in the Logging Functional Tests folder.

Setting Up the Exception Handling Application Block Automated Tests

The Exception Handling Application Block automated tests use the same script as the Logging Application Block to create a database backing store. Follow the instructions for running the script that creates a logging database in Setting Up the Logging Application Block Automated Tests. The automated tests are in the Exception Handling Functional Tests folder.

Setting Up the Security Application Block Automated Tests

The Security Application Block automated tests use the same script as the Caching Application Block to create a database backing store. Follow the instructions for running the script that creates a caching database in Setting Up the Caching Application Block Automated Tests. The automated tests are in the Security Functional Tests folder.

Contents of the Enterprise Library Test Guide

The *Enterprise Library Test Guide* contains the following chapters:

- *Testing the Enterprise Library Core*. This chapter explains how the test teams used functional testing to test the Enterprise Library Core.
- *Testing the Caching Application Block*. This chapter explains how the test teams used functional testing to test the Caching Application Block.
- *Testing the Cryptography Application Block*. This chapter explains how the test teams used functional testing to test the Cryptography Application Block.
- *Testing the Data Access Application Block*. This chapter explains how the test teams used functional testing to test the Data Access Application Block.
- *Testing the Exception Handling Application Block*. This chapter explains how the test teams used functional testing to test the Exception Handling Application Block.
- *Testing the Logging Application Block*. This chapter explains how the test teams used functional testing to test the Logging Application Block.
- T*esting the Security Application Block*. This chapter explains how the test teams used functional testing to test the Security Application Block.
- *Testing for Security Best Practices*. This chapter explains how the test teams tested the application blocks to see whether they conformed to security best practices.
- *Testing for Globalization Best Practices*. This chapter explains how the test teams tested the application blocks to see whether they conformed to globalization best practices.
- *Testing for Performance and Scalability*. This chapter explains how the test teams tested the application blocks to see whether they conformed to the performance and scalability requirements.
- *Using the Test Cases*. This chapter includes examples of the different types of bugs the test teams uncovered when they tested the application blocks.

Acknowledgments

The following people created this guide:

- Program management: Mohammad Al-Sabt (Microsoft Corporation)
- Test: Carlos Farre (Microsoft Corporation); Gokulaprakash Thilagar (Infosys Technologies Ltd)
- Documentation: RoAnn Corbisier and Nelly Delgado (Microsoft Corporation); Roberta Leibovitz (Modeled Computation LLC); Tina Burden McGrayne (TinaTech Inc); Claudette Siroky (WadeWare LLC)

Testing the Enterprise Library Core

This chapter explains how functional testing techniques were used to test the Enterprise Library Core. If you want to extend or modify the core, you can use the same techniques and adapt the chapter's templates and checklists to test your own work.

Requirements for the Core

The core has the following requirements:

- The core should include base factory classes that create instance objects and a base factory class that creates singleton objects for types that support them.
- The core should include configuration watchers that monitor the file configuration source and the system configuration source.
- The core should include the ability to bind the instrumentation listeners to the providers at run time.
- The core should include installers that install performance counters and the event log.

These requirements must be incorporated into the design and implemented by the code.

Selecting the Test Cases

The first step in a functional review is to make sure that the design and the code support these requirements. You do this by deciding the test cases that they must satisfy so that the design and the code fulfill all of their requirements.

Table 1 lists the test cases that the core's design must satisfy.

Table 1: Core Design Cases

Priority	Design test case
High	Verify that the core implements a base strategy class that supports the Enterprise Library strategies and that retrieves useful information from the context, such as the configuration source.
High	Verify that the core implements strategy classes that create objects from the configuration source and that bind the instrumentation.

continued

Priority	Design test case
High	Verify that the core implements base factory classes that create instance objects and a base factory class that creates singleton objects for types that support them.
High	Verify that the core provides a façade for a generic object instance building mechanism for the **ObjectBuilder** subsystem.
High	Verify that the core implements configuration watchers that monitor the configuration sources.
High	Verify that the core includes configuration classes that read various types of configuration elements, such as **name**, **type**, and **collection**.
High	Verify that the core includes factory classes that support polymorphism.
High	Verify that the core includes base classes that implement performance counters.
High	Verify that the core includes installer classes that install performance counters and the event log.

After you identify the design issues, you should do the same for the code. Table 2 lists the test cases that the core code must satisfy.

Table 2: Core Code Test Cases

Priority	Code test case
High	Verify that the core caches the configuration reflection data and reuses the reflection data.
High	Verify that the **NameTypeConfigurationElement** class converts the **type** configuration element to the **Type** class.
High	Verify that the core includes a generic helper class that manages custom provider configuration objects.
High	Verify that the configuration sources and the configuration watchers are extensible.
High	Verify that the core's system configuration source and its file configuration source have configuration watchers that monitor changes to their configuration sources. If there are changes, these watchers should notify the appropriate class, such as the **FileConfigurationSourceImplentation** class, to update the configuration data.
High	Verify that the configuration source has configuration watchers that monitor specific sections of the external configuration source. If there are changes, these watchers should notify the appropriate class to update the configuration data.
High	Verify that there are methods to properly dispose of the configuration watchers when the watchers are unregistered from the configuration source.
High	Verify that the instrumentation is only bound to an object when the instrumentation is enabled.
Medium	Verify that the core requests or demands the appropriate code access security permissions to access protected system resources and operations.
High	Verify that the core follows exception management best practices.
High	Verify that the core follows security best practices.
Medium	Verify that the core follows globalization best practices.
High	Verify that the core follows performance best practices.

Verifying the Test Cases

After you identify all the design test cases, you can verify that the design satisfies them. Table 3 lists how each of the design test cases were verified for the Enterprise Library Core.

Table 3: Core Design Verification

Design test case	Implemented?	Feature that implements design
Verify that the core implements a base strategy class that supports the Enterprise Library strategies and that retrieves useful information from the context, such as the configuration source.	Yes	The **EnterpriseLibraryStrategy** class derives from the **BuilderStrategy** class, which is the base class for Enterprise Library strategies. The **EnterpriseLibraryStrategy** retrieves the selected configuration source.
Verify that the core implements strategy classes that create objects from the configuration source and that bind the instrumentation.	Yes	The **ConfiguredObjectStrategy** class retrieves the configuration data and creates the objects. The **InstrumentationStrategy** class injects the instrumentation attachment process into the **ObjectBuilder** subsystem.
Verify that the core implements base factory classes that create instance objects and a base factory class that creates singleton objects for types that support them	Yes	The **NameTypeFactoryBase** class is a generic factory class that creates instance objects. The **LocatorNameTypeFactoryBase** class is a generic factory class that creates singleton object for types that support them.
Verify that the core provides a façade for a generic object instance building mechanism that is based on the ObjectBuilder subsystem.	Yes	The **EnterpriseLibraryFactory** is a façade that provides the generic building mechanism for the **ObjectBuilder** subsystem.
Verify that the core implements configuration watchers that monitor the configuration sources.	Yes	The **ConfigurationFileSourceWatcher** class monitors configuration files. Derivations of the **ConfigurationSourceWatcher** abstract base class monitor alternative configuration sources.
Verify that the core includes configuration classes that read various types of configuration elements such as **name**, **type**, and **collection**.	Yes	The **NamedConfigurationElement** class represents a **ConfigurationElement** object that is a **name** element. The **NamedElementCollection** class is a generic collection of **NamedConfigurationElement** objects. The **NameTypeConfigurationElement** class represents a **ConfigurationElement** object that has both a **name** element and a **type** element. The **NameTypeConfigurationElementCollection** class is a generic collection of **NameTypeConfigurationElement** objects.

continued

Design test case	Implemented?	Feature that implements design
Verify that the core includes factory classes that support polymorphism.	Yes	The **AssemblerBasedCustomFactory** class and the **AssemblerBasedObjectFactory** class are factories that build objects with polymorphic hierarchies that are based on single configuration objects.
Verify that the core includes base classes that implement performance counters.	Yes	The **EnterpriseLibraryPerformanceCounter** class allows applications to maintain both individually named counters and a single counter that represents the total number of all the named counter values.
Verify that the core includes installer classes that install performance counters and the event log.	Yes	The **EventLogInstallerBuilder** class installs the event log sources. The **PerformanceCounterInstallerBuilder** class installs the performance counters.

After the code is implemented, you can review it to see if it satisfies its test cases. Table 4 lists the results of a code review for the core.

Table 4: Core Code Verification

Code test case	Implemented?	Feature that is implemented

Verify that the core caches the configuration reflection data and reuses the reflection data.	Yes	The **ConfigurationReflectionCache** class maintains a dictionary collection that stores the reflection data. The **CreateCustomFactory** method reflects the attributes of the specified type to create a custom factory object. This custom factory is then stored in the cache. This is shown in the following code example.
		`public ICustomFactory GetCustomFactory(Type type)`
		`{`
		`…`
		`exists = typeCustomFactories.TryGetValue(type, out storedObject);`
		`…`
		`if (!exists)`
		`{`
		`storedObject = CreateCustomFactory(type);`
		`lock (typeCustomFactoriesLock)`
		`{`
		`typeCustomFactories[type] = storedObject;`
		`}`
		`}`
		`return storedObject;`
		`}`

Code test case	Implemented?	Feature that is implemented

Verify that the **NameTypeConfigurationElement** class converts the **type** configuration element to the **Type** class.	Yes	The **NameTypeConfigurationElement** class exposes the **type** configuration element as a property. The **AssemblyQualifiedTypeNameConverter** class converts it to the **Type** class. This is shown in the following code example.
		`[ConfigurationProperty(typeProperty, IsRequired=true)]`
		`[TypeConverter(typeof(AssemblyQualifiedTypeNameConverter))]`
		
		`public Type Type`
		`{}`
		The **AssemblyQualifiedTypeNameConverter** class derives from the **ConfigurationConverterBase** class and overrides the **ConvertTo** and **ConvertFrom** methods to convert the **type** element to the **Type** class.

Verify that the core includes a generic helper class that manages custom provider configuration objects.	Yes	The core implements the **CustomProviderDataHelper** class to support custom properties such as **NameValueCollection** objects.
Verify that the configuration sources and configuration watchers are extensible	Yes	The configuration sources **SystemConfigurationSource**, **FileConfigurationSource**, and **DictionaryConfigurationSource** all implement the **IConfigurationSource** interface. The following code shows how the **SystemConfigurationSource** implements the **IConfigurationSource** interface. `public class SystemConfigurationSource : IConfigurationSource {}`

Verify that the core's system configuration source and its file configuration source have configuration watchers that monitor changes to their configuration sources. If there are changes, these watchers should notify the appropriate class, such as the **FileConfigurationSourceImplentation** class, to update the configuration data.	Yes	A **SystemConfigurationSource** object holds a static reference to the SystemConfigurationSourceImplementation class. This reference adds a watcher for a given section when the **SystemConfigurationSource.GetSection** method is called for the first time. This is shown in the following code example.
		`public override ConfigurationSection GetSection(string sectionName)`
		`{`
		`ConfigurationSection configurationSection = ConfigurationManager.GetSection(sectionName) as ConfigurationSection;`
		
		`SetConfigurationWatchers(sectionName, configurationSection);`
		
		`return configurationSection;`
		`}`

continued

Code test case	Implemented?	Feature that is implemented

Verify that the Enterprise Library Core has configuration watchers that monitor specific sections of the external configuration source. If there are changes, these watchers should notify the appropriate class to update the configuration data.	Yes	The **BaseFileConfigurationSourceImplementation** class implements the **CreateWatcherForConfigSource** method that adds a watcher. This watcher monitors specific sections of the external configuration source and notifies the **OnExternalConfigurationChanged** method when the configuration data changes. The **OnExternalConfigurationChanged** method then calls the **RefreshExternalSections** method of a class that derives from the **BaseFileConfigurationSourceImplementation** class to refresh the configuration data with changes. This is shown in the following code.
		`private ConfigurationSourceWatcher CreateWatcherFo rConfigSource(string configSource)`
		`{`
		`…`
		`watcher = new ConfigurationFileSourceWatcher(… ,configSource,…,new ConfigurationChangedEventHandle r(OnExternalConfigurationChanged));`
		`…`
		`}`
		In the following code example, the **FileConfigurationSourceImplementation** object, which derives from the **BaseFileConfigurationSourceImplementation** class, refreshes the configuration data by calling the **UpdateCache** method.
		`protected override void RefreshExternalSections(s tring[] sectionsToRefresh)`
		`{`
		`UpdateCache();`
		`}`

Verify that there are methods to properly dispose of the configuration watchers when the watchers are unregistered from the configuration source.	Yes	The **SystemConfigurationSourceImplementation** class derives from the **BaseFileConfigurationSourceImplementation** class and implements a method named **RemoveConfigSourceWatcher**. This method disposes of the watchers when they are no longer needed. The same holds true for the **FileConfigurationSourceImpementation** class, which also derives from the **BaseFileConfigurationSourceImplementation** class.
		The following code example shows how the **RemoveConfigSourceWatcher** method disposes of the configuration watchers.
		`private void RemoveConfigSourceWatcher(Configuratio nSourceWatcher watcher)`
		`{`
		`…`
		`(watcher as IDisposable).Dispose();`
		`}`

Code test case	Implemented?	Feature that is implemented

Verify that the instrumentation is only bound to an object when the instrumentation is enabled.	Yes	The **InstrumentationStrategy** class binds the instrumentation to an object. The **InstrumentationStrategy** class calls the **AttachInstrumentation** method on the **InstrumentationAttachtmentStrategy** class. This method validates that at least one of the instrumentation attributes is enabled before it binds the instrumentation to the object. This is shown in the following code example.
		`private void AttachInstrumentation(…)`
		`{`
		`…`
		`if (section.InstrumentationIsEntirelyDisabled) return;`
		`….`
		`BindInstrumentationTo(createdObject, constructorArgs, reflectionCache);`
		`}`
		
		`internal bool InstrumentationIsEntirelyDisabled`
		`{`
		`get { return (PerformanceCountersEnabled || EventLoggingEnabled || WmiEnabled) == false; }`
		`}`

To learn how the test teams tested the application blocks to see if they conformed to security best practices, see *Testing for Security Best Practices*. To learn how the test teams tested the application blocks to see if they conformed to globalization best practices, see *Testing for Globalization Best Practices*. To learn how the test teams tested the application blocks to see if they met the performance and scalability requirements, see *Testing for Performance and Scalability*.

Using Automated Tests

Automated tests ensure that the application block functions in accordance with its requirements. Automated tests make regression testing easier and certain tests, such as simulating a large number of users to test a multithreading scenario, require automation.

The automated tests for the Enterprise Library Core use a sample custom application block that tests the core's functionality. This application block supports two types of configuration source sections. The <itemsConfiguration> section contains a collection of polymorphic items that are used by the **CommonItem** class. The **CommonItem** class uses the **price** and the **quantity** elements to calculate a total price. The <Car> section contains configuration data for a single element named **Car**.

The **Car** class uses this data. The automated tests verify that the core creates a **Car** object as a singleton object and a **CommonItem** object as a named instance. The **Car** class provides methods to increase and decrease the car's speed and to change the wheel. The following code shows the two configuration file sections.

```
<itemConfiguration defaultItem="MicrosoftXP">
  <items>
    <add
    name="MicrosoftXP"
    type="….CommonItem,…"
    price="100" quantity="0" />
    …
  </items>
</itemConfiguration>
<carConfiguration>
  <Car name ="MyCar" model="2005" brand="Porsche">
  <CarParts wheeltype="Alloy">
    </CarParts>
  </Car>
</carConfiguration>
```

Table 5 lists the Visual Studio Team System tests that were used with the Enterprise Library Core.

Table 5: Visual Studio Team System Tests for the Core

Test case	Result	Automated test

Verify that the core can use data from the configuration source to create both a named instance and a default instance of a **CommonItem** object.	Passed	The following test creates a named instance of a **CommonItem** object.
		`//This is the named instance test`
		`[TestMethod]`
		`public void CreateNamedTypeObject()`
		`{`
		`ItemProviderFactory factory = new ItemProviderFac-tory();`
		`IItem item = factory.Create("MicrosoftXP");`
		`Assert.IsTrue(item.CalculateTotal() == 0);`
		`}`
		The following test creates a default instance of a **CommonItem** object.
		`// This is the default instance test`
		`[TestMethod]`
		`public void CreateDefaultObject()`
		`{`
		`ItemProviderFactory factory = new ItemProviderFac-tory();`
		`IItem item = factory.CreateDefault();`
		`Assert.IsTrue(item.CalculateTotal() == 0);`
		`}`

Test case	Result	Automated test

Verify that the core can use data from the configuration source to create a **SoftwareItem** object. This object derives from the **CommonItem** class.	Passed	The following is the data from the configuration source.
		`<items>`
		`<itemConfiguration>`
		`<add`
		`name="IE7"`
		`type="….SoftwareItem, …"`
		`price="100" quantity="10" version="1.0.0.0"/>`
		`</items>`
		`</itemConfiguration>`
		The following is the test.
		`[TestMethod]`
		`public void CreateNamedDerivedTypeObject()`
		`{`
		`ItemProviderFactory factory = new ItemProviderFac-`
		`tory();`
		`IItem item = factory.Create("IE7");`
		`Assert.IsTrue(item.CalculateTotal() == 1000);`
		`}`

Verify that the core can use data from a dictionary configuration source to create a class that implements the **IItem** interface.	Passed	The following is the data from the dictionary configuration source.
		`ItemSettings setting = new ItemSettings();`
		`setting.Items.Add(new SoftwareItemData("IE7", Type.`
		`GetType("ConfigurationCoreTests.SoftwareItem,Configurat`
		`ionCoreTests"), 100, 10,"1.0.0.0"));`
		`source.Add("itemConfiguration", setting);`
		The following is the test.
		`[TestMethod]`
		`public void CreateNamedDerivedTypeObject()`
		`{`
		`ItemProviderFactory factory = new ItemProviderFactor`
		`y(source);`
		`IItem item = factory.Create("IE7");`
		`Assert.IsTrue(item.CalculateTotal() == 1000);`
		`}`

continued

Test case	Result	Automated test

Verify that the class that implements the **IItem** interface notifies the instrumentation when it calculates a total price.	Passed	The following test case verifies that the **Item Categories Calcuated/sec** performance counter is incremented when the **CalculateTotal** method calculates the total price for the **SoftwareItem** object. The **GetCounterValue** method returns the current counter value of the countername parameter.
		`[TestMethod]`
		`public void VerifyItemCalculatedPerfCounter()`
		`{`
		`string counterName = "Item Categories Calculated/sec";`
		`int initialCount = GetCounterValue(counterName, "Total");`
		`ItemProviderFactory factory = new ItemProviderFactory();`
		`SoftwareItem item = (SoftwareItem)factory.Create("IE7");`
		`item.CalculateTotal();`
		`int loggedCount = GetCounterValue(counterName, "Total");`
		`Assert.IsTrue((loggedCount - initialCount) == 1);`
		`}`
		In the following example, the quantity is set to zero for the **CommonItem** named MicrosoftXP. The **CommonItem** class throws an exception if the quantity is less than or equal to zero. This test case verifies that when an exception is thrown, an error is logged to the event log through the instrumentation provider.
		`[TestMethod]`
		`public void VerifyErrorLoggedToEventLog()`
		`{`
		`using (EventLog log = new EventLog("Application"))`
		`{`
		`int initialCount = log.Entries.Count;`
		`ItemProviderFactory factory = new ItemProviderFactory();`
		`IItem item = factory.Create("MicrosoftXP");`
		`item.CalculateTotal();`
		`int finalCount = log.Entries.Count;`
		`Assert.IsTrue(finalCount - initialCount == 1);`
		`Assert.IsTrue(log.Entries[finalCount - 1].Message.Contains("The quantity value should be greater than zero"));`
		`Assert.IsTrue(log.Entries[finalCount - 1].Source.Equals("Enterprise Library Test"));`
		`}`
		`}`

Test case	Result	Automated test

Verify that the core can use data from a configuration source to create an instance of the **Car** class.	Passed	The following is the test.
		`[TestMethod]`
		`public void CreateObjectTest()`
		`{`
		`CarFactory factory = new CarFactory();`
		`Car obj1 = factory.Create("MyCar");`
		`Assert.IsNotNull(obj1);`
		`}`

Verify that the core can create a **Car** singleton object.	Passed	The following is the test.
		`[TestMethod]`
		`public void VerifyObjectCreatedOnlyOnce()`
		`{`
		`CarFactory factory = new CarFactory();`
		`Car obj1 = factory.Create("MyCar");`
		`Car obj2 = factory.Create("MyCar");`
		`Assert.IsNotNull(obj1);`
		`Assert.IsNotNull(obj2);`
		`Assert.IsTrue(object.ReferenceEquals(obj1,obj2));`
		`}`

Testing the Caching Application Block

This chapter explains how functional testing techniques were used to test the Caching Application Block. If you have modified or extended the Caching Application Block, you can use the same techniques and adapt the chapter's templates and checklists to test your own work.

Requirements for the Caching Application Block

The Caching Application Block has the following requirements:

- The application block should support common caching operations such us adding items to the cache, removing items from the cache, retrieving items from the cache, and flushing the cache.
- The application block should provide the ability to configure expiration policies.
- The application block should provide the ability to cache data in persistent stores, such as a database and isolated storage.
- The application block should be extensible.
- The application block should be able to read configuration information from any configuration source, such as an XML file or a database.
- The application block should support configurable instrumentation, including WMI (Windows Management Instrumentation), performance counters, and event logs.

These requirements must be incorporated into the design and implemented by the code.

Selecting the Test Cases

The first step in a functional review is to verify that the design and the code support the requirements. You do this by deciding the test cases that the design and code must satisfy. Table 1 lists the test cases that the Caching Application Block's design must satisfy.

Table 1: Caching Application Block Design Test Cases

Priority	Design test case
High	Verify that the caching stores, expiration policies, and encryption providers are extensible.
High	Verify that there is a consistent approach to create caching stores, encryption providers, and **CacheManager** instances.
High	Verify that the **CacheManager** class supports simple methods for adding items to the cache, removing items from the cache, retrieving items from the cache, and flushing the cache.
High	Verify that application block can encrypt cached items in the backing store.
High	Verify that the ability to create the application block's domain objects from configuration data follows the Dependency Injection pattern.
High	Verify that the application block can retrieve configuration data from different sources, such as an application configuration file, a database, or from memory.
High	Verify that the instrumentation is implemented with loosely coupled events.
High	Verify that the design addresses situations that can cause exceptions and that the application block logs the exceptions through the instrumentation.
High	Verify that the application block supports custom property collections for the custom caching stores.

After you identify the design issues, you should do the same for the code. Table 2 lists the test cases that the Caching Application Block's code must satisfy.

Table 2: Caching Application Block Code Test Cases

Priority	Code test case
High	Verify that the application block creates only one **CacheManager** instance for a particular instance name.
High	Verify that the assembler classes that implement the **IAssembler** interface create the backing store providers and the encryption providers, and verify that the assembler classes inject the configuration object values into those domain objects.
High	Verify that the application block uses performance counters to monitor caching operations when the performance counters are enabled.
High	Verify that the application block uses WMI and the event log to monitor errors during caching operations when WMI and the event log are enabled.
High	Verify that the application block loads the cached data from the correct partition in the backing store.
High	Verify that the application block performs scavenging when the cached items exceed the configured limit and that the number of items to be scavenged is configurable.
High	Verify that the application block uses a cached item's priority setting and the time it was last accessed to scavenge items from the cache.
High	Verify that the application block operations, such as the **Add** and **Remove** methods, enforce a strong exception guarantee. This means that if an operation fails, the state of the cache rolls back to what it was before the attempted operation.
High	Verify that methods calls on the **CacheManager** object are thread safe.

Priority	Code test case
High	Verify that the **CacheFactory** class uses the **CacheManagerFactory** class to create the **CacheManager** instance.
High	Verify that the database backing store uses a named instance of the store and uses the Data Access Application Block to create that instance.
High	Verify that the application block can use an instance name to create a **CacheManager** object.
High	Verify that the application block can use a default instance name to create a **CacheManager** object.
High	Verify that the performance counters and the event log that are required by the application block are installed during installation.
Medium	Verify that the application block requests or demands the appropriate code access security permissions to access protected system resources and operations.
Medium	Verify that the application block follows exception management best practices.
High	Verify that the application block follows security best practices.
Medium	Verify that the application block follows globalization best practices.
High	Verify that the application block follows performance best practices.

Verifying the Test Cases

After you identify all the design test cases, you can verify that the design satisfies them. Table 3 lists how each of the design test cases was verified for the Caching Application Block.

Table 3: Caching Application Block Design Verification

Design test case	Implemented?	Feature that implements design
Verify that the caching stores, expiration policies, and encryption providers are extensible.	Yes	New backing stores must implement either the **BaseBackingStore** abstract class or the **IBackingStore interface**. New expiration policies must implement the **ICacheItemExpiration** interface and the **ICacheItemRefreshAction** interface. New encryption providers must implement the **IStorageEncryptionProvider** interface.
Verify that there is a consistent approach to creating caching stores, encryption providers, and **CacheManager** instances.	Yes	The **BackingStoreCustomFactory** class creates the backing store providers. The **StorageEncryptionProviderCustomFactory** creates the encryption providers. The **CacheFactory** class creates the **CacheManager** instances.

continued

Design test case	Implemented?	Feature that implements design
Verify that the **CacheManager** class supports simple methods for adding items to the cache, removing items from the cache, retrieving items from the cache, and flushing the cache.	Yes	The **CacheManager** class includes the **Add** method to add items to the cache, the **Remove** method to remove items from the cache, the **GetData** method to retrieve items from the cache and the **Flush** method to flush the cache.
Verify that the application block can encrypt cached items in the backing store.	Yes	The application block uses a class that implements the **ISymmetricCryptoProvider** interface to encrypt cached items.
Verify that the ability to create the application block's domain objects from configuration data follows the Dependency Injection pattern.	Yes	The **CacheManagerFactory** class derives from the **LocatorNameTypeFactoryBase** class, which takes the configuration source as input, creates the domain object, and injects the dependent configuration data into the domain object.
Verify that the application block can retrieve configuration data from different sources, such as an application configuration file, a database, or from memory.	Yes	The **CacheManagerFactory** class has a constructor that accepts a configuration source as an input parameter.
Verify that the instrumentation is implemented with loosely coupled events.	Yes	The methods in the **CachingInstrumentationProvider** class that raise the events bind to the methods in the **CachingInstrumentationListener** class at run time.
Verify that the design addresses situations that can cause exceptions and that the application block logs the exceptions through the instrumentation.	Yes	For example, the **Cache** class includes the **InstrumentationProvider** property. This property retrieves the instrumentation provider that defines the events for the caching provider. The provider notifies WMI and logs the exceptions to the event log.
Verify that the application block supports custom property collections for the custom caching stores.	Yes	The **CustomCacheStorageData** class provides the ability to configure custom property collections for custom caching stores.

After the code is implemented, you can review it to see if it satisfies its test cases. Table 4 lists the results of a code review for the Caching Application Block.

Table 4: Caching Application Block Code Verification

Code test case	Implemented?	Feature that is implemented
Verify that the application block creates only one **CacheManager** instance for the given instance name.	Yes	The **CacheManagerFactory** class creates a **CacheManager** instance. The **CacheManagerFactory** class derives from the **LocatorNameTypeFactoryBase** class. This class creates only one instance for a given instance name.

Verify that assembler classes that implement the **IAssembler** interface create the backing store providers and the encryption providers, and inject the configuration object values into those domain objects.	Yes	The following code demonstrates how the **DataBackingStoreAssembler** class, which implements the **IAssembler** interface, creates a backing store.
		`public class DataBackingStoreAssembler : IAssembler<IBackingStore, CacheStorageData>`
		`{`
		`public IBackingStore Assemble(…)`
		`{`
		`IBackingStore createdObjet = new DataBackingStore(…);`
		`return createdObjet;`
		`}`
		`}`
		The following code demonstrates how the **SymmetricStorageEncryptionProviderAssembler**, which implements the **IAssembler** interface, creates a symmetric encryption provider.
		`public class SymmetricStorageEncryptionProviderAssembler : IAssembler<IStorageEncryptionProvider, StorageEncryptionProviderData>`
		`{`
		`public IStorageEncryptionProvider Assemble(…)`
		`{`
		`IStorageEncryptionProvider createdObject = new SymmetricStorageEncryptionProvider(symmetricCrytoProvider);`
		`return createdObject;`
		`}`
		`}`

continued

Code test case	Implemented?	Feature that is implemented

Verify that the application block uses performance counters to monitor caching operations when the performance counters are enabled.	Yes	For example, the **CacheManager.GetData** method calls the **Cache.GetData** method. When this method retrieves an item from the cache, it uses an instance of the **CachingInstrumentationProvider** class to increment the **Cache Hits/sec** performance counter. If the item is not in the cache, the method increments the **Cache Misses/sec** performance counter. This is shown in the following code example.
		`public object GetData(string key)`
		`{`
		`…`
		`instrumentationProvider.`
		`FireCacheAccessed(key, false);`
		`…`
		`}`
		The **CachingInstrumentationProvider** class raises the **cacheAccessed** event. The **CacheAccessed** method consumes the event and increments the performance counters, This is shown in the following code example.
		`[InstrumentationConsumer("CacheAccesse`
		`d")]`
		`public void CacheAccessed(…`
		`,CacheAccessedEventArgs e)`
		`{`
		`if (PerformanceCountersEnabled)`
		`{`
		`cacheAccessAttemptsCounter.Increment();`
		`if (e.Hit)`
		`{`
		`cacheHitRatioCounter.Increment();`
		`cacheHitsCounter.Increment();`
		`}`
		`else`
		`{`
		`cacheMissesCounter.Increment();`
		`}`
		`}`
		`}`

Code test case	Implemented?	Feature that is implemented

Verify that the application block uses WMI and the event log to monitor errors during caching operations when WMI and the event log are enabled.	Yes	For example, if an exception occurs when the application block refreshes the cache, the **RefreshActionInvoker.InvokeRefreshAction** method handles the exception. This method uses an instance of the **CachingInstrumentationProvider** to notify WMI and the event log. This is shown in the following code example.
		`public static void InvokeRefreshAction(…)`
		`{`
		`try`
		`{`
		`…`
		`refreshActionData.InvokeOnThreadPoolThread();`
		`}`
		`catch (Exception e)`
		`{`
		`instrumentationProvider.FireCacheFailed(…);`
		`}`
		`}`
		The **CachingInstrumentationProvider** class raises the **cacheFailed** event. The **CacheFailed** method consumes the event and notifies WMI and logs the exception to the event log. This is shown in the following code example.
		`[InstrumentationConsumer("CacheFailed")]`
		`public void CacheFailed(…)`
		`{`
		`if (WmiEnabled)`
		`{`
		`System.Management.Instrumentation.Instrumentation.Fire(new CacheFailureEvent(instanceName, e.ErrorMessage, e.Exception.ToString()));`
		`}`
		`if (EventLoggingEnabled)`
		`{`
		`…`
		`EventLog.WriteEntry(GetEventSourceName(), entryText, EventLogEntryType.Error);`
		`}`
		`}`

continued

Code test case	Implemented?	Feature that is implemented

Verify that the application block loads the cached data from the correct partition in the backing store.	Yes	Instances of the **DataBackingStore** class load the data from the partition that is specified in the configuration source. The **AddInParameter** method accepts the partition name as a parameter. This is shown in the following code example.
		`protected override Hashtable LoadDataFromStore()`
		`{`
		`DbCommand loadDataCommand = database.GetStoredProcCommand("LoadItems");`
		
		`database.AddInParameter(loadDataCommand, "@partitionName", DbType.String, partitionName);`
		
		`DataSet dataToLoad = database.ExecuteDataSet(loadDataCommand);`
		`}`
		The process is similar for instances of the **IsolatedStorageBackingStore** class.

Code test case	Implemented?	Feature that is implemented

Verify that the application block performs scavenging when the cached items exceed the configured limit. In addition, verify that the number of items to be scavenged is configurable.	Yes	When an item is added to an instance of the **Cache** class, the application block checks to see if the limit set in the configuration source has been reached. If it has, the scavenging process begins. This is shown in the following code example.
		`public void Add(…)`
		`{`
		`if (scavengingPolicy.IsScavengingNeeded (inMemoryCache.Count))`
		`{`
		`cacheScavenger.StartScavenging();`
		`}`
		`}`
		The number of items that are scavenged depends on the value set in the configuration source. This value is injected into the **ScavengerTask** class. The **RemoveScavengableItems** method checks for this value before it removes the specified number of items from the cache. This is shown in the following code example.
		`private void RemoveScavengableItems(SortedList scavengableItems)`
		`{`
		`int scavengedItemCount = 0;`
		`foreach (CacheItem scavengableItem in scavengableItems.Values)`
		`{`
		`…`
		`if (scavengedItemCount == NumberOfItemsToBeScavenged)`
		`{`
		`break;`
		`}`
		`}`
		`}`

continued

Code test case	Implemented?	Feature that is implemented

Verify that the application block uses a cached item's priority setting and the time it was last accessed to scavenge items from the cache.	Yes	When scavenging is required, the items in the cache are sorted based on their priority settings and the times that they were last accessed. The items are then removed from the cache. This is shown in the following code example.
		`public void DoScavenging()`
		`{`
		`…`
		`if (scavengingPolicy.IsScavengingNeeded (currentNumberItemsInCache))`
		`{`
		`SortedList scavengableItems = SortItems ForScavenging(liveCacheRepresentation);`
		`RemoveScavengableItems(scavengableItem s);`
		`}`
		`}`
		The **SortedList** class uses the **PriorityDate-Comparer** class to sort items in the cache. This class implements the **IComparer** interface. The following code demonstrates how items in the cache are sorted.
		`public int Compare(…)`
		`{`
		`…`
		`return leftCacheItem.ScavengingPrior- ity == rightCacheItem.ScavengingPrior- ity? leftCacheItem.LastAccessedTime. CompareTo(rightCacheItem.LastAccessed- Time): leftCacheItem.ScavengingPriority - rightCacheItem.ScavengingPriority;`
		`}`

Code test case	Implemented?	Feature that is implemented

Verify that the application block operations, such as the **Add** and **Remove** methods enforce a strong exception guarantee. This means that if an operation fails, the state of the cache rolls back to what it was before the attempted operation.	Yes	The **CacheManager** class has an internal reference to a **Cache** object. This object guarantees exception-safe caching operations. For example, if an exception occurs during an add operation, the state of the cache rolls back to what it was before the attempted operation. This is shown in the following code example.
		`public void Add(…)`
		`{`
		`try`
		`{`
		`…  backingStore.`
		`Add(newCacheItem);`
		`inMemoryCache[key] = cacheItemBefore-`
		`Lock;`
		`}`
		`catch`
		`{`
		`backingStore.Remove(key);`
		`inMemoryCache.Remove(key);`
		`throw;`
		`}`
		`}`

continued

Code test case	Implemented?	Feature that is implemented

Verify that methods calls on the **CacheManager** object are thread safe.	Yes	The **CacheManager** class has an internal reference to a **Cache** object. This object implements thread-safe caching operations. For example, when the application block adds an item to the in-memory cache, it locks and synchronizes the cache, adds the item to the backing store, and then releases the lock. This is shown in the following code example.
		`public void Add(…)`
		`{`
		`…`
		`lock (inMemoryCache.SyncRoot)`
		`{`
		`…`
		`lockWasSuccessful = Monitor.TryEnter(cacheItemBeforeLock);`
		`}`
		`try`
		`{`
		`…`
		`backingStore.Add(newCacheItem);`
		`inMemoryCache[key] = cacheItemBeforeLock;`
		`…`
		`}`
		`finally`
		`{`
		`Monitor.Exit(cacheItemBeforeLock);`
		`}`
		`}`

Verify that the **CacheFactory** class uses the **CacheManagerFactory** class to create a **CacheManager** instance.	Yes	The **CacheFactory** class uses the **CacheManagerFactory** class to create the **CacheManager** instance. This is shown in the following code example.
		`public static class CacheFactory`
		`{`
		`private static CacheManagerFactory factory = new CacheManagerFactory(ConfigurationSourceFactory.Create());`
		
		`public static CacheManager GetCacheManager()`
		`{`
		`…`
		`return factory.CreateDefault();`
		`}`
		`…`
		`}`

Code test case	Implemented?	Feature that is implemented

Verify that the database backing store uses a named instance of the store and uses the Data Access Application Block to create that instance.	Yes	The **ObjectBuilder** subsystem uses the Data Access Application Block to create the **Database** instance that is the database backing store. This is shown in the following code example.
		`public class DataBackingStoreAssembler : IAssembler<IBackingStore, CacheStorageData>`
		`{`
		`public IBackingStore Assemble(…)`
		`{`
		`…`
		`Data.Database database = (Data.Database)context.HeadOfChain.BuildUp(context,typeof(Data.Database), null,castedObjectConfiguration.DatabaseInstanceName);`
		
		`IBackingStore createdObjet = new DataBackingStore(database,castedObjectConfiguration.PartitionName,encryptionProvider);`
		`}`
		`}`
		The following configuration example shows how the database instance name is configured in the configuration source.
		`<backingStores>`
		`<add name="Data Cache Storage" type="Microsoft.Practices.EnterpriseLibrary.Caching.Database.DataBackingStore, Microsoft.Practices.EnterpriseLibrary.Caching.Database" databaseInstanceName="CachingDatabase" partitionName="Partition1" />`

Verify that the application block can use an instance name to create a **CacheManager** object.	Yes	The **CacheManager.GetCacheManager** method has two overloads. One of them includes an instance name as a parameter to create a specific **CacheManager** object. This is shown in the following code example. `public static CacheManager GetCacheManager(string cacheManagerName){}`
Verify that the application block can use a default instance name to create a **CacheManager** object.	Yes	The **CacheManager.GetCacheManager** method has two overloads. One of them creates a default **CacheManager** object. This is shown in the following code example. `public static CacheManager GetCacheManager(){}`

continued

Code test case	Implemented?	Feature that is implemented
Verify that the performance counters and the event log that are required by the application block are installed during installation.	Yes	For example, the **CachingInstrumentationListener** class includes the installer attribute type **[HasInstallableResourcesAttribute]**. The installer classes **EventLogInstallerBuilder** and **PerformanceCounterInstallerBuilder** recognize this attribute and install the performance counters and event logs.

To learn how the test teams tested the application blocks to see if they conformed to security best practices, see *Testing for Security Best Practices*. To learn how the test teams tested the application blocks to see if they conformed to globalization best practices, see *Testing for Globalization Best Practices*. To learn how the test teams tested the application blocks to see if they met the performance and scalability requirements, see *Testing for Performance and Scalability*.

Using Automated Tests

Automated tests ensure that the application block functions in accordance with its requirements. Automated tests make regression testing easier and certain tests, such as simulating a large number of uses to test a multithreading scenario, require automation.

Table 5 lists the Visual Studio Team System tests that were used with the Caching Application Block.

Table 5: Visual Studio Team System Tests for the Caching Application Block

Test case	Result	Automated test

Verify that the application block throws the appropriate exception when it receives invalid data.	Passed	The following test uses the name of a **CacheManager** object that is not defined in the configuration source.
		`[TestMethod]`
		`[ExpectedException(typeof(System.Configuration.ConfigurationErrorsException))]`
		`public void IsolatedStoreEncConfigErrorTest()`
		`{`
		`itemCache = CacheFactory.GetCacheManager("IsolatedConfigErrorEncryption");`
		
		`ItemDetails itemDetails = new ItemDetails(1, "Toy1", 25);`
		`itemCache.Add(itemDetails.ItemId.ToString(), itemDetails);`
		`Assert.AreEqual(1,itemCache.Count);`
		`}`

Test case	Result	Automated test

Verify that the application block adds valid items to the backing store.	Passed	The following test adds an item to the database backing store and then checks to see that the item is available both from the backing store and from memory. The following is the configuration file. It sets the backing store.
		`<connectionStrings>`
		`<add name="CachingDatabase"`
		`providerName="System.Data.SqlClient"`
		`connectionString="..." />`
		`</connectionStrings>`
		`<backingStores>`
		`<add name="Data Cache Storage"`
		`type="Microsoft.Practices.EnterpriseLibrary.Caching.Database.DataBackingStore, Microsoft.Practices.EnterpriseLibrary.Caching.Database"`
		`databaseInstanceName="CachingDatabase" partitionName="Partition1" />`
		`</backingStores>`
		`<cacheManagers>`
		`<add name="DBCacheManager"`
		…
		`backingStoreName="Data Cache Storage" />`
		`</cacheManagers>`
		The following is the test.
		`TestMethod]`
		`public void DBStoreAddDefaultTest()`
		`{`
		`CacheManager itemCache = CacheFactory.GetCacheManager("DBCacheManager");`
		`ItemDetails itemDetails = new ItemDetails(1, "Toy1", 25);`
		`itemCache.Add(itemDetails.ItemId.ToString(), itemDetails);`
		
		`SqlConnection conn = new SqlConnection();`
		`conn.ConnectionString = connStr;`
		`conn.Open();`
		`SqlCommand cmd = new SqlCommand("select * from CacheData where [key] in ('1') and partitionname ='Partition1'",conn);`
		`SqlDataReader dr = cmd.ExecuteReader();`
		`while (dr.Read())`
		`{`
		`count = count + 1;`
		`}`
		`dr.Close();`
		`conn.Close();`
		`Assert.AreEqual(1, count);`
		`ItemDetails writtenItem = new ItemDetails();`
		`writtenItem = (ItemDetails)itemCache.GetData("1");`
		`Assert.AreEqual("Toy1", writtenItem.Name);`
		`Assert.AreEqual(Convert.ToDecimal(25), writtenItem.Price);`
		`}`

continued

Test case	Result	Automated test

Verify that a **CacheManager** object can be created from an in-memory dictionary configuration source.	Passed	The following test creates a **CacheManager** object from a dictionary configuration source and verifies that the object was created. The following is the dictionary configuration source.
		`public class CachingDictionarySource`
		`{`
		`public DictionaryConfigurationSource BuildDictionarySourceSection()`
		`{`
		`DictionaryConfigurationSource sections = new DictionaryConfigurationSource();`
		
		`CacheManagerSettings settings = new CacheManagerSettings();`
		
		`settings.DefaultCacheManager = "InMemoryCacheManager";`
		
		`CacheStorageData csd = new CacheStorageData();`
		`csd.Name = "inMemoryDic";`
		`csd.Type = Type.GetType("Microsoft.Practices.EnterpriseLibrary.Caching.BackingStoreImplementations.NullBackingStore, Microsoft.Practices.EnterpriseLibrary.Caching");`
		`settings.BackingStores.Add(csd);`
		
		`CacheManagerData cmd = new CacheManagerData();`
		`cmd.Name = "InMemoryCacheManager";`
		`//configure the expiration and scavenging policies`
		`…`
		`cmd.CacheStorage = "inMemoryDic";`
		`settings.CacheManagers.Add(cmd);`
		`sections.Add("cachingConfiguration", settings);`
		`}`
		`return sections;`
		`}`
		The following is the test.
		`[TestMethod]`
		`public void CacheManagerInstanceTest()`
		`{`
		`CachingDictionarySource sourceSection = new CachingDictionarySource();`
		`DictionaryConfigurationSource configSource = sourceSection.BuildDictionarySourceSection();`
		
		`CacheManagerFactory factory = new CacheManagerFactory(configSource);`
		`CacheManager itemCache = factory.Create("InMemoryCacheManager");`
		`Assert.IsNotNull(itemCache);`
		`}`

Test case	Result	Automated test

Verify that custom backing stores can be added to the application block and configured from information in the configuration source.	Passed	In the following test case, the **MockBackingStore** class is a custom backing store that implements the **IBackingStore** interface. The following is the configuration information for the custom backing store.
		`<cacheManagers>`
		`<add name="CustomCacheManager"`
		`expirationPollFrequencyInSeconds="1"`
		`maximumElementsInCacheBeforeScavenging="10"`
		`numberToRemoveWhenScavenging="3"`
		`backingStoreName="CustomStore" />`
		`</cacheManagers>`
		`<backingStores>`
		`<add name="CustomStore"`
		`type="CachingCoreTests.MockBackingStore, Caching-CoreTests" itemid ="1" itemdescription="Item1"`
		`itemprice="28.75"/>`
		`<backingStores>`
		The following is the **MockBackingStore** class.
		`[ConfigurationElementType(typeof(CustomCacheStorage Data))]`
		`public class MockBackingStore : IBackingStore`
		`{`
		`public static NameValueCollection attributes;`
		`public static int count;`
		`public MockBackingStore(NameValueCollection attri-butes)`
		`{`
		`MockBackingStore.attributes = attributes;`
		`}`
		`public int Count`
		`{`
		`get { return count; }`
		`}`
		
		`public void Add(CacheItem newCacheItem)`
		`{`
		`count++;`
		`}`
		
		`public void Remove(string key)`
		`{`
		`count--;`
		`}`
		`public void UpdateLastAccessedTime(string key, Da-teTime timestamp){}`
		
		`public void Flush()`
		`{`
		`count = 0;`
		`}`
		
		`public Hashtable Load()`

continued

Test case	Result	Automated test

		{
		Hashtable items = new Hashtable();
		return items;
		}
		
		public void Dispose()
		{
		}
		}
		The following is the test.
		[TestMethod]
		public void MockStoreAddTest()
		{
		CacheManager itemCache = CacheFactory.GetCacheMana
		ger("CustomCacheManager");
		itemCache.Add("1", "test");
		string cacheData = (string) itemCache.GetData("1");
		Assert.AreEqual("test", cacheData);
		Assert.AreEqual(3, MockBackingStore.attributes.
		Count);
		Assert.AreEqual("1", MockBackingStore.
		attributes["itemid"]);
		Assert.AreEqual("Item1", MockBackingStore.attribut
		es["itemdescription"]);
		Assert.AreEqual("28.75", MockBackingStore.attribut
		es["itemprice"]);
		}

Testing the Cryptography Application Block

This chapter explains how functional testing techniques were used to test the Cryptography Application Block. If you have modified or extended the Cryptography Application Block, you can use the same techniques and adapt the chapter's templates and checklists to test your own work.

Requirements for the Cryptography Application Block

The Cryptography Application Block has the following requirements:

- The application block should support common cryptography operations.
- The application block should be extensible.
- The symmetric encryption providers and the hash providers should be configurable.
- The application block should support configurable instrumentation, including WMI (Windows Management Instrumentation), performance counters, and event logs.
- The application block should be able to read configuration information from any configuration source, such as an XML file or a database.
- The symmetric key that encrypts the data should be cached in memory in an encrypted form.
- The application block should work with desktop applications and with Web applications.

These requirements must be incorporated into the design and implemented by the code.

Selecting the Test Cases

The first step in a functional review is to make sure that the design and the code support the requirements. You do this by deciding the test cases that the design and code must satisfy. Table 1 lists the test cases that the application block's design must satisfy.

Table 1: Cryptography Application Block Design Test Cases

Priority	Design test case
High	Verify that the symmetric algorithm providers and the hash providers are extensible.
High	Verify that there is a consistent approach to creating symmetric algorithm providers and hash providers.
High	Verify that there is a façade that mediates between the client code and the application block's cryptographic functions, such as encryption, decryption, and hashing.
High	Verify that the application block uses a façade to generate the symmetric keys.
High	Verify that the application block addresses key management issues, such as the ability to read a key from an output stream, the ability to write a key to an output stream, the ability to archive a key, and the ability to transfer keys between computers.
High	Verify that the application block caches the encrypted key in memory.
High	Verify that the application block supports a method that decrypts the cached encrypted key.
High	Verify that the ability to create the application block's domain objects from the configuration data follows the Dependency Injection pattern.
High	Verify that the application block can retrieve configuration data from different sources, such as an application configuration file, a database, or from memory.
High	Verify that the instrumentation is implemented with loosely coupled events.
High	Verify that situations that can cause exceptions are addressed and that the application block logs the exceptions through the instrumentation.
High	Verify that the application block supports custom property collections for custom symmetric key providers and for custom hash providers.

After you identify the design issues, you should do the same for the code. Table 2 lists the test cases that the Cryptography Application Block code must satisfy.

Table 2: Cryptography Application Block Code Test Cases

Priority	Code test case
High	Verify that the **Cryptographer** façade exposes all public members as static and supports methods for encryption, decryption, and hashing.
High	Verify that the **Cryptographer** façade uses the **SymmetricCryptoProviderFactory** class and the **HashProviderFactory** class to create the cryptography providers.
High	Verify that the assembler classes that implement the **IAssembler** interface create the symmetric and hash providers, and verify that the assembler classes inject the configuration object values into those domain objects.
High	Verify that the application block uses performance counters to monitor hash and symmetric operations when the performance counters are enabled.
High	Verify that the application block uses WMI and the event log to monitor errors during encryption, decryption, and hash operations when WMI and the event log are enabled.

Priority	Code test case
High	Verify that the configuration properties of the providers are exposed as public and are strongly typed.
High	Verify that the configuration properties for the custom providers are exposed as public and that they are implemented as custom property collections.
High	Verify that there is a way to remove decrypted keys from memory after they have been used.
High	Verify that there is a way to properly dispose of the symmetric and hash algorithms after they are used.
High	Verify that the **SymmetricAlgorithmProvider** class and the **KeyedHashAlgorithmProvider** class support both machine mode encryption and user mode encryption.
High	Verify that the application block can either use an absolute path or a relative path to read the key file.
Medium	Verify that the application block validates the input at all the entry points, such as the **Cryptographer** façade.
High	Verify that the application block reads the symmetric key only once from the input stream and then caches it so that it can be used for the cryptography operations.
High	Verify that the symmetric key is cached in memory in a thread safe manner.
High	Verify that the **HashProviderFactory** and the **SymmtericCryptoProviderFactory** classes create new instances of the symmetric and hash providers for each request.
High	Verify that the performance counters and the event log that are required by the application block are installed during installation.
Medium	Verify that the application block requests or demands the appropriate code access security permissions to access protected system resources and operations.
High	Verify that the application block follows exception management best practices.
High	Verify that the application block follows security best practices.
Medium	Verify that the application block follows globalization best practices.
High	Verify that the application block follows performance best practices.

Verifying the Test Cases

After you identify all the design test cases, you can verify that the design does in fact satisfy them. Table 3 lists how each of the design test cases were verified for the Cryptography Application Block.

Table 3: Cryptography Application Block Design Verification

Design test case	Implemented?	Feature that implements design
Verify that the symmetric algorithm providers and the hash providers are extensible.	Yes	The **ISymmetricCryptoProvider** interface allows users to implement or extend a configurable symmetric provider. The **IHashProvider** interface allows users to implement or extend a hash provider.

continued

Design test case	Implemented?	Feature that implements design
Verify that there is a consistent approach to creating symmetric algorithm providers and hash providers.	Yes	The **SymmetricCryptoProviderFactory** class is the factory that creates the **SymmetricProvider** objects. The **HashProviderFactory** class is the factory that creates the **HashProvider** objects.
Verify that there is a façade that mediates between the client code and the application block's cryptographic functions, such as encryption, decryption, and hashing.	Yes	The **Cryptographer** class is a façade that acts as the interface between the client code and the application block.
Verify that the application block uses a façade to generate the symmetric keys.	Yes	The **KeyManager** class is a static façade that exposes methods that generate the keys. It implements the **GenerateSymmetricKey** method and the **GenerateKeyedHashKey** method. These methods have multiple overloads that generate either an encrypted key or an unencrypted key.
Verify that the application block addresses key management issues, such as the ability to read a key from an output stream, the ability to write a key to an output stream, the ability to archive a key, and the ability to transfer keys between computers.	Yes	The **Key Manager** class is a static façade that includes the **Read** method to read keys from an output stream, the **Write** method to write keys to an output stream, the **ArchiveKey** method to archive keys and the **RestoreKey** method to transfer keys between computers.
Verify that the application block caches an encrypted key.	Yes	The **ProtectedKeyCache** class caches an encrypted key inside a collection. Note that this is an internal class.
Verify that the application block supports a method to decrypt the cached encrypted key.	Yes	The **ProtectedKey** class includes the **Unprotect** method that retrieves and decrypts a key.
Verify that the ability to create the application block's domain objects from the configuration data follows the dependency injection pattern.	Yes	The **SymmetricCryptoProviderFactory** class derives from the **NameTypeFactoryBase** generic type, which takes the configuration source as input, creates the domain object, and injects the relevant configuration data into the domain object.
Verify that the application block can retrieve configuration data from different sources, such as an application configuration file, a database, or from memory.	Yes	The **SymmetricCryptoProviderFactory** class and the **HashProviderFactory** class have constructors that accept a configuration source as an input parameter.

Design test case	Implemented?	Feature that implements design
Verify that the instrumentation is implemented with loosely coupled events.	Yes	The methods in the **HashAlgorithmInstrumentationProvider** class that raise the events bind to the methods in the **HashAlgorithmInstrumentationListener** class at run time.
Verify that situations that can cause exceptions are addressed and that the application block logs the exceptions through the instrumentation.	Yes	For example, both the **HashAlgorithmProvider** class and the **SymmetricAlgorithmProvider** class include the **InstrumentationProvider** property. This property retrieves the instrumentation provider that defines the events for the cryptography provider. The provider logs the events to WMI and the event log.
Verify that the application block supports custom property collections for the custom symmetric key providers and for custom hash providers.	Yes	The **CustomHashProviderData** class and the **CustomSymmetricCryptoProviderData** class provides the ability to configure custom property collections for custom providers.

After the code is implemented, you can review it to see if it satisfies its test cases. Table 4 lists the results of a code review for the Cryptography Application Block.

Table 4: Cryptography Application Block Code Verification

Code test case	Implemented?	Feature that is implemented

Verify that the **Cryptographer** façade exposes all public members as static and supports methods for encryption, decryption, and hashing.	Yes	The **Cryptographer** class is a façade that exposes the **CreateHash** method to compute the hash value of plain text, the **CompareHash** method to compare plain text with a hash value, the **EncryptSymmetric** method to encrypt plain text, and the **DecryptSymmetric** method to decrypt a symmetrically encrypted secret. These methods are shown in the following code.
		`public static byte[] CreateHash(string hashInstance, byte[] plaintext) {}`
		
		`public static bool CompareHash(string hashInstance, string plaintext, string hashedText) {}`
		
		`public static string EncryptSymmetric(string symmetricInstance, string plaintext) {}`
		
		`public static string DecryptSymmetric(string symmetricInstance, string ciphertextBase64) {}`

continued

Code test case	Implemented?	Feature that is implemented

Verify that the **Cryptographer** façade uses the **SymmetricCryptoProviderFactory** class and the **HashProviderFactory** class to create the cryptography providers.	Yes	The client code calls static methods on the **Cryptographer** class to create hashes, compare hashes, encrypt data, and decrypt data. Each static method instantiates a factory class and passes the configuration source to the factory class's constructor. The factory uses the configuration data to determine the type of provider to create. The following code demonstrates how the **Cryptographer.EncryptSymmetric** method calls the **SymmetricCryptoProviderFactory** class to create a symmetric provider. The process is similar for hash providers.
		`public static byte[] EncryptSymmetric(string symmetricInstance, byte[] plaintext)`
		`{`
		`…`
		`SymmetricCryptoProviderFactory factory = new SymmetricCryptoProviderFactory(ConfigurationSourceFactory.Create());`
		`…`
		`}`

Verify that assembler classes that implement the **IAssembler** interface create the symmetric and hash providers, and inject the configuration object values into those domain objects.	Yes	The following code demonstrates how the **SymmetricAlgorithmProviderAssembler** class, which implements the **IAssembler** interface, creates a symmetric provider. The process is similar for hash providers.
		`public class SymmetricAlgorithmProviderAssembler : IAssembler<ISymmetricCryptoProvider, SymmetricProviderData>`
		`{`
		`public ISymmetricCryptoProvider Assemble(IBuilderContext context, SymmetricProviderData objectConfiguration, IConfigurationSource configurationSource, ConfigurationReflectionCache reflectionCache)`
		`{`
		
		`SymmetricAlgorithmProviderData castedObjectConfiguration = (SymmetricAlgorithmProviderData)objectConfiguration;`
		
		`ISymmetricCryptoProvider createdObject = new SymmetricAlgorithmProvider(castedObjectConfiguration.AlgorithmType, castedObjectConfiguration.ProtectedKeyFilename,castedObjectConfiguration.ProtectedKeyProtectionScope);`
		
		`return createdObject;`
		`}`
		`}`

Code test case	Implemented?	Feature that is implemented

Verify that the application block uses performance counters to monitor hash and symmetric operations when the performance counters are enabled.	Yes	For example, the **SymmtericAlgorithmProvider.Encrypt** method increments the symmetric encryption performance counter whenever there is a successful encryption operation. This is shown in the following code example. The process is similar for hash operations.
		`public byte[] Encrypt(…)`
		`{`
		`crypto.Encrypt(plaintext);`
		
		`InstrumentationProvider.FireSymmetricEncryptionPerformed();`
		`}`
		`[InstrumentationConsumer("SymmetricEncryptionPerformed")]`
		`public void SymmetricEncryptionPerformed(object sender, EventArgs e)`
		`{`
		`if (PerformanceCountersEnabled) symmetricEncryptionPerformedCounter.Increment();`
		`}`

Verify that the application block uses WMI and the event log to monitor errors during encryption, decryption, and hash operations when WMI and event log are enabled.	Yes	For example, the **SymmtericAlgorithmProvider.Encrypt** method uses WMI and the event log when the encryption operation fails. This is shown in the following code example. The process is similar for other operations.
		`public byte[] Encrypt(…)`
		`{`
		`try`
		`{`
		`output = crypto.Encrypt(plaintext);`
		`}`
		`catch (Exception e)`
		`{`
		`InstrumentationProvider.FireCyptographicOperationFailed(…);`
		`throw;`
		`}`
		`}`
		
		`[InstrumentationConsumer("CyptographicOperationFailed")]`
		`public void CyptographicOperationFailed(…)`
		`{`
		`if (EventLoggingEnabled)`
		`{`
		`EventLog.WriteEntry(GetEventSourceName(), entryText, EventLogEntryType.Error);`
		`}`
		`if (WmiEnabled) ManagementInstrumentation.Fire(new SymmetricOperationFailedEvent(…));`
		`}`

continued

Code test case	Implemented?	Feature that is implemented

Verify that the configuration properties of the providers are exposed as public and that they are strongly typed.	Yes	In the **CryptographySettings** class, the **SymmetricProviders** property can only contain a **SymmetricProviderData** collection. The following code demonstrates this. The **HashProviders** property is similar.
		`[ConfigurationProperty(symmetricCryptoProvidersProperty, IsRequired= false)]`
		`public NameTypeConfigurationElementCollection<SymmetricProviderData> SymmetricCryptoProviders`
		`{`
		`}`

Verify that the configuration properties for the custom providers are exposed as public and that they are implemented as custom property collections.	Yes	The **CustomSymmetricCryptoProviderData** class and the **CustomHashProviderData** class have references to the **CustomProviderDataHelper** class. This class defines a **NameValueCollection** class that holds the attributes in custom property collections. This is shown in the following code example.
		`private NameValueCollection attributes;`
		
		`private void AddAttributesFromConfigurationProperties()`
		`{`
		`foreach (ConfigurationProperty property in propertiesCollection)`
		`{`
		`…`
		`attributes.Add(property.Name, (string)helpedCustomProviderData.BaseGetPropertyValue(property));`
		`}`
		`}`

Verify that there is a way to remove decrypted keys from memory after they have been used.	Yes	The **CryptographicUtility.ZeroOutBytes** method clears the decrypted key from memory. This is shown in the following code example.
		`public byte[] Encrypt(…)`
		`{`
		`CryptographyUtility.ZeroOutBytes(this.algorithm.Key);`
		`}`

Code test case	Implemented?	Feature that is implemented

Verify that there is a way to properly dispose of the symmetric and hash algorithms after they are used.	Yes	The **SymmetricCryptographer.Dispose** method clears a symmetric algorithm. This is shown in the following code example.
		`protected virtual void Dispose(…)`
		`{`
		`if(algorithm != null)`
		`{`
		`algorithm.Clear();`
		`algorithm = null;`
		`}`
		`}`
		The **HashCryptographer** class implements a hash algorithm within a **using** statement. The **using** statement obtains resources, executes a statement, and disposes of the resources. In this case, the resource is the hash algorithm. This is shown in the following code example.
		`public byte[]`
		`ComputeHash(byte[] plaintext)`
		`{`
		`…`
		`using (HashAlgorithm`
		`algorithm = GetHashAlgorithm())`
		`{}`
		`…`
		`}`

Verify that the **SymmetricAlgorithmProvider** class and the **KeyedHashAlgorithmProvider** class support both machine mode and user mode encryption.	Yes	In the configuration source, the **scope** property supports both machine mode and user mode. This is shown in the following configuration example.
		`<symmetricCryptoProviders>`
		`<add scope="CurrentUser" type="…" name="…" />`
		`<add scope="LocalMachine" type="…" name="…" />`
		`</symmetricCryptoProviders>`
		The **CryptographySettings** class reads the configuration information and injects it into either the **KeyedHashAlgorithmProvider** class or the **SymmetricAlgorithmProvider** class (whichever is applicable). In addition, the constructor for either of these classes accepts a **DataProtectionScope** enumeration as a parameter. This parameter defines whether the scope of the protection is **CurrentUser** or **LocalMachine** mode. This is shown in the following code example.
		`public KeyedHashAlgorithmProvider (`
		`Type algorithmType,`
		`bool saltEnabled,`
		`string protectedKeyFileName,`
		`DataProtectionScope protectedKeyF-`
		`ileProtectionScope`
		`)`

continued

Code test case	Implemented?	Feature that is implemented

Verify that the application block can either use an absolute path or a relative path to read the key file.	Yes	In the configuration source, the **ProtectedKeyFilename** property supports both relative and absolute paths. This is shown in the following configuration example.
		`<hashProviders>`
		`<add protectedKeyFilename="hmac1-1.key" protectedKeyProtectionScope="…" algorithm-Type="…" saltEnabled="…" type="…" name="…" />`
		
		`<add protectedKeyFilename="C:\\HMacMd5.key" protectedKeyProtectionScope="…" algorithm-Type="…" saltEnabled="…" type="…" name="…" />`
		`</hashProviders>`

Verify that the application block validates the input at all the entry points, such as the **Cryptographer** façade.	Yes	The following code is an example of how the application block checks whether the input is valid. If the input is invalid, the application block throws an exception.
		`public static string DecryptSymmetric(string symmetricInstance, string ciphertextBase64)`
		`{`
		`if (string.IsNullOrEmpty(symmetricInstance))`
		`throw new ArgumentException(…);`
		
		`if (string.IsNullOrEmpty(ciphertextBase64))`
		`throw new ArgumentException(…);`
		
		`}`

Verify that the application block reads the symmetric key only once from the input stream and then caches it so that it can be used for the cryptography operations.	Yes	The **KeyManager.Read** method reads an encrypted key from the input stream and then caches it. This is shown in the following code example.
		`public static ProtectedKey Read(…)`
		`{`
		`if (cache[completeFileName] != null)`
		`return cache[completeFileName];`
		`}`

Verify that the symmetric key is cached in memory in a thread safe manner.	Yes	The **ProtectedKeyCache** class locks the cache before it stores the key. This is shown in the following code example.
		`public ProtectedKey this[string keyFileName]`
		`{`
		` set`
		` {`
		`  lock (cache)`
		`  {`
		`   cache[keyFileName] = value;`
		`  }`
		` }`
		`}`

Code test case	Implemented?	Feature that is implemented
Verify that the **HashProviderFactory** and the **SymmtericCryptoProviderFactory** classes create new instances of the symmetric and hash providers for each request.	Yes	The **HashProviderFactory** class and the **SymmtericCryptoProviderFactory** class derive from the **NameTypeFactoryBase** class. This class creates a new instance of a provider for each request.
Verify that the performance counters and the event log that are required by the application block are installed during installation.	Yes	For example, the **HashAlgorithmInstrumentationListener** class contains the installer attribute type **HasInstallableResourcesAttribute**. The **EventLogInstallerBuilder** and the **PerformanceCounterInstallerBuilder** installer classes, which are part of the Enterprise Library Core, recognize this attribute and install the performance counters and event logs.

To learn how the test teams tested the application blocks to see if they conformed to security best practices, see *Testing for Security Best Practices*. To learn how the test teams tested the application blocks to see if they conformed to globalization best practices, see *Testing for Globalization Best Practices*. To learn how the test teams tested the application blocks to see if they met the performance and scalability requirements, see *Testing for Performance and Scalability*.

Using Automated Tests

Automated tests ensure that the application block functions in accordance with its requirements. Automated tests make regression testing easier, and certain tests, such as simulating a large number of uses to test a multithreading scenario, require automation.

Table 5 lists the Visual Studio Team System tests that were used with the Cryptography Application Block.

Table 5: Visual Studio Team System Tests for the Cryptography Application Block

Test case	Result	Automated test

Verify that the application block throws the appropriate exception when it receives invalid data.	Passed	The following test uses a null message as input and expects an **ArgumentNullException** exception.
		`[ExpectedException(typeof(ArgumentNullException))]`
		`public void TestForNullPlainMessage()`
		`{`
		`Cryptographer.EncryptSymmetric(("DESCryptoServiceProvider", (String)null);`
		`}`

Verify that the application block encrypts valid input data.	Passed	The following test encrypts a string.
		`[TestMethod]`
		`public void TestEncryptionForValidInput()`
		`{`
		`String encryptedStrMessage = Cryptographer.EncryptSymmetric(("DESCryptoServiceProvider","Sample Text");`
		`Assert.IsFalse(encryptedStrMessage.Equals(""));`
		`Assert.IsFalse(encryptedStrMessage.Equals("SampleText"));`
		`}`

Verify that the symmetric and hash providers can be created directly, without using a configuration source.	Passed	The following test uses the **new** operator to directly create the hash provider and verifies that the object is created.
		`[TestMethod]`
		`public void TestHashAlgorithmProviderDirectly()`
		`{`
		`HashAlgorithmProvider provider = new HashAlgorithmProvider(Type.GetType("System.Security.Cryptography.SHA1Managed, mscorlib, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"), true);`
		`Assert.IsNotNull(provider);`
		`}`

Test case	Result	Automated test

Verify that a custom class can be added from information in the configuration source.	Passed	In this example, a custom hash provider, the **Custom Hash** class, implements the **IHashProvider** interface. The Enterprise Library Core uses information in the configuration source to add the custom class to the application block.
		CustomHash class
		
		[ConfigurationElementType(typeof(CustomHashProviderData))]
		public class CustomHash:IHashProvider
		{
		public NameValueCollection ObjCol;
		
		public CustomHash(NameValueCollection obj)
		{
		this.ObjCol = obj;
		}
		byte[] CreateHash(byte[] plaintext){…}
		bool CompareHash(byte[] plaintext, byte[] hashedtext){…}
		}
		The following is the information in the configuration file.
		<add key1="value1" key2="value2" type="….CustomHash, …" name="CustomHash1" />
		The following is the test case.
		[TestMethod]
		public void TestCustomHash()
		{
		HashProviderFactory factory = new HashProviderFactory(new SystemConfigurationSource());
		IHashProvider provider = factory.Create("CustomHash1");
		
		Assert.IsNotNull(provider);
		Assert.IsTrue(provider is CustomHash);
		}

continued

Test case	Result	Automated test

Verify that a provider that is configured with data in a dictionary configuration source can encrypt data.	Passed	This example uses a dictionary as the configuration source for a hash provider. The following code shows the dictionary configuration source.
		`DictionaryConfigurationSource section = new DictionaryConfigurationSource();`
		`CryptographySettings setting = new CryptographySettings();`
		`setting.HashProviders.Add(new HashAlgorithmProviderData("SHA512ManagedWithNoSalt",Type.GetType("System.Security.Cryptography.SHA512Managed, mscorlib, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"), false));`
		
		`section.Add("securityCryptographyConfiguration", setting);`
		The following is the test case.
		`[TestMethod]`
		`public void TestHashingWithDicSource()`
		`{`
		`HashProviderFactory factory = new HasProviderFactory(section);`
		`IHashProvider provider = (IHashProvider)factory.Create("SHA512ManagedWithNoSalt");`
		
		`byte[] hash = provider.CreateHash(PlainByteMessage);`
		`Assert.IsTrue(provider.CompareHash(PlainByteMessage, hash));`
		`}`

Testing the Data Access Application Block

This chapter explains how functional testing techniques were used to test the Data Access Application Block. If you have modified or extended the Data Access Application Block, you can use the same techniques and adapt the chapter's templates and checklists to test your own work.

Requirements for the Data Access Application Block

The Data Access Application Block has the following requirements:

- The application block should support common data access operations.
- The application block should be extensible.
- The application block should manage database connections.
- The application block should provide a straightforward way to handle parameters for stored procedures.
- The application block should be able to read configuration information from any configuration source, such as an XML file or a database.
- The application block should support configurable instrumentation, including WMI (Windows Management Instrumentation), performance counters, and event logs.
- The application block should have good performance metrics.
- The application block should work with desktop applications and with Web applications.

These requirements must be incorporated into the design and implemented by the code.

Selecting the Test Cases

The first step in a functional review is to verify that the design and the code support the requirements. You do this by deciding the test cases that the design and code must satisfy. Table 1 lists the test cases that the Data Access Application Block's design must satisfy.

Table 1: Data Access Application Block Design Test Cases

Priority	Design test case
High	Verify that the database providers are extensible.
High	Verify that there is a consistent approach to creating any database provider, such as a SQL Server database, Oracle database, or a generic database.
High	Verify that the application block supports database-specific features that are implemented in the appropriate **Database**-derived class. For example, the **SqlDatabase** class supports the ability to read data that is in XML format.
High	Verify that the application block supports the parameter discovery feature.
High	Verify that the ability to create the application block's domain objects from configuration data follows the Dependency Injection pattern.
High	Verify that the application block can retrieve configuration data from different sources, such as an application configuration file, a database, or from memory.
High	Verify that the application block can manage database connections.
High	Verify that the instrumentation is implemented with loosely coupled events.
High	Verify that the design addresses situations that can cause exceptions and that the application block logs the exceptions through the instrumentation.

After you identify the design issues, you should do the same for the code. Table 2 lists the test cases that the Data Access Application Block's code must satisfy.

Table 2: Data Access Application Block Code Test Cases

Priority	Code test case
High	Verify that the **DatabaseProviderFactory** class creates a new **Database**-derived instance for each request.
High	Verify that an assembler class that implements the **IDatabaseAssembler** interface creates the **Database**-derived objects and injects the configuration object values into those domain objects.
High	Verify that the application block uses performance counters to monitor database operations when the performance counters are enabled.
High	Verify that the application block uses WMI and the event log to monitor errors during database operations when WMI and the event log are enabled.
High	Verify that the connection string property is configurable and is implemented as a connection string section that is supported by the .NET Framework.
High	Verify that the application block can use an instance name to create a **Database**-derived object.
High	Verify that the application block can use a default instance name to create a **Database**-derived object.
High	Verify that the application block exposes configuration properties for Oracle packages as public and that they are configurable.
High	Verify that the application block caches parameter discovery information and then retrieves it from the cache when it is required.

Priority	Code test case
High	Verify that the parameter naming convention for a particular database is handled by the application block instead of being included in the application code. For example, a SQL Server database requires that parameter names begin with the "@" character. This should be appended by the application block so that the code remains portable.
High	Verify that the performance counters and the event log that are required by the application block are installed during installation.
Medium	Verify that the application block requests or demands the appropriate code access security permissions for access to protected system resources and operations.
Medium	Verify that the application block follows exception management best practices.
High	Verify that the application block follows security best practices.
Medium	Verify that the application block follows globalization best practices.
High	Verify that the application block follows performance best practices.

Verifying the Test Cases

After you identify all the design test cases, you can verify that the design satisfies them. Table 3 lists how each of the design test cases were verified for the Data Access Application block.

Table 3: Data Access Application Block Design Verification

Design test case	Implemented?	Feature that implements design
Verify that the database providers are extensible.	Yes	The **SqlDatabase** class and the **OracleDatabase** class derive from the **Database** base class. This class defines a common interface that users can extend or modify to develop their own custom database providers.
Verify that there is a consistent approach to creating any database provider, such as a SQL Server database, Oracle database, or a generic database.	Yes	The **DatabaseProviderFactory** class is the factory that creates all objects that derive from the **Database** class.
Verify that the application block supports database-specific features that are implemented in the appropriate **Database**-derived class. For example, the **SqlDatabase** class supports the ability to read data that is in XML format.	Yes	For example, there is a **SqlDatabase.ExecuteXMLReader** method that allows the application block to access XML data stored in a SQL Server database.

continued

Design test case	Implemented?	Feature that implements design
Verify that the application block supports the parameter discovery feature.	Yes	The **Database** base class implements the **DiscoverParameters** method that dynamically discovers the parameter types.
Verify that the ability to create the application block's domain objects from configuration data follows the Dependency Injection pattern.	Yes	The **DatabaseProviderFactory** class derives from the **NameTypeFactoryBase** class, which takes the configuration source as input, creates the domain object, and injects the relevant configuration data into the domain object.
Verify that the application block can retrieve configuration data from different sources, such as an application configuration file, a database, or from memory.	Yes	The **DatabaseProviderFactory** class has a constructor that accepts a configuration source as an input parameter.
Verify that the application block can manage database connection.	Yes	The application block closes the database connections after it is finished with them. For example, the implementation of the **ExecuteNonQuery** method includes a **using** statement. The **using** statement obtains resources, executes a statement, and disposes of the resources. In this case, the resource is the database connection. In the case of the **ExecuteReader** method, the application block uses the **CommandBehavior.CloseConnection** method to close the connection after the reader closes.
Verify that the instrumentation is implemented with loosely coupled events.	Yes	The methods in the **DataInstrumentationProvider** class that fire the events bind to the methods in the **DataInstrumentationListener** class at run time.
Verify that the design addresses situations that can cause exceptions and that the application block logs the exceptions through the instrumentation.	Yes	For example, the **Database** class includes the **InstrumentationProvider** property. This property retrieves the instrumentation provider that defines the events for the database provider. The provider logs the events to WMI and the event log.

After the code is implemented, you can review it to see if it satisfies its test cases. Table 4 lists the results of a code review for the Data Access Application Block.

Table 4: Data Access Application Block Code Verification

Code test case	Implemented?	Feature that is implemented
Verify that the **DatabaseProviderFactory** class creates a new instance of a **Database**-derived object for each request.	Yes	The **DatabaseFactory** class calls the **DatabaseProviderFactory** class, which derives from the **NameTypeFactory** base class. For each request, this class creates a new instance of a database provider that derives from the **Database** class.

Verify that an assembler class that implements the **IDatabaseAssembler** interface creates the **Database**-derived objects and injects the configuration object values into those domain objects.	Yes	The following code demonstrates how the **SqlDatabaseAssembler** class, which implements the **IDatabaseAssembler** interface, creates a SQL Server database provider. The process is similar for Oracle databases and generic databases.
		`public class SqlDatabaseAssembler : IDatabaseAssembler`
		`{`
		`public Database Assemble(string name, ConnectionStringSettings connectionStringSettings, IConfigurationSource configurationSource)`
		`{`
		`return new SqlDatabase(…);`
		`}`
		`}`

Verify that the application block uses performance counters to monitor database operations when the performance counters are enabled.	Yes	For example, the **Database.OpenConnection** method increments the **Connections Opened/sec** performance counter when a database connection succeeds. This is shown in the following code example.
		`protected DbConnection OpenConnection()`
		`{`
		`connection = CreateConnection();`
		`connection.Open();`
		`instrumentationProvider.FireConnectionOpenedEvent();`
		`}`
		The **FireConnectionOpenedEvent** method raises the **connectionOpened** event. The **ConnectionOpened** method consumes this event and increments the performance counter. This is shown in the following code example.
		`[InstrumentationConsumer("ConnectionOpened")]`
		`public void ConnectionOpened(…)`
		`{`
		`if (PerformanceCountersEnabled) connectionOpenedCounter.Increment();`
		`}`

continued

Code test case	Implemented?	Feature that is implemented

Verify that the application block uses WMI and the event log to monitor errors during database operations when WMI and the event log are enabled.	Yes	For example, the **Database.OpenConnection** method uses WMI, the event log, and performance counters when the connection fails. This is shown in the following code example.
		`protected DbConnection OpenConnection()`
		`{`
		`try`
		`{`
		`connection = CreateConnection();`
		`connection.Open();`
		`}`
		`catch (Exception e)`
		`{`
		`instrumentationProvider.FireConnectionFailedEvent(ConnectionStringNoCredentials, e);`
		`throw;`
		`}`
		`}`
		The **FireConnectionFailedEvent** method raises the **connectionFailed** event. The **ConnectionFailed** method consumes this event and notifies WMI, logs the exception to the event log, and increments the **Connections failed/sec** performance counter. This is shown in the following code example.
		`[InstrumentationConsumer("ConnectionFailed")]`
		`public void ConnectionFailed(…)`
		`{`
		`if (PerformanceCountersEnabled) connectionFailedCounter.Increment();`
		`if (WmiEnabled) System.Management.Instrumentation.Instrumentation.Fire(new ConnectionFailedEvent(…);`
		`if (EventLoggingEnabled)`
		`{`
		`…`
		`EventLog.WriteEntry(GetEventSourceName(), entryText, EventLogEntryType.Error);`
		`}`
		`}`

Code test case	Implemented?	Feature that is implemented

Verify that the connection string property is configurable and is implemented as a connection string section that is supported by the .NET Framework.	Yes	In the application configuration file, the connection string is exposed as a <connectionStrings> section that is supported by the .NET Framework. This is shown in the following XML example.
		`<connectionStrings>`
		`<add name="DataSQLTest"`
		`providerName="System.Data.SqlClient"`
		`connectionString="server=(local)\sqlexpress;database=TestDatabase;Integrated Security=true" />`
		`</connectionStrings>`
		The **GetConnectionStringSettings** method on the **DatabaseConfigurationView** class reads the connection string section from the configuration source. This is shown in the following code example.
		`public ConnectionStringSettings GetConnectionStringSettings(string name)`
		`{`
		`…`
		`ConfigurationSection configSection = configurationSource.GetSection("connectionStrings");`
		`if ((configSection != null) && (configSection is ConnectionStringsSection))`
		`{`
		`ConnectionStringsSection connectionStringsSection = configSection as ConnectionStringsSection;`
		`connectionStringSettings = connectionStringsSection.`
		`ConnectionStrings[name];`
		`}`
		`else`
		`connectionStringSettings = ConfigurationManager.ConnectionStrings[name];`
		`…`
		`}`

Verify that the application block can use an instance name to create a **Database**-derived object.	Yes	The **DatabaseFactory.Create** method has two overloads. One of the methods accepts an instance name as a parameter to create a **Database**-derived object for the specified instance name. This is shown in the following code example.
		`public static Database CreateDatabase(string name)`
		`{`
		`}`

continued

Code test case	Implemented?	Feature that is implemented

Verify that the application block can use a default instance name to create a **Database**-derived object.	Yes	The **DatabaseFactory.Create** method has two overloads. One of the methods creates a **Database**-derived object from the default instance name in the configuration source. This is shown in the following code example.
		`public static Database CreateDatabase()`
		`{`
		`}`

Verify that the application block exposes configuration properties for Oracle packages as public and that they are configurable.	Yes	The <oracleConnectionSettings> section in the configuration source exposes the configuration properties for Oracle packages. This is shown in the following configuration example.
		`<oracleConnectionSettings>`
		`<add name="OracleInstance">`
		`<packages>`
		`<add name="…" prefix="…" />`
		`</packages>`
		`</add>`
		`</oracleConnectionSettings>`
		The **OracleConnectionData**, **OraclePackageData** and **OracleConnectionSettings** classes read this configuration information.

Verify that the application block caches parameter discovery information and then retrieves it from the cache when it is required.	Yes	The **ParameterCache** class caches the parameters. The **SetParameters** method populates the parameter collection from the cache if the parameters are already stored there. This is shown in the following code example.
		`public void SetParameters(DbCommand command, Database database)`
		`{`
		`…`
		`if (AlreadyCached(command, database))`
		`{`
		`AddParametersFromCache(command, database);`
		`}`
		`else`
		`{`
		`database.DiscoverParameters(command);`
		`}`

Code test case	Implemented?	Feature that is implemented

Verify that the parameter naming convention for a particular database is handled by the application block instead of being included in the application code. For example, a SQL Server database requires that parameter names begin with the "@" character. This should be appended by the application block so that the code remains portable.	Yes	The **Database** class contains the **BuildParameterName** virtual method. In the case of a SQL Server database, parameter names begin with the "@" symbol. The **SqlDatabase** class overrides the **BuildParameterName** method and appends the "@" symbol. This is shown in the following code example.
		`public override string BuildParameterName(string name)`
		`{`
		`if (name[0] != this.ParameterToken)`
		`{`
		`return name.Insert(0, new string(this.ParameterToken, 1));`
		`}`
		`return name;`
		`}`

Verify that the performance counters and the event log that are required by the application block are installed during installation.	Yes	For example, the **DataInstrumentationListener** class contains the installer attribute type **[HasInstallableResourcesAttribute]**. The **EventLogInstallerBuilder** and **PerformanceCounterInstallerBuilder** installer classes recognize this attribute and install the performance counters and event logs.

To learn how the test teams tested the application blocks to see if they conformed to security best practices, see *Testing for Security Best Practices*. To learn how the test teams tested the application blocks to see if they conformed to globalization best practices, see *Testing for Globalization Best Practices*. To learn how the test teams tested the application blocks to see if they met the performance and scalability requirements, see *Testing for Performance and Scalability*.

Using Automated Tests

Automated tests ensure that the application block functions in accordance with its requirements. Automated tests make regression testing easier and certain tests, such as simulating a large number of uses to test a multithreading scenario, require automation.

Table 5 lists the Visual Studio Team System tests that were used with the Data Access Application Block.

Table 5: Visual Studio Team System Tests for the Data Access Application Block

Test case	Result	Automated test

Verify that the application block throws the appropriate exception when it receives invalid data.	Passed	The following test uses a null command.
		`[TestMethod]`
		`[ExpectedException(typeof(System.ArgumentNullException))]`
		`public void NullCommandTest()`
		`{`
		`Database db = DatabaseFactory.CreateDatabase("OracleTest");`
		`DbCommand dbCommand = null;`
		`db.ExecuteScalar(dbCommand);`
		`}`

Verify that the application block successfully executes a query when it receives valid data.	Passed	For example, the following test case tests the **ExecuteScalar** method with a **DbCommand** object as input.
		`[TestMethod]`
		`public void ExecuteScalarDBCommandSPTest()`
		`{`
		`Database db = DatabaseFactory.CreateDatabase("DataSQLTest");`
		`string spName = "ItemsDescriptionGet";`
		
		`DbCommand dbCommandWrapper = db.GetStoredProcCommand(spName);`
		`object actualResult = db.ExecuteScalar(dbCommandWrapper);`
		`Assert.AreEqual(actualResult.ToString().Trim(),"Digital Image Pro");`
		`}`

Verify that **Database**-derived classes, such as **SqlDatabase** and **OracleDatabase**, can be created directly, without using a configuration source.	Passed	The following test uses a constructor to create a **SqlDatabase** object.
		`[TestMethod]`
		`public void ExecuteScalarDBCommandSPDirectTest()`
		`{`
		`SqlDatabase db = new SqlDatabase(@"server=(local)\sqlexpress;database=TestDatabase;Integrated Security=true");`
		
		`Assert.IsNotNull(db);`
		`}`

Test case	Result	Automated test

Verify that database providers, such as OLEDB and ODBC, work with the application block.	Passed	The following is the information in the configuration file. It tells the application block that it should use an OLEDB provider.
		`<connectionStrings>`
		`<add`
		`name="GenericSQLTest"`
		`providerName="System.Data.OleDb"`
		`connectionString="Provider=SQLOLEDB;Data Source=(local)\sqlexpress;Initial Catalog=TestDatabase;Integrated Security=SSPI" />`
		The following is the test.
		`[TestMethod]`
		`public void NonQueryDBCmdQueryTest()`
		`{`
		`bool isPresent = false;`
		`Database db = DatabaseFactory.CreateDatabase("GenericSQLTest");`
		
		`string sqlCommand = "Insert into CustomersOrders values(13, 'John',3, 214)";`
		
		`DbCommand dbCommandWrapper = db.GetSqlStringCommand(sqlCommand);`
		`db.ExecuteNonQuery(dbCommandWrapper);`
		
		`SqlConnection conn = new SqlConnection();`
		
		`conn.ConnectionString = connStr;`
		`conn.Open();`
		`SqlCommand cmd = new SqlCommand("select * from CustomersOrders where CustomerName = 'John'", conn);`
		`SqlDataReader dr = cmd.ExecuteReader();`
		`while (dr.Read())`
		`{`
		`Assert.AreEqual(3, Convert.ToInt32(dr["ItemId"].ToString()));`
		`Assert.AreEqual(214, Convert.ToInt32(dr["QtyOrdered"].ToString()));`
		`isPresent = true;`
		`}`
		`dr.Close();`
		`Assert.AreEqual(true, isPresent);`
		`}`

continued

Test case	Result	Automated test

Verify that custom database classes that are derived from the **Database** class can be added to the application block and configured from information in the configuration source.	Passed	In this example, the custom database class **CustomMockDatabase** derives from the **Database** class. The Enterprise Library Core uses information in the configuration file to add the custom class to the application block.
		`[DatabaseAssembler(typeof(CustomDatabaseCustomAssembler))]`
		`public class CustomMockDatabase : Database`
		`{`
		`public CustomMockDatabase(string connectionString):base(connectionString,SqlClientFactory.Instance)`
		`{`
		`…`
		`}`
		
		`protected override void DeriveParameters(System.Data.Common.DbCommand discoveryCommand)`
		`{`
		`…`
		`);`
		`}`
		
		`public class CustomDatabaseCustomAssembler : IDatabaseAssembler`
		`{`
		`public Database Assemble(string name, System.Configuration.ConnectionStringSettings connectionStringSettings, Microsoft.Practices.EnterpriseLibrary.Common.Configuration.IConfigurationSource configurationSource)`
		`{`
		`return new CustomMockDatabase(connectionStringSettings.ConnectionString);`
		`}`
		`}`
		The following is the information in the configuration file.
		`<connectionStrings>`
		`<add name="CustomDatabase"`
		`providerName="CustomDatabase"`
		`connectionString="…" />`
		`<connectionStrings>`
		`<providerMappings>`
		`<add databaseType="….CustomMockDatabase,…"`
		`name="CustomDatabase"/>`
		`</providerMappings>`
		The following is the test case.
		`[TestMethod]`
		`public void CreateCustomDatabaseTest()`
		`{`
		`Database db= DatabaseFactory.CreateDatabase("CustomDatabase");`
		`Assert.IsNotNull(db);`
		`}`

Testing the Exception Handling Application Block

This chapter explains how functional testing techniques were used to test the Exception Handling Application Block. If you have modified or extended the Exception Handling Application Block, you can use the same techniques and adapt the chapter's templates and checklists to test your own work.

Requirements for the Exception Handling Application Block

The Exception Handling Application Block has the following requirements:

- The application block should support commonly used exception handling operations, such as logging the exceptions, replacing the original exception with another exception and wrapping the original exception with another exception.
- The application block should be able to combine the exception handlers. For example, it should be possible to log the exception information and then replace the original exception with another.
- The application block should support configurable instrumentation, including WMI (Windows Management Instrumentation), performance counters, and event logs.
- The application block should be extensible.
- The application block should be able to read configuration information from any configuration source, such as an XML file or a database.
- The application block should work with desktop applications and with Web applications.

These requirements must be incorporated into the design and implemented by the code.

Selecting the Test Cases

The first step in a functional review is to make sure that the design and the code support these requirements. You do this by deciding the test cases that the design and code must satisfy. Table 1 lists the test cases that the application block's design must satisfy.

Table 1: Exception Handling Application Block Design Test Cases

Priority	Design test case
High	Verify that the exception handlers and the exception formatters are extensible.
High	Verify that there is a consistent approach to creating exception policies, exception handlers, and exception formatters.
High	Verify that there is a façade that mediates between the client code and the application block to handle exceptions.
High	Verify that the application block generates a new GUID named the **HandlingInstanceID** for every exception that is handled.
High	Verify that the ability to create the application block's domain objects from the configuration data follows the Dependency Injection pattern.
High	Verify that the application block can retrieve configuration data from different sources, such as an application configuration file, a database, or from memory.
High	Verify that the instrumentation uses loosely coupled events.
High	Verify that situations that can cause exceptions are addressed and that the application block logs the exceptions through the instrumentation.
High	Verify that the application block supports custom property collections for the custom exception handlers.

After you identify the design issues, you should do the same for the code. Table 2 lists the test cases that the Exception Handling Application Block code must satisfy.

Table 2: Exception Handling Application Block Code Test Cases

Priority	Code test case
High	Verify that the **ExceptionPolicy** façade exposes a public method to handle the exceptions.
High	Verify that the **ExceptionPolicy** façade uses the **ExceptionPolicyFactory** class to create the exception handling provider.
High	Verify that an assembler class creates exception handlers and injects the configuration object values into those domain objects.
High	Verify that the application block uses performance counters to monitor exception handling operations when the performance counters are enabled.
High	Verify that the application block uses WMI and the event log to monitor errors during exception handling operations when the WMI and event log are enabled.
High	Verify that the configuration properties of the exception handling policies are exposed as public and that they are strongly typed.
High	Verify that the configuration properties for the custom exception handlers are exposed as public and that they are implemented as a collection.
Medium	Verify that the application block validates the input at all the entry points, such as the **ExceptionPolicy** façade.
High	Verify that the **ExceptionPolicyFactory** class creates only a single instance of the exception handling provider for all the requests.
High	Verify that the application block supports post handling actions, such as **None**, **NotifyRethrow**, and **ThrowNewException**, after it executes the exception handlers.

Priority	Code test case
High	Verify that the application block uses the Logging Application Block to log the exceptions.
High	Verify that the performance counters and the event log that are required by the application block are installed during installation.
Medium	Verify that the application block requests or demands the appropriate code access security permissions to access protected system resources and operations.
High	Verify that the application block follows exception management best practices.
High	Verify that the application block follows security best practices.
Medium	Verify that the application block follows globalization best practices.
High	Verify that the application block follows performance best practices.

Verifying the Test Cases

After you identify all the design test cases, you can verify that the design satisfies them. Table 3 lists how each of the design test cases were verified for the Exception Handling Application Block.

Table 3: Exception Handling Application Block Design Verification

Design test case	Implemented?	Feature that implements design
Verify that the exception handlers and the exception formatters are extensible.	Yes	The **IExceptionHandler** interface allows users to implement a configurable exception handler. The **ExceptionFormatter** class allows users to extend an exception formatter.
Verify that there is a consistent approach to creating exception policies, exception handlers, and exception formatters.	Yes	The **ExceptionHandlerCustomFactory** class is the factory that creates the objects that implement the **IExceptionHandler** interface.
Verify that there is a façade that mediates between the client code and the application block to handle exceptions.	Yes	The **ExceptionPolicy** class is a façade that acts as the interface between the client code and the application block.
Verify that the application block generates a new GUID named the **HandlingInstanceID** for every exception that is handled.	Yes	The **ExceptionPolicyEntry.Handle** method generates a new GUID named the **HandlingInstanceID** and passes that GUID to the exception handlers.
Verify that the ability to create the application block's domain objects from the configuration data follows the Dependency Injection pattern.	Yes	The **ExceptionPolicyFactory** class derives from the **LocatorNameTypeFactoryBase** generic type, which takes the configuration source as input, creates the domain object, and injects the relevant configuration data into the domain object.

continued

Design test case	Implemented?	Feature that implements design
Verify that the application block can retrieve configuration data from different sources, such as an application configuration file, a database, or from memory.	Yes	The **ExceptionPolicyFactory** class has a constructor that accepts a configuration source as an input parameter.
Verify that the instrumentation is implemented with loosely coupled events.	Yes	The methods in the **ExceptionHandlingInstrumentationProvider** class that raise the events bind to the methods in the **ExceptionHandlingInstrumentationListener** class at run time.
Verify that situations that can cause exceptions are addressed and that the application block logs the exceptions through the instrumentation.	Yes	For example, the **ExceptionPolicyImpl** class, which handles the exception policies, includes the **InstrumentationProvider** property. This property retrieves the instrumentation provider that defines the events for exception handling. The provider logs the events to WMI and the event log.
Verify that the application block supports custom property collections for the custom exception handlers.	Yes	The **CustomHandlerData** class provides the ability to configure custom property collections for custom exception handlers.

After the code is implemented, you can review it to see if it satisfies its test cases. Table 4 lists the results of a code review for the Exception Handling Application Block.

Table 4: Exception Handling Application Block Code Verification

Code test case	Implemented?	Feature that is implemented
Verify that the **ExceptionPolicy** façade exposes a public method to handle the exceptions.	Yes	The **ExceptionPolicy** class is a façade that exposes the **HandleException** method to handle an exception. This is shown in the following code. `public static bool HandleException(Exception exceptionToHandle, string policyName){}`

Code test case	Implemented?	Feature that is implemented

Verify that the **ExceptionPolicy** façade uses the **ExceptionPolicyFactory** class to create an **ExceptionPolicyImpl** object, which handles exception policies.	Yes	The client code calls static methods on the **ExceptionPolicy** class to handle exceptions. The static method instantiates a factory class. The factory uses the configuration data from the configuration file and creates the exception policies. The following code demonstrates how the **ExceptionPolicy.HandleException** method calls the **ExceptionPolicyFactory** class to create an **ExceptionPolicyImpl** object that handles the exception.
		`public static class ExceptionPolicy`
		`{`
		`private static readonly ExceptionPolicyFac-`
		`tory factory = new ExceptionPolicyFactory();`
		
		`public static bool HandleException(Exception`
		`exceptionToHandle, string policyName)`
		`{`
		`…`
		`ExceptionPolicyImpl policy = GetExceptionPoli`
		`cy(exceptionToHandle, policyName);`
		
		`…`
		
		`}`
		
		`private static ExceptionPolicyImpl GetExcept`
		`ionPolicy(Exception exception, string policy-`
		`Name)`
		`{`
		`…`
		`return factory.Create(policyName);`
		`…`
		
		`}`

continued

Code test case	Implemented?	Feature that is implemented

Verify that an assembler class that implements the **IAssembler** interface creates the exception handlers and injects the configuration object values into those domain objects.	Yes	The following code demonstrates how the **LoggingExceptionHandlerAssembler** class, which implements the **IAssembler** interface, creates a **LoggingExceptionHandler** object. The **Assemble** method in the **LoggingExceptionHandlerAssembler** class injects the configuration data from the **LoggingExceptionHandler** object into the **LoggingExceptionHandler** domain object. The process is similar for all the exception handlers that implement the **IExceptionHandler** interface.
		`public class LoggingExceptionHandlerAssembler : IAssembler<IExceptionHandler, ExceptionHandlerData>`
		`{`
		`public IExceptionHandler Assemble(IBuilderContext context, ExceptionHandlerData objectConfiguration, IConfigurationSource configurationSource, ConfigurationReflectionCache reflectionCache)`
		`{`
		`LoggingExceptionHandlerData castedObjectConfiguration= (LoggingExceptionHandlerData)objectConfiguration;`
		`…`
		`LoggingExceptionHandler createdObject = new LoggingExceptionHandler(castedObjectConfiguration.LogCategory,castedObjectConfiguration.EventId,castedObjectConfiguration.Severity,castedObjectConfiguration.Title,castedObjectConfiguration.Priority,castedObjectConfiguration.FormatterType,writer);`
		
		`return createdObject;`
		`}`
		`}`

Code test case	Implemented?	Feature that is implemented

Verify that the application block uses performance counters to monitor exception handling when the performance counters are enabled.	Yes	For example, the **ExceptionPolicyEntry.Handle** method increments the **Exceptions Handled/sec** performance counter whenever the application block successfully handles the exceptions according to the exception policy. This is shown in the following code example.
		`public bool Handle(Exception exceptionToHandle)`
		`{`
		`…`
		`Exception chainException = ExecuteHandlerChain(exceptionToHandle, handlingInstanceID);`
		
		`if (InstrumentationProvider != null) InstrumentationProvider.FireExceptionHandledEvent();`
		`…`
		`}`
		The **ExceptionHandlingInstrumentationProvider** class fires the **exceptionHandled** event. The **ExceptionHandled** method consumes the event and increments the performance counters. This is shown in the following code example.
		`[InstrumentationConsumer("ExceptionHandled")]`
		`public void ExceptionHandled(…)`
		`{`
		`if (PerformanceCountersEnabled) exceptionHandledCounter.Increment();`
		`}`

continued

Code test case	Implemented?	Feature that is implemented

Verify that the application block uses WMI and the event log to monitor errors during exception handling operations when the WMI and event log are enabled.	Yes	For example, the **ExceptionPolicyEntry.Handle** method calls the private method **ExecuteHandlerChain** to execute the exception handlers that are defined in the exception policy. If an exception occurs while the **ExecuteHandlerChain** method handles the exceptions, the **FireExceptionHandlingErrorOccurred** method notifies WMI and the event log. The **FireExceptionHandlingErrorOccurred** method sends the error message to the **ExceptionHandlingFailureEvent** WMI event and logs the error to the Enterprise Library Exception Handling event log. This is shown in the following code example.
		`private Exception ExecuteHandlerChain(Exception ex, Guid handlingInstanceID)`
		`{`
		`try`
		`{`
		`…`
		`ex = handler.HandleException(ex, handlingInstanceID);`
		`}`
		`catch (…)`
		`{`
		`…`
		`InstrumentationProvider.FireExceptionHandlingErrorOccurred(…));`
		`…`
		`}`
		The **ExceptionHandlingInstrumentationProvider** class fires the **exceptionHandlingErrorOccured** event. The **ExceptionHandlingErrorOccured** method consumes the event and notifies WMI and logs the exception to the event log. This is shown in the following code example.
		`[InstrumentationConsumer("ExceptionHandlingErrorOccurred")]`
		`public void ExceptionHandlingErrorOccurred(…)`
		`{`
		`if (EventLoggingEnabled)`
		`{`
		`…`
		`EventLog.WriteEntry(GetEventSourceName(), entryText, EventLogEntryType.Error);`
		`}`
		`if (WmiEnabled) ManagementInstrumentation.Fire(new ExceptionHandlingFailureEvent(instanceName, e.Message));`
		`}`

Code test case	Implemented?	Feature that is implemented

Verify that the configuration properties of the exception handling policies are exposed as public and that they are strongly typed.	Yes	For example, in the **ExceptionHandlingSettings** class, the **ExceptionPolicies** property can only contain an **ExceptionPolicyData** collection. The following code demonstrates this.
		`[ConfigurationProperty(policiesProperty)]`
		`public NamedElementCollection<ExceptionPolicyData> ExceptionPolicies`
		`{`
		`get { return (NamedElementCollection<ExceptionPolicyData>)this[policiesProperty];`
		`}`
		`}`

Verify that the configuration properties for the custom exception handlers are exposed as public and that they are implemented as a collection.	Yes	The **CustomHandlerData** class has a reference to the **CustomProviderDataHelper** class. This class adds the properties that are defined in the configuration source to an instance of the **NameValueCollection** class. This is shown in the following code example.
		`private NameValueCollection attributes;`
		
		`private void AddAttributesFromConfigurationProperties()`
		`{`
		`foreach (ConfigurationProperty property in propertiesCollection)`
		`{`
		`…`
		`attributes.Add(property.Name, (string)helpedCustomProviderData.BaseGetPropertyValue(property));`
		`}`
		`}`

Verify that the application block validates the input at all the entry points, such as the **ExceptionPolicy** façade.	Yes	The **ExceptionPolicy** façade validates the input parameters that are passed to the **HandleException** method. This is shown in the following code example.
		`public static bool HandleException(Exception exceptionToHandle, string policyName)`
		`{`
		`if (exceptionToHandle == null) throw new ArgumentNullException("exceptionToHandle");`
		
		`if(string.IsNullOrEmpty(policyName))`
		`throw new ArgumentException(…);`
		`}`

Verify that the **ExceptionPolicyFactory** class only creates a single instance of the exception handling provider for all requests.	Yes	The **ExceptionPolicyFactory** class derives from the **LocatorNameTypeFactoryBase** class. This class only creates a single instance for all requests.

continued

Code test case	Implemented?	Feature that is implemented

Verify that the application block supports post handling actions such as **None**, **NotifyRethrow**, and **ThrowNewException** after it executes the exception handlers.	Yes	The **ExceptionPolicyEntry** class executes the exception handlers for the given exception. This class contains a method named **Handle** that calls the private method **RethrowRecommended**. This method executes the post handling action. This is shown in the following code example.
		`private bool RethrowRecommended(Exception chainException, Exception originalException)`
		`{`
		`if (postHandlingAction == PostHandlingAction.None) return false;`
		
		`if (postHandlingAction == PostHandlingAction.ThrowNewException)`
		`{`
		`throw IntentionalRethrow(chainException, originalException);`
		`}`
		`return true;`
		`}`

Verify that the application block uses the Logging Application Block to log the exceptions.	Yes	The **LoggingExceptionHandler** class logs the exceptions. This class has a method named **WriteToLog** that uses an instance of the Logging Application Block's **LogWriter** class. This is shown in the following code example.
		`public class LoggingExceptionHandler : IExceptionHandler`
		`{`
		`private readonly LogWriter logWriter;`
		
		`protected virtual void WriteToLog(string logMessage, IDictionary exceptionData)`
		`{`
		`this.logWriter.Write(entry);`
		`}`
		`}`

Verify that the performance counters and the event log that are required by the application block are installed during installation.	Yes	For example, the **ExceptionHandlingInstrumentationListener** class contains the installer attribute type **[HasInstallableResourcesAttribute]**. The **EventLogInstallerBuilder** and the **PerformanceCounterInstallerBuilder** installer classes, which are in the Enterprise Library Core, recognize this attribute and install the performance counters and event logs.

To learn how the test teams tested the application blocks to see if they conformed to security best practices, see *Testing for Security Best Practices*. To learn how the test teams tested the application blocks to see if they conformed to globalization best practices, see *Testing for Globalization Best Practices*. To learn how the test teams tested the application blocks to see if they met the performance and scalability requirements, see *Testing for Performance and Scalability*.

Using Automated Tests

Automated tests ensure that the application block functions in accordance with its requirements. Automated tests make regression testing easier and certain tests, such as simulating a large number of uses to test a multithreading scenario, require automation.

Table 5 lists the Visual Studio Team System tests that were used with the Exception Handling Application Block.

Table 5: Visual Studio Team System Tests for the Exception Handling Application Block

Test case	Result	Automated test

Verify that the application block throws the appropriate exception when it receives invalid data.	Passed	The following test uses a null policy.
		`[TestMethod]`
		`[ExpectedException(typeof(ArgumentException))]`
		`public void NoPolicyTest()`
		`{`
		`Exception originalException = new System.Security.SecurityException("No Policy defined");`
		`bool rethrow = ExceptionPolicy.HandleException(originalException, null);`
		`}`

continued

Test case	Result	Automated test

Verify that the post handling action can be configured as **NotifyRethrow**, **None**, or **ThrowNewException**. Also verify that the exception handler executes the correct post handling action.	Passed	In the following test, the post handling action is **NotifyRethrow**. This means that after the exception handler executes, the application block should return **true** to the calling code. There are similar tests for the other post handling actions.
		`<exceptionHandling>`
		`<exceptionPolicies>`
		`<add name="Wrap Rethrow">`
		`<exceptionTypes>`
		`<add name="FileNotFoundException" type="System.IO.FileNotFoundException, …" postHandlingAction="NotifyRethrow">`
		`<exceptionHandlers>`
		`<add name="Wrap Handler" type="Microsoft.Practices.EnterpriseLibrary.ExceptionHandling.WrapHandler, Microsoft.Practices.EnterpriseLibrary.ExceptionHandling" exceptionMessage="…" wrapExceptionType="…" />`
		`</exceptionHandlers>`
		`</add>`
		`</exceptionTypes>`
		`</add>`
		`</exceptionPolicies>`
		`</exceptionHandling>`
		
		`[TestMethod]`
		`public void WrapPostHandlingRethrowTest()`
		`{`
		`bool rethrow = false;`
		`Exception originalException = new System.IO.FileNotFoundException("Original Exception: File Not found exception");`
		`rethrow = ExceptionPolicy.HandleException(originalException, "Wrap Rethrow");`
		`Assert.IsTrue(rethrow);`
		`}`

Test case	Result	Automated test

Verify that the logging exception handlers log the exceptions to the trace listeners that are in the configuration source.	Passed	In the following test, the **LoggingExceptionHandler** class is configured to log exceptions to the event log. (The Logging Application Block configuration is not shown.) The test case verifies that the exception is logged to the event log. Similar tests exist for other handlers, such as the **WrapHandler** and the **ReplaceHandler** classes.
		The following is the configuration information.
		`<exceptionHandling>`
		`<exceptionPolicies>`
		`<add name="Logging To Event Sink PostHandlingAction None">`
		`<exceptionTypes>`
		`<add name="SecurityException" type="System.Security.SecurityException, …" postHandlingAction="None">`
		`<exceptionHandlers>`
		`<add name="Logging Handler" type="Microsoft.Practices.EnterpriseLibrary.ExceptionHandling.Logging.LoggingExceptionHandler, Microsoft.Practices.EnterpriseLibrary.ExceptionHandling.Logging" logCategory="FormattedEventCategory" eventId="…" severity="…" title="…" formatterType="…" priority="…" />`
		`</exceptionHandlers>`
		`</add>`
		`</exceptionTypes>`
		`</add>`
		`</exceptionPolicies>`
		`</exceptionHandling>`
		The following is the test.
		`[TestMethod]`
		`public void LoggingNoneActionEventSinkTest()`
		`{`
		`Exception originalException = new System.Security.SecurityException("Security Exception logged in event sink");`
		`bool rethrow = ExceptionPolicy.HandleException(originalException, "Logging To Event Sink PostHandlingAction None");`
		`using (EventLog eventlog = new EventLog("Application"))`
		`{`
		`int entryCount = eventlog.Entries.Count;`
		`Assert.AreEqual("ExceptionHandlingLogging", eventlog.Entries[eventlog.Entries.Count - 1].Source);`
		`Assert.AreEqual(Convert.ToInt64(100), eventlog.Entries[eventlog.Entries.Count - 1].InstanceId);`
		`}`
		`}`

continued

Test case	Result	Automated test

Verify that **ExceptionHandler** objects can be created directly, without using a configuration source.	Passed	The following example verifies that a **WrapHandler** object can be created directly, without using a configuration source. There are similar tests for **ReplaceHandler** objects and **LoggingExceptionHandler** objects.
		`[TestMethod]`
		`public void WrapExceptionCreateDirect()`
		`{`
		`Exception originalException = new System.Arithmetic Exception("Arithmetic Exception is not handled");`
		`WrapHandler handler = new WrapHandler("Wrapped Exception: Arithmetic Exception wrapped with Security Exception", Type.GetType("System.Security. SecurityException, mscorlib, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089") );`
		`Exception wrappedException = handler.HandleExceptio n(originalException, new Guid());`
		`Assert.IsTrue(wrappedException is System.Security. SecurityException);`
		`Assert.AreEqual(wrappedException.Mes- sage, "Wrapped Exception: Arithmetic Exception wrapped with Security Exception");`
		`Assert.IsTrue(wrappedException.InnerEx- ception is ArithmeticException);`
		`}`

Verify that custom exception handlers can be added to the application block from information in the configuration source.	Passed	In this example, a custom exception handler implements the **IExceptionHandler** interface. The test verifies that that custom handler class can be added to the application block from information in the configuration source.
		The following is the custom exception hander.
		`CustomExceptionHandler class`
		`[ConfigurationElementType(typeof(CustomHandlerData) )]`
		`public class CustomExceptionHandler : IException- Handler`
		`{`
		`…`
		`NameValueCollection attributes;`
		
		`public CustomExceptionHandler(){}`
		`public CustomExceptionHandler(NameValueCollection attributes){…}`
		
		`public Exception HandleException(Exception exception, Guid correlationID)`
		`{`
		`…`
		`return exception;`
		`}`
		`}`

Test case	Result	Automated test

		The following is the information in the configuration file.
		<add name="Custom Handler">
		<exceptionTypes>
		<add name="SecurityException" type="System.Security.SecurityException, …" postHandlingAction="ThrowNewException">
		<exceptionHandlers>
		<add name="Custom Handler Throw" type="….CustomExceptionHandler,…" />
		</exceptionHandlers>
		</add>
		</exceptionTypes>
		</add>
		The following is the test case.
		[TestMethod]
		public void CustomExceptionHandlerTest()
		{
		string expectedException = "";
		try
		{
		ExceptionPolicy.HandleException(new SecurityException("To test custom handler"), "Custom Handler");
		}
		catch(Exception ex)
		{
		expectedException = ex.Message;
		}
		Assert.AreEqual(expectedException, "To test custom handler");
		}

continued

Test case	Result	Automated test

Verify that the application block can be configured with a dictionary configuration source.	Passed	This test verifies that the application block can be configured from information in a dictionary configuration source. In this test, the replace handler replaces the original exception message with the one that is configured in the dictionary source. There are similar tests for other handlers.
		`public class DictionarySourceSection`
		`{`
		`public DictionaryConfigurationSource BuildDictionarySourceSection()`
		`{`
		`DictionaryConfigurationSource section = new DictionaryConfigurationSource();`
		`ExceptionHandlingSettings setting = new ExceptionHandlingSettings();`
		
		`ExceptionPolicyData policyData = new ExceptionPolicyData("Replace Handler Test");`
		`ExceptionTypeData typeData = new ExceptionTypeData ("SecurityException",Type.GetType("System.Security.SecurityException, …),PostHandlingAction.ThrowNewException);`
		`ReplaceHandlerData replaceHandlerData = new ReplaceHandlerData("Replace Handler", "Testing Dictionary Source", Type.GetType(…));`
		`typeData.ExceptionHandlers.Add(replaceHandlerData);`
		`policyData.ExceptionTypes.Add(typeData);`
		`setting.ExceptionPolicies.Add(policyData);`
		`section.Add("exceptionHandling", setting);`
		`…`
		`return section;`
		`}`
		`}`

Test case	Result	Automated test

		To execute the test method the source has to be created. The **Initialize** method creates the source and passes it to the factory.
		`private DictionaryConfigurationSource configSource;`
		`private ExceptionPolicyFactory factory;`
		
		`[TestInitialize()]`
		`public void Initialize()`
		`{`
		`DictionarySourceSection sourceSection = new Dictionary-SourceSection();`
		`configSource = sourceSection.BuildDictionarySourceSection();`
		`factory = new ExceptionPolicyFactory(configSource);`
		`}`
		The following is the test.
		`[TestMethod]`
		`public void ReplaceHandlerDicSourceTest()`
		`{`
		`string expectedMessage = "";`
		`try`
		`{`
		`ExceptionPolicyImpl policy = factory.Create("ReplaceHandler Test");`
		`policy.HandleException(new System.Security.SecurityException("Original Exception"));`
		`}`
		`catch (Exception ex)`
		`{`
		`expectedMessage = ex.Message;`
		`}`
		`Assert.AreEqual (expectedMessage, "Testing Dictionary Source");`
		`}`

Testing the Logging Application Block

This chapter explains how functional testing techniques were used to test the Logging Application Block. If you have modified or extended the Logging Application Block, you can use the same techniques and adapt the chapter's templates and checklists to test your own work.

Requirements for the Logging Application Block

The Logging Application Block has the following requirements:

- The application block should be extensible.
- The application block should support common logging operations.
- The application block should be able to distribute logging information to multiple sources.
- The application block should support tracing to mark the start and end of an activity.
- The trace listeners, filters, and formatters should be configurable.
- The application block should be able to read configuration information from any configuration source, such as an XML file or a database.
- The application block should support configurable instrumentation, including WMI (Windows Management Instrumentation), performance counters, and event logs.
- The application block should work with desktop applications and with Web applications.

Selecting the Test Cases

The first step in a functional review is to make sure that the design and the code support these requirements. You do this by deciding the test cases that the design and code must satisfy. Table 1 lists the test cases that the application block's design must satisfy.

Table 1: Logging Application Block Design Test Cases

Priority	Design test case
High	Verify that the application block can be extended with custom trace listeners, custom formatters, and custom filters.
High	Verify that the application block uses a simple façade to log messages.
High	Verify that there is a consistent approach to creating filters, formatters, and trace listeners.
High	Verify that the application block can retrieve configuration data from different sources, such as an application configuration file, a database, or from memory.
High	Verify that the design addresses situations that can cause exceptions and that the application block logs the exceptions through the instrumentation.
High	Verify that the instrumentation is implemented with loosely coupled events.
High	Verify that the application block can use .NET trace listeners.
High	Verify that the application block supports custom property collections for the custom trace listeners, custom formatters and custom filters.

After you identify the design issues, you should do the same for the code. Table 2 lists the test cases that the Logging Application Block code must satisfy.

Table 2: Logging Application Block Code Test Cases

Priority	Code test case
High	Verify that the **Logger** façade exposes all public members as static and that it uses overloads to support different ways to log information.
High	Verify that the application block validates the input at all of its entry points.
High	Verify that any errors that occur during a logging operation are logged to the errors special trace source, if it is in the configuration source.
High	Verify that the application block can distribute information to multiple trace sources.
High	Verify that the application block responds to run-time configuration changes.
High	Verify that the application block disposes of the old **LogWriteStructureHolder** object and creates a new one, if the configuration information changes at run time.
High	Verify that the **Logger** façade creates the **LogWriter** object only once and that it uses the same object for all logging requests that come from the same application.
High	Verify that all logged messages are also logged to the all events special trace source, if it is in the configuration source.
High	Verify that the assembler classes that implements the **IAssembler** interface inject the configuration values into the domain objects.
High	Verify that the application block can trace an activity from start to finish.
High	Verify that the application block uses performance counters to monitor the logging operations, if the performance counters are enabled.
High	Verify that the application block uses WMI and the event log to monitor and record errors that occur during logging operations, if WMI and the event log are enabled.
High	Verify that the application block exposes a dictionary collection property so that the application block can log customized logging information.

Priority	Code test case
High	Verify that the logging application block can log the same message to multiple categories.
High	Verify that the application block logs messages to the trace listeners in a thread safe manner.
High	Verify that the performance counters and the event log that are required by the application block are installed during installation.
Medium	Verify that the application block requests or demands the appropriate code access security permissions to access protected resources and operations.
High	Verify that the application block follows exception management best practices.
High	Verify that the application block follows security best practices.
Medium	Verify that the application block follows globalization best practices.
High	Verify that the application block follows performance best practices.

Verifying the Test Cases

After you identify all the design test cases, you can verify that the design satisfies them. Table 3 lists how each of the design test cases were verified for the Logging Application Block.

Table 3: Logging Application Block Design Verification

Design test case	Implemented?	Feature that implements design
Verify that the application block can be extended with custom trace listeners, custom formatters, and custom filters.	Yes	The **CustomTraceListener** class and the **TraceListener** class allow users to create custom trace listeners. The **ILogFormatter** interface and the **LogFormatter** class allow users to create custom formatters. The **ILogFilter** interface and the **LogFilter** class allow users to create custom filters.
Verify that the application block uses a simple façade to log messages.	Yes	The **Logger** class is a façade that provides simple methods to log messages.
Verify that there is a consistent approach to creating trace listeners, formatters, and filters.	Yes	The **TraceListenerCustomFactory** class creates trace listener objects. The **LogFormatterCustomFactory** class creates formatter objects. The **LogFilterCustomFactory** class creates filter objects.
Verify that the application block can retrieve configuration data from different sources, such as an application configuration file, a database, or from memory.	Yes	The **LogWriterFactory** class has a constructor that accepts a configuration source as an input parameter.

continued

Design test case	Implemented?	Feature that implements design
Verify that the design addresses situations that can cause exceptions and that the application block logs the exceptions through the instrumentation.	Yes	For example, the **LogWriter** class includes the **InstrumentationProvider** property. This property retrieves the instrumentation provider that defines the events for the application block. It logs the events to WMI and to the event log.
Verify that the instrumentation is implemented with loosely coupled events.	Yes	The methods in the **LoggingInstrumentationProvider** class that raise the events bind to the methods in the **LoggingInstrumentationListener** class at run time.
Verify that the application block can use .NET trace listeners.	Yes	The **SystemDiagnosticsTraceListenerData** class configures .NET trace listeners so that the application block can use them.
Verify that the application block supports custom property collections for the custom trace listeners, custom formatters, and custom filters.	Yes	The **CustomTraceListenerData** class supports custom property collections for custom trace listeners. Similarly, the **CustomFilterData** class and the **CustomFormatterData** class support custom property collections for custom filters and custom formatters.

After the code is implemented, you can review it to see if it satisfies its test cases. Table 4 lists the results of a code review for the Logging Application Block.

Table 4: Logging Application Block Code Verification

Code test case	Implemented?	Feature that is implemented

Verify that the **Logger** façade exposes all public members as static and that it uses overloads to support different ways to log information.	Yes	The **Logger.Write** method has multiple overloads to log messages that have different properties. Some of the overloads are shown in the following code examples.
		`// writes a message to the category specified`
		`// in the log object`
		`public static void Write(LogEntry log){}`
		`// writes a log entry to the default category`
		`public static void Write(object message){}`
		`// writes a log message to a specific category`
		`public static void Write(object message, string category){}`
		`// writes a log message to a specific category`
		`// and assigns a priority to the message`
		`public static void Write(object message, string category, int priority){}`

Code test case	Implemented?	Feature that is implemented

Verify that the application block validates the input at all of its entry points.	Yes	For example, the **Tracer** class that traces the log methods first validates the **LogWriter** object before it starts to trace a method. An invalid input causes an exception. This is shown in the following code example.
		`public Tracer(…, LogWriter writer, …)`
		`{`
		`if (writer == null) throw new ArgumentNullException("writer", …);`
		`}`

Verify that the errors that occur during a logging operation are logged to the errors special trace source, if it is in the configuration source.	Yes	For example, in the following code, if the **missingCategories** category is not in the configuration source, the application block logs an error to the errors special trace source that is in the configuration source.
		`private void ReportMissingCategories(List<string> missingCategories, LogEntry logEntry)`
		`{`
		`…`
		`structureHolder.ErrorsTraceSource.TraceData(…);`
		`…`
		`}`

Verify that the application block can log information to multiple trace sources.	Yes	The **Logger.Write** method calls the **LogWriter.ProcessLog** private method. This method logs the information to multiple trace sources. This is shown in the following code example.
		`private void ProcessLog(LogEntry log)`
		`{`
		`…`
		`foreach (LogSource traceSource in matchingTraceSources)`
		`{`
		`…`
		`traceSource.TraceData(…);`
		`…`
		`}`
		`}`

continued

Code test case	Implemented?	Feature that is implemented

Verify that the application block responds to run-time configuration changes.	Yes	The **LogWriterStructureHolderUpdater** internal class registers a handler with the configuration source that will be notified if its data changes. This is shown in the following code example.
		`configurationSource.AddSectionChangeHandler(LoggingSettings.SectionName, UpdateLogWriter);`
		If the configuration source changes at run time, the configuration source notifies the **UpdateLogWriter** method. This method replaces the old object with a new **LogWriterStructureHolder** object. This is shown in the following code example.
		`public void UpdateLogWriter(…)`
		`{`
		`…`
		`logWriter.ReplaceStructureHolder(newStructureHolder);`
		`}`

Verify that the application block disposes of the old **LogWriteStructureHolder** object and creates a new one when the configuration information changes during run time.	Yes	The **UpdateLogWriter** method calls the **LogWriter**.**ReplaceStructureHolder** method when the configuration data changes. This method creates a new **LogWriterStructureHolder** object and disposes of the old one. This is shown in the following code example.
		`internal void ReplaceStructureHolder(LogWriterStructureHolder newStructureHolder)`
		`{`
		`LogWriterStructureHolder oldStructureHolder = structureHolder;`
		
		`structureHolder = newStructureHolder;`
		
		`oldStructureHolder.Dispose();`
		`}`

Verify that the **Logger** façade creates the **LogWriter** object only once and that it uses the same object for all logging requests that come from the same application.	Yes	The **Logger** class only creates a **Writer** object if the object is currently null and is static. This is shown in the following code example.
		`public static LogWriter Writer`
		`{`
		`get`
		`{`
		`if (writer == null)`
		`{`
		`…`
		`writer = factory.Create();`
		`}`
		`}`
		`}`

Code test case	Implemented?	Feature that is implemented

Verify that all logged messages are also logged to the all events special trace source, if it is in the configuration source.	Yes	The **LogWriter.DoGetMatchingTraceSources** method adds the **AllEventsTraceSource** object to to the matching trace sources for every request. This is shown in the following code example.
		`private IEnumerable<LogSource> DoGetMatchingTraceSources(LogEntry logEntry)`
		`{`
		`…`
		`matchingTraceSources.Add(structureHolder.AllEventsTraceSource);`
		`…`
		`}`

Verify that an assembler class that implements the **IAssembler** interface injects the configuration values into the domain objects.	Yes	For example, the **PriorityFilterAssembler** class creates the **PriorityFilter** domain objects. The **TextFormatterAssembler** class creates the **TextFormatter** domain objects and the **FormattedEventLogListenerAssembler** class creates the **FormattedEventLogListener** domain objects. All these classes implement the **IAssembler** interface.

Verify that the application block can trace an activity from start to finish.	Yes	The **Tracer** class traces an activity from start to finish. The trace is initialized when the application block creates a **Tracer** object in the **Tracer** class constructor. Tracing stops when the **Dispose** method of the **Tracer** object is invoked. This is shown in the following code example.
		`public Tracer(string operation)`
		`{`
		`…`
		`Initialize(operation,…);`
		`}`
		
		`protected virtual void Dispose(bool disposing)`
		`{`
		`…`
		`if (IsTracingEnabled()) WriteTraceEndMessage(endTitle);`
		`…`
		`}`

continued

Code test case	Implemented?	Feature that is implemented

Verify that the application block uses performance counters to monitor the logging operations, if the performance counters are enabled.	Yes	For example, the **LogSource** class increments the **Trace Listener Entries Written/sec** performance counter when it writes to a trace listener. This is shown in the following code example.
		`public void TraceData(TraceEventType eventType, int id, LogEntry logEntry, TraceListenerFilter traceListenerFilter)`
		`{`
		`…`
		`listener.TraceData(…);`
		`instrumentationProvider.FireTraceListenerEntry-WrittenEvent();`
		`…`
		`}`
		The **LoggingInstrumentationProvider** class raises the **traceListenerEntryWritten** event. The **TraceListenerEntryWritten** method consumes this event and increments the performance counters. This is shown in the following code example.
		`[InstrumentationConsumer("TraceListenerEntryWritten")]`
		`public void TraceListenerEntryWritten(…)`
		`{`
		`if (PerformanceCountersEnabled) traceListenerEntryWritten.Increment();`
		`}`

Code test case	Implemented?	Feature that is implemented

Verify that the application block uses WMI and the event log to monitor and record errors that occur during logging operations, if WMI and the event log are enabled.	Yes	If an error occurs while the application block logs an error message, the application block uses the instrumentation provider to log exceptions to WMI and to the event log. In addition, the **LogWriter** class logs an exception to the errors special trace source, if it is included in the configuration file. This is shown in the following code example.
		`private void ReportUnknownException(…)`
		`{`
		`try`
		`{`
		`…`
		`structureHolder.ErrorsTraceSource.TraceData(…);`
		`}`
		`catch (Exception ex)`
		`{`
		`instrumentationProvider.FireFailureLoggingEr-`
		`rorEvent(…);`
		`}`
		`}`
		If the error cannot be logged to the errors special trace source, the **LoggingInstrumentationProvider** class raises the **failureLoggingError** event. The **FailureLoggingError** method consumes the event and logs the exception to WMI and to the event log. This is shown in the following code example.
		`[InstrumentationConsumer("FailureLoggingError"`
		`)]`
		`public void FailureLoggingError(…)`
		`{`
		`if (WmiEnabled) ManagementInstrumentation.`
		`Fire(new LoggingFailureLoggingErrorEvent(…);`
		`if (EventLoggingEnabled)`
		`{`
		`EventLog.WriteEntry(GetEventSourceName(), en-`
		`tryText, EventLogEntryType.Error);`
		`}`
		`}`

continued

Code test case	Implemented?	Feature that is implemented

Verify that the application block exposes a dictionary collection property so that the application block can log customized logging information.	Yes	The **LogEntry.ExtendedProperties** property is a dictionary collection that supports customized logging information. This is shown in the following code example.
		`private IDictionary<string, object> extendedProperties;`
		
		`public IDictionary<string, object> ExtendedProperties`
		`{`
		`get`
		`{`
		`if (extendedProperties == null)`
		`{`
		`extendedProperties = new Dictionary<string, object>();`
		`}`
		`return this.extendedProperties;`
		`}`
		`set { this.extendedProperties = value; }`
		`}`

Verify that the logging application block can log the same message to multiple categories.	Yes	The **LogEntry** class has a categories collection that allows the application block to log a message to multiple categories. This is shown in the following code example. `private ICollection<string> categories`

Verify that the application block logs messages to the trace listeners in a thread safe manner.	Yes	The **LogSource** class verifies that the trace listeners are thread safe before it logs the information. This is shown in the following code example.
		`public void TraceData(…)`
		`{`
		`try`
		`{`
		`if (!listener.IsThreadSafe)`
		`Monitor.Enter(listener);`
		`listener.TraceData(…);`
		`}`
		`finally`
		`{`
		`if (!listener.IsThreadSafe)`
		`Monitor.Exit(listener);`
		`}`
		`}`

Verify that the performance counters and the event log that are required by the application block are installed during installation.	Yes	For example, the **LoggingInstrumentationListener** class contains the installer attribute type **[HasInstallableResourcesAttribute]**. The **EventLogInstallerBuilder** and **PerformanceCounterInstallerBuilder** installer classes, which are part of the Enterprise Library Core, recognize this attribute and install the performance counters and event logs.

To learn how the test teams tested the application blocks to see if they conformed to security best practices, see *Testing for Security Best Practices*. To learn how the test teams tested the application blocks to see if they conformed to globalization best practices, see *Testing for Globalization Best Practices*. To learn how the test teams tested the application blocks to see if they met the performance and scalability requirements, see *Testing for Performance and Scalability*.

Using Automated Tests

Automated tests ensure that the application block functions in accordance with its requirements. Automated tests make regression testing easier and certain tests, such as simulating a large number of uses to test a multithreading scenario, require automation.

Table 5: Visual Studio Team System Tests for the Logging Application Block

Test cases	Result	Automated test

Verify that the application block throws the appropriate exception when it receives invalid data.	Passed	The following test uses an invalid log entry.
		`[TestMethod]`
		`[ExpectedException(typeof(ArgumentNullException)`
		`)]`
		`public void LogNullValueLogEntryMessage()`
		`{`
		`Logger.Write(new LogEntry(null, "", -1, -1,`
		`System.Diagnostics.TraceEventType.Critical, "",`
		`null));`
		`}`

continued

Test cases	Result	Automated test

Verify that, if the input data is valid, the application block can log messages to the **MsmqTraceListener** trace listener.	Passed	The following test uses Message Queuing and verifies that the message is logged to the queue. Similar tests exist to verify that all the other trace listeners are supported by the application block.
		`[TestMethod]`
		`public void LogToMsmq()`
		`{`
		`using (MessageQueue queue = new MessageQueue(@".\Private$\EntlibTest"))`
		`{`
		`queue.Purge();`
		`queue.Formatter = new XmlMessageFormatter();`
		`string[] types = { "System.String" };`
		`((XmlMessageFormatter)queue.Formatter).TargetTypeNames = types;`
		`Logger.Write(new LogEntry("To test Msmq listener using Logger facade","MsmqCategory",1,1,TraceEventType.Critical,"",null));`
		`Message message = queue.Receive(new TimeSpan(0,0,5));`
		`Assert.IsNotNull(message);`
		`string content = (string) message.Body;`
		`Assert.IsTrue(content.Contains("To test Msmq listener using Logger facade"));`
		`}`
		`}`

Verify that trace listeners can be created directly, without using a configuration source.	Passed	The following test uses a constructor to create a database listener and a custom object that is logged to the database.
		`[TestMethod]`
		`public void LogToDbListenerUsingCustomObject()`
		`{`
		`FormattedDatabaseTraceListener listener = new FormattedDatabaseTraceListener(DatabaseFactory.CreateDatabase("SqlServer"), "WriteLog", "AddCategory", new TextFormatter());`
		`listener.TraceData(null, "Error : ", TraceEventType.Critical, 1, new MyObject());`
		
		`Database db = DatabaseFactory.CreateDatabase("SqlServer");`
		
		`DbCommand command = db.GetSqlStringCommand("SELECT TOP 1 Message FROM Log ORDER BY LogId DESC");`
		`string messageContents = Convert.ToString(db.ExecuteScalar(command));`
		`Assert.AreEqual(messageContents, "CustomClasses.MyObject");`
		`}`

Test cases	Result	Automated test
Verify that custom classes can be added to the application block and configured from information in the configuration source.	Passed	In this example, the **CustomRegistryListener** custom trace listener derives from the **CustomTraceListener** class. The Enterprise Library Core uses information in the configuration source to add the custom class to the application block. The test verifies that the custom trace listener logs the registry data to the registry path that is specified in the configuration file. The following code defines the **CustomRegistryListener** class. `CustomRegistryListener class` `[ConfigurationElementType(typeof(CustomTraceListenerData))]` `public class CustomRegistryListener: CustomTraceListener` `{` `public CustomRegistryListener():base(){}` `public CustomRegistryListener(string initializeData): base(){}` `public override void TraceData(…)` `{` `…write it to registry` `}` `public override void Write(){…}` `public override void WriteLine(string message){…}` The following is the information in the configuration file. `<listeners>` `<add regKey=" SOFTWARE\EntlibTestApp" regName="LogEntry" listenerDataType="Microsoft.Practices.EnterpriseLibrary.Logging.Configuration.CustomTraceListenerData, Microsoft.Practices.EnterpriseLibrary.Logging, Version=2.0.0.0, Culture=neutral, PublicKeyToken=null" traceOutputOptions="***" type="***.CustomRegistryListener, ***" name="***" initializeData="" formatter="***" />` The following is the test. `[TestMethod]` `public void LogToRegistryListener()` `{` `LogEntry log = new LogEntry();` `log.Message = "To test CustomTrace Listener that derives from CustomTraceListener";` `log.Title= "LogToRegistryListener";` `log.Categories.Add("RegistryCategory");` `Logger.Write(log);` `RegistryKey key = Registry.LocalMachine.OpenSubKey(@"Software\EntlibTestApp");` `string content = (string) key.GetValue("LogEntry");` `key.Close();` `Assert.IsTrue(content.Contains(log.Message));` `}`

continued

Test cases	Result	Automated test

Verify that a provider that is configured with data in a dictionary configuration source can log messages.	Passed	The following example uses a dictionary configuration source to log messages to a database.
		`[TestMethod]`
		`public void LogToDb()`
		`{`
		`LogEntry log = new LogEntry("", "DatabaseCatego-`
		`ry", 4, 1000, TraceEventType.Critical, "", null);`
		`log.Title = "Enterprise Library";`
		`log.Message = "Test";`
		`LogWriterFactory factory = new LogWriterFactory(`
		`source);`
		`using (LogWriter writer = factory.Create())`
		`{`
		`  writer.Write(log);`
		`}`
		
		`DatabaseProviderFactory Dbfactory = new DatabaseP`
		`roviderFactory(source);`
		`Database db = Dbfactory.Create("SqlServer");`
		`DbCommand command = db.GetSqlStringCommand("SELEC`
		`T TOP 1 Message FROM Log ORDER BY LogId DESC");`
		`string messageContents = Convert.ToString(db.`
		`ExecuteScalar(command));`
		`Assert.AreEqual(messageContents, log.`
		`Message);`
		`}`

Test cases	Result	Automated test

Verify that the trace listeners are thread safe.	Passed	This test verifies that the **FormattedEventLogListener** method is thread safe.
		`[TestMethod]`
		`public void FormattedEventLogListenerTest()`
		`{`
		`using (EventLog eventlog = new EventLog(HelperClass.GetEventLogName()))`
		`{`
		`int initialCount = eventlog.Entries.Count;`
		`category = "General";`
		`InitiateThread();`
		`int finalCount = eventlog.Entries.Count;`
		`Assert.IsTrue((finalCount - initialCount) == threadCount, "Count : " + (finalCount - initialCount));`
		`}`
		`}`
		The test uses the **InitiateThread** and **LogToEventLog** helper methods.
		`private void InitiateThread()`
		`{`
		`Thread[] th = new Thread[threadCount];`
		`for (int i = 0; i < threadCount; i++)`
		`{`
		`th[i] = new Thread(new ThreadStart(LogToEventLog));`
		`th[i].Start();`
		`}`
		`for (int i = 0; i < threadCount; i++)`
		`{`
		`th[i].Join();`
		`}`
		`}`
		
		`private void LogToEventLog()`
		`{`
		`LogEntry entry = new LogEntry();`
		`entry.Message = "Test: Thread Test";`
		`entry.Categories.Add(category);`
		`Logger.Write(entry);`
		`}`

Testing the Security Application Block

This chapter explains how functional testing techniques were used to test the Security Application Block. If you have modified or extended the Security Application Block, you can use the same techniques and adapt the chapter's templates and checklists to test your own work.

Requirements for the Security Application Block

The Security Application Block has the following requirements:

- The application block should be extensible.
- The application block should provide a simple interface for authorization operations.
- The application block should provide a simple interface for saving, expiring, and retrieving the identity from a caching store.
- The application block should provide a standard provider model for authorization and security-related caching.
- Authorization providers and security cache providers should be configurable.
- The application block should be able to read configuration information from any configuration source, such as an XML file or a database.
- The application block should support configurable instrumentation, including WMI (Windows Management Instrumentation), performance counters, and event logs.
- The application block should work with desktop applications and with Web applications.

These requirements must be incorporated into the design and implemented by the code.

Selecting the Test Cases

The first step in a functional review is to make sure that the design and the code support these requirements. You do this by deciding the test cases that they must satisfy so that the application block fulfills all its requirements.

Table 1 lists the test cases that the application block's design must satisfy.

Table 1: Security Application Block Design Cases

Priority	Design test case
High	Verify that the application block can be extended with custom authorization providers and custom security cache providers.
High	Verify that there is a consistent approach to creating authorization providers and security cache providers.
High	Verify that the application block provides a simple method to perform authorization.
High	Verify that the application block provides simple methods to save, expire, and retrieve an identity from a caching store.
High	Verify that the application block implements the standard provider model for authorization and security-related operations.
High	Verify that the ability to create the application block's domain objects from the configuration data follows the Dependency Injection pattern.
High	Verify that the application block can retrieve configuration data from different sources, such as an application configuration file, a database, or from memory.
High	Verify that the instrumentation is implemented with loosely coupled events.
High	Verify that situations that can cause exceptions are addressed and that the application block logs the exceptions through the instrumentation.
High	Verify that the application block supports custom property collections for the custom authorization providers and custom caching stores.

After you identify the design issues, you should do the same for the code. Table 2 lists the test cases that the Security Application Block code must satisfy.

Table 2: Security Application Block Code Test Cases

Priority	Code test case
High	Verify that the application block creates one instance of the **AuthorizationProvider** class for each request.
High	Verify that the application block only creates one instance of the **SecurityCacheProvider** class for the specified instance name.
High	Verify that the assembler classes that implements the **IAssembler** interface create the authorization providers and the security cache providers, and verify that the assembler classes inject the configuration object values into those domain objects.
High	Verify that the configuration properties of the authorization and security cache providers are exposed as public and that they are strongly typed.
High	Verify that the configuration properties for the custom providers are exposed as public and that they are implemented as custom property collections.

Priority	Code test case
High	Verify that the application block can use default instance names that are defined in the configuration source to create the authorization providers and the security cache providers. Verify that the application block can also use specific instance names to create these providers.
High	Verify that the security cache provider encrypts the data stored in the cache.
High	Verify that the application block uses performance counters and WMI to monitor authorization and caching operations, if the instrumentation is enabled.
High	Verify that the performance counters and the event log that are required by the application block are installed during installation.
Medium	Verify that the application block requests or demands access to protected system resources and operations.
High	Verify that the application block follows exception management best practices.
High	Verify that the application block follows security best practices.
Medium	Verify that the application block follows globalization best practices.
High	Verify that the application block follows performance best practices.

Verifying the Test Cases

After you identify all the design test cases, you can verify that the design satisfies them. Table 3 lists how each of the design test cases were verified for the Security Application Block.

Table 3: Security Access Application Block Design Verification

Design test case	Implemented?	Feature that implements design
Verify that the application block can be extended with custom authorization providers and security cache providers.	Yes	The **IAuthorizationProvider** interface and the **AuthorizationProvider** class allow users to implement or extend a configurable authorization provider. The **ISecurityCacheProvider** interface and the **SecurityCacheProvider** class allow users to implement or extend a security cache provider.
Verify that there is a consistent approach to creating authorization providers and security cache providers.	Yes	The **AuthorizationProviderFactory** class creates the **AuthorizationProvider** objects. The **SecurityCacheProviderFactory** class creates the **SecurityCacheProvider** objects.
Verify that the application block provides a simple method to perform authorization.	Yes	Both the **IAuthorizationProvider** interface and the **AuthorizationProvider** class expose simple methods named **Authorize** that perform authorization. The **AuthorizationRuleProvider** class that derives from the **AuthorizationProvider** class implements the **Authorize** method.

continued

Design test case	Implemented?	Feature that implements design
Verify that the application block provides simple methods to save, expire, and retrieve an identity from a caching store.	Yes	The **CachingStoreProvider** class that derives from the **SecurityCacheProvider** exposes the **SaveIdentity** method to save an identity, the **ExpireIdentity** method to expire an identity, and the **GetIdentity** method to retrieve an identity. Similarly, it also it exposes methods to retrieve, save, and expire principals and profiles.
Verify that the application block implements the standard provider model for authorization and security related-caching and that the provider is configurable.	Yes	The provider model that is implemented in the application block is similar to the standard provider model implemented by ASP.NET. The ASP.NET provider model defines a contract in the **ProfileProvider** class that all providers, such as **SqlProfileProvider**, implement. Any custom provider that derives from the **ProfileProvider** class can be added to the configuration file and used without ASP.NET knowing any of the implementation details. Similarly, the Security Application Block defines a contract in the **IAuthorizationProvider** and **ISecurityCacheProvider** interfaces. These interfaces are implemented by providers such as **AuthorizationRuleProvider**, **AzmanAuthorization Provider**, and **CachingStoreProvider**. Also, custom providers that derive from the **IAuthorizationProvider** and the **ISecurityCacheProvider** interfaces can be added to the application block and used without the application block knowing any of the implementation details.
Verify that the ability to create the application block's domain objects from the configuration data follows the dependency injection pattern.	Yes	The **SecurityCacheProviderFactory** class derives from the **LocatorNameTypeFactoryBase** generic type, which takes the configuration source as input, creates the domain object and injects the relevant configuration data into the domain object. Similarly, the **AuhthorizationProviderFactory** class derives from the **NameTypeFactoryBase**, which follows the same pattern.
Verify that the application block can retrieve configuration data from different sources such as an application configuration file, a database, or from memory.	Yes	The **SecurityCacheProviderFactory** class and the **AuhthorizationProviderFactory** class have constructors that accept configuration sources as input parameters.

Design test case	Implemented?	Feature that implements design
Verify that the instrumentation uses loosely coupled events.	Yes	The methods in the **SecurityCacheProviderInstrumentationProvider** class and the **AuthorizationProviderInstrumentationProvider** class raise the events that bind to the methods in the **SecurityCacheProviderInstrumentationListener** class and the **AuthorizationProviderInstrumentationListener** class respectively during run time.
Verify that situations that can cause exceptions are addressed and that the application block logs the exceptions through the instrumentation.	Yes	For example, both the **CachingStoreProvider** class and the **AuthorizationRuleProvider** class include the **InstrumentationProvider** property. This property retrieves the instrumentation provider that defines the events for the security cache and the authorization provider. The provider logs the events to WMI and the event log.
Verify that the application block supports custom property collections for custom security cache providers and for custom authorization providers.	Yes	The **CustomAuthorizationProviderData** class and the **CustomSecurityCacheProviderData** class support custom property collections for custom providers.

After the code is implemented, you can review it to see if it satisfies its test cases. Table 4 lists the results of a code review for the Security Application Block.

Table 4: Security Application Block Code Verification

Code test case	Implemented?	Feature that is implemented
Verify that, for each request, the application block only creates one instance of a class that derives from the **AuthorizationProvider** or that implements the **IAuthorizationProvider** interface.	Yes	The **AuthorizationFactory** class creates instances of classes that derive from the **AuthorizationProvider** class as well as instances of classes that implement the **IAuthorizationProvider** interface. It does this by calling the **AuthorizationProviderFactory** class. The **AuthorizationProviderFactory** class derives from the **NameTypeFactoryBase** class. This class creates a new instance of a provider for every request. This derivation is shown in the following code. `public class AuthorizationProviderFactory : NameTypeFactoryBase<IAuthorizationProvider>{}`

continued

Code test case	Implemented?	Feature that is implemented
Verify that, for a specific instance name, the application block only creates one instance of a class that derives from the **SecurityCacheProvider** class or that implements the **ISecurityCacheProvider** interface.	Yes	The **SecurityCacheFactory** class creates instances of classes that derive from the **SecurityCacheProvider** class as well as instances of classes that implement the **ISecurityCacheProvider** interface. It does this by calling the **SecurityCacheProviderFactory** class. The **SecurityCacheProviderFactory** class derives from the **LocatorNameTypeFactoryBase**. This class creates a single instance of the provider for the given instance name. This derivation is shown in the following code. `public class SecurityCacheProviderFactory : LocatorNameTypeFactoryBase<ISecurityCacheProvider>{}`

Verify that an assembler class that implements the **IAssembler** interface creates the authorization providers and the security cache providers and injects the configuration object values into those domain objects.	Yes	The following code demonstrates how the **AuthorizationRuleProviderAssembler** class, which implements the **IAssembler** interface, creates an **AuthorizationRuleProvider** object. The process is similar for creating **AzmanAuthorizationProvider** objects and **CachingStoreProvider** objects.
		`public class AuthorizationRuleProviderAssembler : IAssembler<IAuthorizationProvider, AuthorizationProviderData>`
		`{`
		`public IAuthorizationProvider Assemble(IBuilderContext context, AuthorizationProviderData objectConfiguration, IConfigurationSource configurationSource, ConfigurationReflectionCache reflectionCache)`
		`{`
		`…`
		`IAuthorizationProvider createdObject= new AuthorizationRuleProvider(authorizationRules);`
		
		`return createdObject;`
		`}`
		`}`

Code test case	Implemented?	Feature that is implemented

Verify that the authorization and security cache providers' configuration properties are exposed as public and that they are strongly typed.	Yes	In the **SecuritySettings** class, the **AuthorizationProviders** property can only contain a collection of type **AuthorizationProviderData**. The following code demonstrates this. The equivalent is true for the **SecurityCacheProviders** property.
		`[ConfigurationProperty(authorizationProvidersProperty, IsRequired= false)]`
		`public NameTypeConfigurationElementCollection<AuthorizationProviderData> AuthorizationProviders`
		`{`
		`get`
		`{`
		`return (NameTypeConfigurationElementCollection<AuthorizationProviderData>)base[authorizationProvidersProperty];`
		`}`
		`}`

Verify that the configuration properties for the custom providers are exposed as public and that they are implemented as custom property collections.	Yes	The **CustomAuthorizationProviderData** class and the **CustomSecuityCacheProviderData** class have references to the **CustomProviderDataHelper** generic class. This class defines a **NameValueCollection** class that holds the attributes as custom property collections. This is shown in the following code example.
		`private NameValueCollection attributes;`
		
		`private void AddAttributesFromConfigurationProperties()`
		`{`
		`foreach (ConfigurationProperty property in propertiesCollection)`
		`{`
		`…`
		`attributes.Add(property.Name, (string)helpedCustomProviderData.BaseGetPropertyValue(property));`
		`}`
		`}`

continued

Code test case	Implemented?	Feature that is implemented

Verify that the application block can use default instance names that are defined in the configuration source to create the authorization providers and the security cache providers. Verify that the application block can also use specific instance names to create these providers.	Yes	The **AuthorizationProviderFactory. CreateDefault** method creates **AuthorizationProvider**-derived objects with the default instance name specified in the configuration source. The **AuthorizationProviderFactory. Create** method creates **AuthorizationProvider**-derived objects with the specified instance name. The **SecurityCacheFactory** class has similar methods to create security cache providers. The following code shows the two **AuthorizationProviderFactory** methods.
		`public static IAuthorizationProvider GetAuthorizationProvider()`
		`{`
		`…`
		`return factory.CreateDefault();`
		`}`
		
		`public static IAuthorizationProvider GetAuthorizationProvider(string authorizationProviderName)`
		`{`
		`…`
		`return factory.Create(authorizationProviderName);`
		`}`

Code test case	Implemented?	Feature that is implemented

Verify that the security cache provider encrypts the data stored in the cache.	Yes	The **CachingStoreProvider** class derives from the **SecurityCacheProvider** class. This class uses a **CacheManager** instance to add, remove, and retrieve an identity from the cache. In the following example, the **CacheManager** instance adds the identity to the caching store. The Caching Application Block's database cache encrypts the data stored in the cache.
		`public class CachingStoreProvider : SecurityCacheProvider`
		`{`
		`private CacheManager securityCacheManager;`
		
		`public override void SaveIdentity(IIdentity identity, IToken token)`
		`{`
		`GetSecurityCacheItem(token, true).Identity = identity;`
		`}`
		
		`private SecurityCacheItem GetSecurityCacheItem(IToken token, bool createIfNull)`
		`{`
		`…`
		`securityCacheManager.Add(token.Value, item, CacheItemPriority.Normal, null, GetCacheExpirations());`
		`…`
		`}`
		`}`
		The following code demonstrates how the **DatabaseBackingStore** object encrypts the data before adding it to the database.
		`public class DatabaseBackingStore : BaseBackingStore`
		`{`
		`protected override void AddNewItem(int storageKey, CacheItem newItem)`
		`{`
		`…`
		`this.encryptionProvider.Encrypt(valueBytes);`
		`…`
		`}`
		`}`
		Similarly, the Caching Application Block's **IsolatedStorageCacheProvider** class also encrypts the data stored in the cache.

continued

Code test case	Implemented?	Feature that is implemented

Verify that the application block uses performance counters and WMI to monitor authorization and caching operations if the instrumentation is enabled.	Yes	In the following example, whenever an **AuthorizationRuleProvider** object performs an authorization, it triggers the **AuthorizationCheckPerformedEvent** event, which notifies WMI and increments the **Authorization Requests/sec** performance counter. This is shown in the following code.
		`public class AuthorizationRuleProvider : AuthorizationProvider`
		`{`
		
		`public override bool Authorize(IPrincipal principal, string ruleName)`
		`{`
		`…`
		`InstrumentationProvider.FireAuthorizationCheckPerformed(principal.Identity.Name, ruleName);`
		`…`
		`}`
		`}`
		The **AuthorizationProviderInstrumentationProvider** class raises the **authorizationCheckPerformed** event. The **SecurityCacheReadPerformed** method consumes this event. It notifies WMI and increments the performance counters. This is shown in the following code example.
		`[InstrumentationConsumer("AuthorizationCheckPerformed")]`
		`public void AuthorizationCheckPerformed(…)`
		`{`
		`if (PerformanceCountersEnabled) authorizationCheckPerformedCounter.Increment();`
		`if (WmiEnabled) ManagementInstrumentation.Fire(new AuthorizationCheckPerformedEvent(…));`
		`}`
		Similarly, whenever the **CachingStoreProvider** object retrieves an identity from the cache, it raises the **SecurityCacheReadPerformedEvent** method. This method notifies WMI and increments the **Security Cache Reads/sec** performance counter.
		`public class CachingStoreProvider : SecurityCacheProvider`
		`{`
		`public override IIdentity GetIdentity(IToken token)`
		`{`
		`…`
		`InstrumentationProvider.FireSecurityCacheReadPerformed(SecurityEntityType.Identity, token);`
		`…`
		`}`
		`}`

Code test case	Implemented?	Feature that is implemented

		The **SecurityCacheInstrumentationProvider** class raises the **securityCacheReadPerformed** event. The **SecurityCacheReadPerformed** method consumes this event, notifies WMI, and increments the performance counters. This is shown in the following code example.
		`[InstrumentationConsumer("SecurityCacheReadPerformed")]`
		`public void SecurityCacheReadPerformed(…)`
		`{`
		`if (PerformanceCountersEnabled) securityCacheReadPerformedCounter.Increment();`
		`if (WmiEnabled) ManagementInstrumentation.Fire(newSecurityCacheReadPerformedEvent(…));`
		`}`

Verify that the performance counters and the event log that are required by the application block are installed during installation.	Yes	For example, the **SecurityCacheProviderInstrumentationListener** class contains the installer attribute type **[HasInstallableResourcesAttribute]**. The **EventLogInstallerBuilder** and the **PerformanceCounterInstallerBuilder** installer classes identify the attribute and install the event logs and performance counters.

To learn how the test teams tested the application blocks to see if they conformed to security best practices, see *Testing for Security Best Practices*. To learn how the test teams tested the application blocks to see if they conformed to globalization best practices, see *Testing for Globalization Best Practices*. To learn how the test teams tested the application blocks to see if they met the performance and scalability requirements, see *Testing for Performance and Scalability*.

Using Automated Tests

Automated tests ensure that the application block functions in accordance with its requirements. Automated tests make regression testing easier and certain tests, such as simulating a large number of users to test a multithreading scenario, require automation.

Table 5 lists the Visual Studio Team System tests that were used with the Security Application Block.

Table 5: Visual Studio Team System Tests for the Security Application Block

Test case	Result	Automated test

Verify that the application block throws the appropriate exception when it receives invalid data.	Passed	The following test supplies an invalid instance name to the **GetAuthorizationProvider** name, which causes the application block to throw the **ConfigurationErrorsException** exception.
		`[TestMethod]`
		`[ExpectedException(typeof(ConfigurationErrorsException))]`
		`public void CreateInvalidAuthRuleProviderFromConfigFile()`
		`{`
		`IAuthorizationProvider authRuleProvider = AuthorizationFactory.GetAuthorizationProvider("InvalidRule");`
		`}`

Test case	Result	Automated test

Verify that if the input data is valid, the application block either creates an **AuthorizationRuleProvider** object with the default instance name or with the specified instance name.	Passed	The following test creates one **AuthorizationRuleProvider** object with a specific instance name and another **AuthorizationRuleProvider** with the default instance name. The following is the configuration data that provides the default instance name.
		`<securityConfiguration defaultAuthorizationInstance="DefaultRuleProvider"`
		`defaultSecurityCacheInstance="">`
		`<authorizationProviders>`
		`<add`
		`name="DefaultRuleProvider"`
		`type="Microsoft.Practices.EnterpriseLibrary.Security.AuthorizationRuleProvider, Microsoft.Practices.EnterpriseLibrary.Security">`
		`<rules>`
		`<add name="TestAnonymousRule expression="I:?"`
		`…`
		`/>`
		`</rules>`
		`</add>`
		`</authorizationProviders>`
		`</securityConfiguration>`
		The following are the tests.
		`// This test is for the specified instance name.`
		`[TestMethod]`
		`public void CreateNamedAuthRuleProviderInConfigFile()`
		`{`
		`IAuthorizationProvider authRuleProvider = AuthorizationFactory.GetAuthorizationProvider("DefaultRuleProvider");`
		`Assert.IsNotNull(authRuleProvider);`
		
		`}`
		
		`// This test is for the default instance name.`
		`[TestMethod]`
		`public void CreateDefaultAuthRuleProviderFromConfigFile()`
		`{`
		`IAuthorizationProvider authRuleProvider = AuthorizationFactory.GetAuthorizationProvider();`
		`Assert.IsNotNull(authRuleProvider);`
		`}`

continued

Test case	Result	Automated test

Verify that the **AuthorizationRuleProvider** object uses the rule in the configuration source to authorize an identity.	Passed	This test applies a rule named **TestAnonymousRule** that is defined in the configuration source. The test verifies that the **AuthorizationRuleProvider** object authorizes an anonymous identity.
		`[TestMethod]`
		`public void AuthorizeAnonymousIdentityTest()`
		`{`
		`AuthorizationRuleProvider authRuleProvider = AuthorizationFactory.GetAuthorizationProvider("DefaultRuleProvider") as AuthorizationRuleProvider;`
		`bool authorized = authRuleProvider.Authorize(new WindowsPrincipal(WindowsIdentity.GetAnonymous()),"TestAnonymousRule");`
		`Assert.IsTrue(authorized);`
		`}`

Verify that the **AzManAuthorizationProvider** object uses the operations specified in the AzMan store to grant permission to an identity to perform a task.	Passed	This test verifies that an anyonymous identity has permission to view a purchase order. The purchase order is defined in the **AzManFuncTestsConfig.xml** AzMan store. The store grants permission to everyone to perform the View Purchase Order operation. The following is the information in the configuration file. `<authorizationProviders>` `<add name="DefaultAzManProvider" type="Microsoft.Practices.EnterpriseLibrary.Security.AzMan.AzManAuthorizationProvider, Microsoft.Practices.EnterpriseLibrary.Security.AzMan" storeLocation="msxml://{currentPath}/AzManFuncTestsConfig.xml" application ="SecurityAzManFuncTests" auditIdentifierPrefix="" scope="" />` `</authorizationProviders>` The following is the test. `[TestMethod]` `public void AzManOperationAuthorizeTest()` `{` `IAuthorizationProvider azManProvider = AuthorizationFactory.GetAuthorizationProvider("DefaultAzManProvider");` `WindowsIdentity identity = WindowsIdentity.GetAnonymous();` `bool isAuthorized = azManProvider.Authorize(new WindowsPrincipal(identity), "O:View Purchase Order");` `Assert.IsTrue(isAuthorized);` `}`

Test case	Result	Automated test

Verify that, if the input data is valid, the application block either creates a **CachingStoreProvider** object with the default instance name or with the specified instance name.	Passed	The following test creates one **CachingStoreProvider** object with a specific instance name and another **CachingStoreProvider** object with the default instance name. The following is the configuration data that provides the default instance name.
		`<securityCacheProviders>`
		`<add`
		`name="CacheProvider"`
		`type="Microsoft.Practices.EnterpriseLibrary.Security.Cache.CachingStore.CachingStoreProvider, Microsoft.Practices.EnterpriseLibrary.Security.Cache.CachingStore"`
		`cacheManagerInstanceName="DefaultCacheManager"`
		`defaultSlidingSessionExpirationInMinutes="1"`
		`defaultAbsoluteSessionExpirationInMinutes="1"`
		`/>`
		`</securityCacheProviders>`
		
		`//Thisis the test for the specified instance name`
		`public void CreateNamedSecurityCacheInConfigFile()`
		`{`
		`ISecurityCacheProvider cacheProvider = SecurityCacheFactory.GetSecurityCacheProvider("CacheProvider");`
		`Assert.IsNotNull(cacheProvider);`
		`}`
		
		`//This is the test for the default instance name`
		`[TestMethod]`
		`public void CreateDefaultSecurityCacheFromConfigFile()`
		`{`
		`ISecurityCacheProvider cacheProvider = SecurityCacheFactory.GetSecurityCacheProvider();`
		`Assert.IsNotNull(cacheProvider);`
		`}`

continued

Test case	Result	Automated test

Verify that a **CachingStoreProvider** object can add an identity to the cache and retrieve an identity from the cache.	Passed	This test uses information in the configuration store to create a secure cache provider. It then uses the cache provider first to save an identity to the cache and then to retrieve an identity from the cache.
		The following is the configuration information.
		`<securityCacheProviders>`
		`<add`
		`name="CacheProviderDB"`
		`type="Microsoft.Practices.EnterpriseLibrary.Security.Cache.CachingStore.CachingStoreProvider, Microsoft.Practices.EnterpriseLibrary.Security.Cache.CachingStore"`
		…
		`/>`
		`</securityCacheProviders>`
		The following is the test.
		`[TestMethod]`
		`public void DBAddIdentityTest()`
		`{`
		`ISecurityCacheProvider securityCache = SecurityCacheFactory.GetSecurityCacheProvider("CacheProviderDB");`
		`Assert.IsNotNull(securityCache);`
		`IToken token = securityCache.SaveIdentity(identity);`
		`Assert.IsNotNull(token);`
		`Assert.IsNotNull(token.Value);`
		
		`IIdentity cachedIdentity = securityCache.GetIdentity(token);`
		`Assert.IsNotNull(cachedIdentity);`
		`Assert.AreEqual(cachedIdentity.Name, "testuser");`
		`}`

Test case	Result	Automated test

Verify that authorization providers and security cache providers can be created directly, without using a configuration source.	Passed	This test uses a constructor to create a **AuthorizationRuleProvider** object. The test then verifies that the object can perform authorizations. The process is similar for **AzmanAuthorizationProvider** objects and **CachingStoreProvider** object.
		`[TestMethod]`
		`public void CreateAuthRuleProviderDirect()`
		`{`
		`IAuthorizationRule ruleData= new AuthorizationRuleData ("TestRule","I:TestUser");`
		`IDictionary<string,IAuthorizationRule> ruleDict= new Dictionary <string,IAuthorizationRule>( );`
		`ruleDict["TestRule"]= ruleData;`
		`IAuthorizationRule ruleData1 = new AuthorizationRuleData("TestRule1", "I:TestUser1");`
		`ruleDict["TestRule1"] = ruleData1;`
		`IAuthorizationProvider provider = new AuthorizationRuleProvider(ruleDict);`
		`bool isAuthorized = provider.Authorize(new GenericPrincipal(new GenericIdentity("TestUser"), new string[]{}),"TestRule");`
		`Assert.IsTrue(isAuthorized);`
		`}`

continued

Test case	Result	Automated test

Verify that authorization providers and security cache providers can be created from a dictionary or from an in-memory configuration source.	Passed	This test verifies that an **AuthorizationRuleProvider** can be created from a dictionary source. Similar tests were used with an **AzmanAuthorizationProvider** object and a **CachingStoreProvider** object. The following code defines the dictionary.
		`public class AuthorizationRuleDictSource`
		`{`
		`public DictionaryConfigurationSource BuildDictionarySourceSection()`
		`{`
		`DictionaryConfigurationSource section = new DictionaryConfigurationSource();`
		`SecuritySettings secSettings = new SecuritySettings();`
		`secSettings.DefaultAuthorizationProviderName = "DefaultRuleProviderDic";`
		`…`
		`arpd.Type = Type.GetType("Microsoft.Practices.EnterpriseLibrary.Security.AuthorizationRuleProvider, Microsoft.Practices.EnterpriseLibrary.Security");`
		`secSettings.AuthorizationProviders.Add(arpd);`
		
		`section.Add("securityConfiguration", secSettings);`
		
		`return section;`
		`}`
		The following is the test.
		`// This initializes the constructor`
		`[TestInitialize()]`
		`public void MyTestInitialize()`
		`{`
		`AuthorizationRuleDictSource sourceSection = new AuthorizationRuleDictSource();`
		`configSource = sourceSection.BuildDictionarySourceSection();`
		`AuthorizationProviderFactory authFactory = new AuthorizationProviderFactory(configSource);`
		`authRuleProvider = authFactory.Create("DefaultRuleProviderDic") as AuthorizationRuleProvider;`
		
		`identity = new GenericIdentity("TestUser");`
		`string[] roles = new string[] { "Manager" };`
		`principal = new GenericPrincipal(identity, roles);`
		`}`
		
		`[TestMethod]`
		`public void AuthorizeIdentityDicTest()`
		`{`
		`bool authorized = authRuleProvider.Authorize(principal, "TestIdentityRuleDic");`
		`Assert.IsTrue(authorized);`
		`}`

Test case	Result	Automated test

Verify that custom authorization classes that derive from the **AuthorizationProvider** class can be added to the application block and configured from the configuration source.	Passed	In this example, the **CustomAuthorizationProvider** custom authorization class derives from the **AuthorizationProvider** class. The Enterprise Library Core uses information in the dictionary configuration source to add the custom class to the application block. The following is the configuration information.
		`public DictionaryConfigurationSource BuildDictionarySourceSection()`
		`{`
		`DictionaryConfigurationSource section = new DictionaryConfigurationSource();`
		
		`SecuritySettings secSettings = new SecuritySettings();`
		`CustomAuthorizationProviderData customAuthProvider = new CustomAuthorizationProviderData("MyAuthorizationProvider",Type.GetType("….CustomAuthorizationProvider,…"));`
		`customAuthProvider.Attributes.Add("key1", "value1");`
		`secSettings.AuthorizationProviders.Add(customAuthProvider);`
		
		`section.Add("securityConfiguration", secSettings);`
		`}`
		The following is the custom class.
		`[ConfigurationElementType(typeof(CustomAuthorizationProviderData))]`
		`public class CustomAuthorizationProvider:AuthorizationProvider`
		`{`
		`public CustomAuthorizationProvider()`
		`{`
		`}`
		`public CustomAuthorizationProvider(System.Collections.Specialized.NameValueCollection collection )`
		`{`
		`}`
		
		`public override bool Authorize(IPrincipal principal, string context)`
		`{`
		`…`
		`}`
		`}`
		The following is the test.
		`[TestMethod]`
		`public void CustomAuthorizeRoleDicTest()`
		`{`

continued

Test case	Result	Automated test

		AuthorizationProviderFactory factory = new AuthorizationProviderFactory(configSource);
		
		IAuthorizationProvider authRuleProvider = factory.Create("MyAuthorizationProvider");
		bool authorized = authRuleProvider.Authorize(null,"");
		Assert.IsTrue(authorized);
		}

Verify that custom security cache classes that derive from the **SecurityCacheProvider** class can be added to the application block and configured from configuration source.	Passed	In this example, the custom authorization class, **CustomSecurityCacheProvider**, derives from the **SecurityCacheProvider** class. The Enterprise Library Core uses information in the dictionary configuration source to add the custom class to the application block.
		The following is the configuration information.
		public DictionaryConfigurationSource BuildDictionarySourceSection()
		{
		DictionaryConfigurationSource section = new DictionaryConfigurationSource();
		
		SecuritySettings secSettings = new SecuritySettings();
		
		CustomSecurityCacheProviderData customData = new CustomSecurityCacheProviderData("MyCache", Type.GetType("SecurityCachingFuncTests.CustomSecurityCacheProvider,SecurityCachingFuncTests"));
		customData.Attributes.Add("key1", "value1");
		secSettings.SecurityCacheProviders.Add(customData);
		
		section.Add("securityConfiguration", secSettings);
		}
		The following is the custom class.
		[ConfigurationElementType(typeof(CustomSecurityCacheProviderData))]
		class CustomSecurityCacheProvider:SecurityCacheProvider
		{
		public override IToken SaveIdentity(IIdentity identity){…}
		
		public override void SaveIdentity(IIdentity identity, IToken token)
		{…}
		
		public override IToken SavePrincipal(IPrincipal principal)
		{…}

Test case	Result	Automated test

		public override void SavePrincipal(IPrincipal principal, IToken token){…}
		
		public override IToken SaveProfile(object profile){…}
		
		public override void SaveProfile(object profile, IToken token) {…}
		
		public override void ExpireIdentity(IToken token){…}
		
		public override void ExpirePrincipal(IToken token){…}
		
		public override void ExpireProfile(IToken token){…}
		
		public override IIdentity GetIdentity(IToken token){…}
		
		public override IPrincipal GetPrincipal(IToken token){…}
		
		public override object GetProfile(IToken token){…}
		}
		The following is the test.
		[TestMethod]
		public void CreateCustomSecurityCache()
		{
		SecurityCacheProviderFactory factory = new SecurityCacheProviderFactory(new SecurityCachingDictSource().BuildDictionarySourceSection());
		
		ISecurityCacheProvider cacheProvider = factory.Create("MyCache");
		Assert.IsNotNull(cacheProvider);
		}

Testing for Security Best Practices

Testing the Enterprise Library application blocks to see if they conform to security best practices involves several activities:

- **Establish the security requirements.** Security requirements are the goals and constraints that affect the confidentiality, integrity, and availability of the application blocks.
- **Analyze the application block.** To analyze an application block, identify such things as its assets, its entry points, and its dependencies. Create a diagram that shows the relevant subsystems that make up the application block
- **Build the threat models**. Threat models allow you to systematically identify and rate the threats that are most likely to affect your applications. By identifying and rating threats based, you can address threats with appropriate countermeasures in a logical order, starting with the threats that present the greatest risk.
- **Perform the security reviews**. Apply a set of security rules, known as security checklists, which constitute the security review. These checklists incorporate the knowledge acquired by performing the previous steps. The review process should be an iterative one that frequently occurs.

This chapter focuses on the security review of the Logging Application Block. However, because the material is intended to show all the aspects of a security review, there are some examples that other application blocks in the Enterprise Library. You can adapt the templates and checklists included here to suit your own security review process.

For more information about what constitutes security best practices, see Additional Resources.

Establishing the Security Requirements

The security requirements for the Enterprise Library focus on maintaining the confidentiality of the data, the integrity of the data, and the availability of the application blocks. In general, maintaining availability meant preventing denial of service attacks. Here are the Enterprise Library security requirements:

- Ensure that an attacker cannot read confidential information, such as a connection string, in the configuration files.
- Ensure that an attacker cannot read confidential information in the log message repositories, such as a database, event log, or flat file.
- Ensure that an attacker cannot alter information in the log message repositories.
- Ensure that an attacker cannot tamper with the configuration files.
- Ensure that an attacker cannot tamper with or replace the assemblies.
- Ensure that an attacker cannot launch denial of service attacks.

These requirements form the basis for the rest of the security review. To make these general requirements more specific requires a thorough understanding of how the application block works.

Analyzing the Logging Application Block

Analyzing the Logging Application Block means identifying its assets, dependencies, and subsystems. Use the security requirements as a guideline for identifying the relevant components. It is also important to identify the underlying assumptions about security that were made when the application block was first designed.

To analyze the application block, you should:

- Identify the assets.
- Create an architectural drawing.
- Identify the entry points.
- Identify the relevant classes.
- Identify the external dependencies.
- Identify the assumptions.
- Identify any other information that might affect the application block's security.

Identifying the Assets

Assets are resources that need to be protected from an attack. Table 1 lists the Logging Application Block’s assets.

Table 1: Logging Application Block Assets

Assets	Vulnerabilities
Assemblies	You should protect the Logging Application Block assemblies from malicious users who could tamper with them, replace them with other assemblies, or override them with other assemblies.
Resource files	Resource files contain static error messages. You should protect them from unauthorized read/write operations.
Configuration files	The configuration files contain separate sections for each application block and for the instrumentation. This is sensitive data that you should protect from unauthorized read/write operations.
Configuration value objects	The configuration value objects hold configuration information in memory. Configuration data is sensitive information that you should protect from unauthorized read/write operations.
Database	The Logging Application Block exposes interfaces that allow you to use the Data Access Application Block to log messages to a database. This information may be sensitive and unauthorized users should not be allowed to see it. This may also true of other information in the database. Also, you should protect the database from unauthorized read/write operations.
Database connection string	The Logging Application Block uses the Data Access Application Block to log information to a database. The Data Access Application Block uses a connection string that is specified in the configuration file to access the database. This file stores the connection string as plaintext. The connection string can include a server name, a database name, a user ID, and a password. You should protect this information from unauthorized read/write operations.
File system log files	The Logging Application Block exposes interfaces that allow you to log messages to a log file. These messages may contain sensitive information. You should use access control lists (ACL) or encryption to protect the log file from unauthorized read/write operations.
SMTP server	The Logging Application Block exposes interfaces that allow you to use an SMTP server to send e-mail. Malicious users can use the server to send unsolicited e-mail messages. You should protect the SMTP server from unauthorized access.
Message queues and distributor service	The Logging Application Block exposes interfaces that allow you to place log messages in a Message Queuing queue and distribute them to trace listeners. These messages may contain sensitive information. You should protect the queue and the messages from unauthorized read/write operations.
Event log	The Logging Application Block exposes interfaces that allow you to log messages in the event log. These messages may contain sensitive information. You should protect them from unauthorized read/write operations.

continued

Assets	Vulnerabilities
WMI events	The Logging Application Block exposes interfaces that allow you to raise instrumentation events. By providing this access, the application block exposes system resources, such as flat files, a SQL Server database, message queues, and the event log, through WMI (Windows Management Instrumentation) events. You should prevent unauthorized users from being able to raise instrumentation events.
System files	The Logging Application Block exposes interfaces that allow you to log messages to a log file. Information in the configuration file determines the location of that file. By changing this location, malicious users can modify or overwrite system files. You should protect this location from unauthorized read/write operations.
System resources	The Logging Application Block exposes interfaces that allow you to extend the application block. For example, you can add a custom log handler. Malicious users can use these extensions to expose system resources that can then be used by an unauthorized application.

Create an Architectural Diagram

A diagram is often helpful in understanding the architecture of a system. Figure 1 is a diagram of the Logging Application Block. The drawing shows the major subsystems that are of concern during a security review. For example, the diagram does not show the trace listener classes because they are trusted components. It does show the custom trace listener classes because custom code may not be trusted.

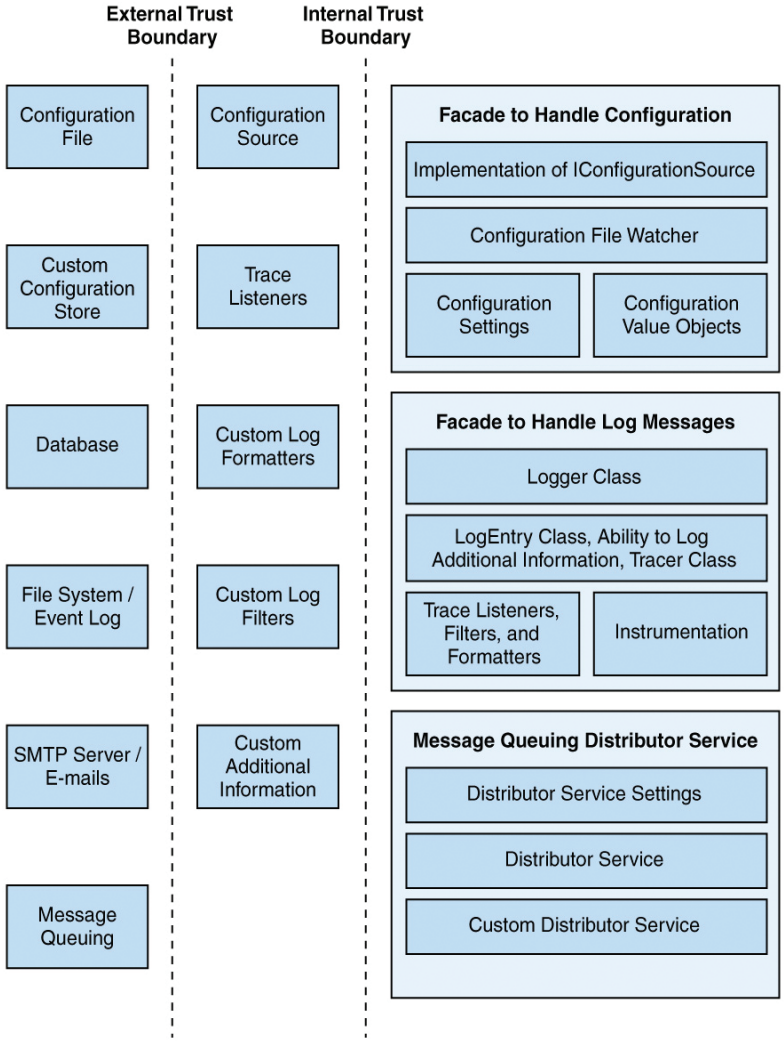

Figure 1 *Architecture of Logging Application Block*

The emphasis of the drawing can be on features instead of on specific classes. After the drawing is in place, you can use it to identify the specific entry points, classes, and dependencies.

Identify the Entry Points

Entry points are points of contact in the application block that allow external clients to interact with it, either directly or by supplying data. Table 2 lists the Logging Application Block entry points.

Table 2: Logging Application Block Entry Points

No.	Entry Points	Clients	Descriptions
1	Assemblies	Administrators and application processes	The Logging Application Block assemblies ship with the application that uses the application block.
2	Configuration file	Administrators and application processes	The Logging Application Block uses the <loggingConfiguration> and possibly the <dataConfiguration> sections of the configuration file.
3	Configuration source	Administrators and application processes	A configuration source implements the **IConfigurationSource** interface.
4	Configuration value objects	Administrators and application processes	Configuration value objects contain data about the application block's configuration.
5	Trace listeners	Administrators and application processes	Trace listeners receive tracing information and send it to an output destination, such as an event log or a database.
6	Formatting handlers	Administrators and application processes	The Logging Application Block can format log messages with a text formatter, a binary formatter, or a custom formatter.
7	Logging filters	Administrators and application processes	The Logging Application Block can filter messages with a category filter, a priority filter, or a custom filter.
8	Database	Administrators and application processes	The Logging Application Block can log messages to a database.
9	Event log	Administrators and application processes	The Logging Application Block can log messages to an event log.
10	Flat files	Administrators and application processes	The Logging Application Block can log messages to a flat file
11	Message queuing and distributor service	Administrators, application processes, Message Queuing, the distributor service, and the Windows Service Identity page	The Logging Application Block's distributor service uses message queuing to asynchronously distribute log messages. The service uses the Windows Service Identity page to access a message queue.
12	Public classes and static methods	Administrators and application processes	The **Logger**, **LogEntry**, and **NativeMethods** classes expose static methods. (The **NativeMethods** class is a managed class that includes wrappers to call Win32 APIs.)

Identify the Relevant Classes

Identify the relevant classes and the ways you expect them to be used. These are classes that, if misused, can affect the security of the application block. Table 3 lists the Logging Application Block classes.

Table 3: Logging Application Block Classes

No.	Scenario
1	The **Logger** class can access the default configuration file.
2	The **EnterpriseLibraryFactory** class can access custom configuration sources and the default configuration source.
3	The **CustomTraceListener** class is the base class for custom trace listeners.
4	The **EmailTraceListener** class creates e-mail messages from log messages and sends them.
5	The **FormattedEventLogTraceListener** class logs information to the event log.
6	The **FormattedTextWriterTraceListener** class uses streams to log messages.
7	The **FlatFileTraceListener** class logs messages to flat files.
8	The **WMITraceListener** class raises instrumentation events.
9	The **MsmqTraceListener** class sends log messages to a message queue.
10	The **FormatterDatabaseTraceListener** class logs information to a database.
11	The **LoggingInstrumentationProvider** class implements the instrumentation. The **LoggingInstrumentationListener** class receives performance counter data.
12	The **CategoryFilter** class filters log messages based on their categories. Users specify these categories when they configure the application block.
13	The **LogEnabledFilter** class enables and disables logging.
14	The **PriorityFilter** class filters log messages based on their priorities.
15	The I**LogFilter** interface is the basis for custom log filters that filter log messages according to their attributes.
16	The **BinaryLogFormatter** class serializes log messages in binary format.
17	The **TextFormatter** class formats log messages according to a specified template.
18	The **ILogFormatter** interface is the basis for custom log formatters.
19	The **ComPlusInformationProvider** class gathers information about COM objects that can be included in a log message.
20	The **DebugInformationProvider** class gathers diagnostic debugging information, such as stack traces that can be included in a log message.
21	The **ManagedSecurityContextInformationProvider** class gathers diagnostic information about the managed code security context that can be included in a log message.
22	The **UnmanagedSecurityContextInformationProvider** class gathers diagnostic information about the unmanaged code security context.
23	The **IExtraInformationProvider** interface is the basis for custom configuration classes, such as dictionaries, that gather diagnostic information.
24	The **MsmqListener** class configures a message queue.
25	The **MsmqLogDistributor** class transfers the message queue SOAP messages to trace sources that are specified in the configuration file.

continued

No.	Scenario
26	The **Tracer** class marks the start and end of a transaction.
27	The **LoggingDatabase.SQL** and **CreateLoggingDb.Cmd** methods create logging databases. (The Enterprise Library Windows Installer creates these methods.)
28	The **LogWriterFactory** class creates **LogWriter** objects.
29	The **LogSource** class specifies the trace listeners that log the messages.
30	The **TracerInstrumentationListener** class instruments **Tracer** class tracing operations.

Identify the External Dependencies

External dependencies are other components in the Enterprise Library or external resources such as databases and mail servers that the Logging Application Block can use. If a malicious user misuses any of these dependencies then the security of the application block may be compromised. The data direction matters because "push" and "pull" have different security implications. The Logging Application Block receives data that moves in the "pull" direction. For example, it receives configuration information from the Enterprise Library Core. It is important to know that this information is reliable. The application block can also "push" information to output sources, such as a database or event log. It is important to know that the output source will not be corrupted by the information.

Table 4 lists the Logging Application Block external dependencies.

Table 4: Logging Application Block External Dependencies

No.	External Dependencies	Descriptions	Data Direction	Trusted?
1	Enterprise Library core	The Enterprise Library Core reads and writes configuration information that is used by the Logging Application Block.	Pull	Yes
2	Data Access Application Block	The Logging Application Block can use the Data Access Application Block to log messages to a database.	Push	Yes
3	SMTP server	The Logging Application Block can use an SMTP server to e-mail log messages.	Push	Yes
4	File system	The Logging Application Block can use the file system to save log messages in a flat file.	Push	Yes
5	Database	The Logging Application Block can save log messages to a database.	Push	Yes
6	Message queuing	The Logging Application Block uses message queuing to distribute log messages. The application block can send these messages to a remote computer.	Push	Yes
7	Event log	The Logging Application Block can save log messages to the event log.	Push	Yes

No.	External Dependencies	Descriptions	Data Direction	Trusted?
8	WMI	The Logging Application Block uses WMI to raise events.	Push	Yes
9	Performance counters	The Logging Application Block uses performance counters to track its performance.	Push	Yes

Identify the Implementation Assumptions

Implementation assumptions are premises about how the application block works. Implementers describe these assumptions when they write the specification for the application block and before they begin writing the code. Typically, these assumptions are reviewed again once the implementation is complete. Table 5 lists the implementation assumptions about the Logging Application Block. The term "application" refers to the application that uses the Logging Application Block.

Table 5: Logging Application Block Implementation Assumptions

No.	Category	Assumptions
1	ACLs for event log	The event log's ACLs protect the log against unauthorized users and processes.
2	Database access privileges	The application has the appropriate privileges to access the database.
3	Message queuing access privileges	The application has the appropriate privileges to access the message queues.
4	File system privileges	The application has the appropriate privileges to access the file system.
5	SMTP server privileges	The application has the appropriate privileges to use the SMTP server to send e-mail messages.
6	WMI privileges	The application has the appropriate privileges to raise WMI events.
7	The application that uses the Data Access Application Block to log messages to a database	The application can identify and authorize the Data Access Application Block. In this case, "identify" means that the application can trust the Data Access Application Block assemblies. Authorization means that the application has the correct SQL Server permissions to allow the appropriate groups of users to read and write to the database.
8	Other subsystems and application blocks	The Logging Application Block is dependent on the Enterprise Library Core for configuration information and the Data Access Application Block to use the **Database Trace Listener**. The Logging Application Block must ensure that it uses the correct Enterprise Library Core assemblies for configuration and the correct Data Access Application Block assemblies to log messages to a database.
9	Performance counters	The Logging Application Block requires the necessary read and write access permissions to use the performance counters.

continued

Identify Any Additional Security Notes

Additional security notes are other threats or information that are not covered elsewhere. Table 6 lists the Logging Application Block additional security notes.

Table 6: Logging Application Block Additional Security Notes

No.	Notes
1	The configuration file or the custom configuration store should be protected by ACLs or, if possible, encrypted.
2	The application must have the proper ACLs and privileges to log messages to different resources such as the event log, a database, a message queue, an SMTP server, and the file system. The application must also have the correct ACLs and privileges to raise WMI events.
3	Log messages that contain sensitive information should be protected by ACLs or else encrypted.
4	Log messages should be encrypted when sent over a network.

Building the Threat Models

After you have analyzed the application block, you can build the threat models. Threat models identify threats against specific resources, assets, and trust boundaries, pinpoint vulnerabilities, and provide countermeasures. Each table contains a STRIDE classification. STRIDE is the acronym used at Microsoft to categorize different threat types. STRIDE stands for Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, and Elevation of privilege. To learn more about STRIDE see *Threats and Countermeasures* in *Improving Web Application Security: Threats and Countermeasures* on MSDN.

A DREAD table follows each threat model. DREAD stands for Damage potential, Reproducibility, Exploitability, Affected users, and Discoverability. In DREAD, you assign each of these categories a number that rates the potential risk it poses to your application. To learn more about DREAD, see *Threat Modeling* in *Improving Web Application Security: Threats and Countermeasures* on MSDN.

Table 7 lists details about threat 1.

Table 7: Logging Application Block Threat 1

Threat 1	Logging Application Block assemblies are not strong named.
Name	Tampering with assemblies
Entry points	Assemblies
Threat description	The Logging Application Block assemblies are not strong named.

Threat 1	Logging Application Block assemblies are not strong named.
Countermeasures	The Logging Application Block assemblies must be strong named in order to prevent malicious users from tampering with them or replacing them. Tampering can occur when the assemblies are distributed or when the application that uses the Logging Application Block is installed. Replacing the legal assemblies with other assemblies can occur after the application is installed. The legal assemblies also can be overridden at run time. In addition to being strong named, assemblies should be signed with Authenticode. Strong naming ensures the integrity of the code and Authenticode ensures its authenticity. For more information about Authenticode, see *Authenticode* on MSDN.
STRIDE classification	Tampering, Information Disclosure, Denial of Service, Elevation of Privileges
Risk	High
Mitigation	No
Investigative notes	None

Table 8 lists the DREAD rating for threat 1.

Table 8: Threat 1 DREAD Rating

D	R	E	A	D	Total	Rating
3	3	2	3	3	14	High

Table 9 lists details about threat 2.

Table 9: Logging Application Block Threat 2

Threat 2	Attackers can alter the configuration files.
Name	Tampering with configuration files
Entry points	Configuration files
Threat description	The configuration files are in plaintext. They contain information about trace listeners, formatters, and filters. An attacker can alter this information and also add custom handlers. These actions can change the behavior of the application block and expose sensitive information.
Countermeasures	The configuration files should be encrypted or else plaintext configuration files should not be used. The application should use the Cryptography Application Block in conjunction with the Logging Application Block to encrypt the configuration file.
STRIDE classification	Tampering, Information Disclosure, Denial of Service
Risk	High
Mitigation	No
Investigative notes	None

Table 10 lists the DREAD rating for threat 2.

Table 10: Threat 2 DREAD Rating

D	R	E	A	D	Total	Rating
3	3	2	3	3	14	High

Table 11 lists details about threat 3.

Table 11: Logging Application Block Threat 3

Threat 3	Attackers can flood the event log with false error messages.
Name	Flooding the event log
Entry points	Public classes and static methods
Threat description	The Logging Application Block exposes interfaces that allow you to log messages to the event log. An attacker can use these interfaces to flood the event log with false messages. This can constitute a denial of service attack. In addition, the event log may contain sensitive information.
Countermeasures	Check the event log to see if it has reached its threshold limit, which controls how many messages can be in the log. If the event log has reached its limit, generate an exception. The application should appropriately handle this exception. Also, the application should include code that uses an instance of the **EventLogPermission** class. Finally, the administrator should assign the appropriate ACLs to the event log for read/ write operations.
STRIDE classification	Tampering, Information Disclosure, Denial of Services, Elevation of Privileges
Risk	High
Mitigation	No
Investigative notes	None

Table 12 lists the DREAD rating for threat 3.

Table 12: Threat 3 DREAD Rating

D	R	E	A	D	Total	Rating
3	3	2	3	3	14	High

Table 13 lists details about threat 4.

Table 13: Logging Application Block Threat 4

Threat 4	Attackers can flood the database log with false messages.
Name	Flooding the database
Entry points	Public classes and static methods

Threat 4	Attackers can flood the database log with false messages.
Threat description	The Logging Application Block exposes interfaces that allow an application to use the Data Access Application Block to log messages to a database. An attacker can use these interfaces to flood the database with false messages. This can eventually cause a denial of service. In addition, log messages in the database may contain sensitive information.
Countermeasures	The Data Access Application Block requires explicit Code Access Security permissions to access SQL Server or an Oracle database. No such permissions are necessary to access a generic database. The **GenericDatabase** class's access methods have overloads that can request custom permissions that derive from the .NET Framework **DBDataPermission** class. These permissions ensure that the application has the proper level of security to access the database.
STRIDE classification	Tampering, Denial of Services, Elevation of Privileges
Risk	High
Mitigation	Yes
Investigative notes	None

Table 14 lists the DREAD rating for threat 4.

Table 14: Threat 4 DREAD Rating

D	R	E	A	D	Total	Rating
3	3	2	3	3	14	High

Table 15 lists details about threat 5.

Table 15: Logging Application Block Threat 5

Threat 5	Attackers can flood the flat file log with false messages.
Name	Flooding the flat file log
Entry points	Public classes and static methods
Threat description	The Logging Application Block exposes interfaces that allow you to log messages to a flat file. An attacker can use these interfaces to flood the flat file with false messages. This can constitute a denial of service attack. In addition, log messages in the flat file may contain sensitive information.
Countermeasures	Check the hard disk space to see if it has reached its threshold limit, which controls how many messages can be in the file. If the flat file has reached its limit, generate an exception. The application should appropriately handle this exception. Also, the application should include code that uses an instance of the **FileIOPermission** class. Finally, the administrator should assign the appropriate ACLs to the event log for read/ write operations.

continued

Threat 5	Attackers can flood the flat file log with false messages.
STRIDE classification	Tampering, Information Disclosure, Denial of Services, Elevation of Privileges
Risk	High
Mitigation	No
Investigative notes	None

Table 16 lists the DREAD rating for threat 5.

Table 16: Threat 5 DREAD Rating

D	R	E	A	D	Total	Rating
3	3	2	3	3	14	High

Table 17 lists details about threat 6.

Table 17: Logging Application Block Threat 6

Threat 6	Log messages that are stored in the event log, the database, a message queue, or a flat file are not protected.
Name	Unprotected log messages
Entry points	Flat files, database, event log, message queue, and distributor service
Threat description	The Logging Application Block exposes interfaces that allow you to log messages to different resources. These resources include the event log, flat files, a database, message queuing, WMI events, and e-mail. Log messages are stored as plaintext both in memory and in a physical storage system. An attacker can read these messages if they are not properly protected with ACLs, authorization, or encryption.
Countermeasures	The log messages should be encrypted whether they are stored or being transferred over a network or with message queuing. If the log messages are not encrypted, attackers can use network monitoring tools to read them. Also, the administrator should assign the appropriate ACLs to the event log, database, flat file, or message queue and distributor service.
STRIDE classification	Tampering, Information Disclosure
Risk	High
Mitigation	No
Investigative notes	None

Table 18 lists the DREAD rating for threat 6.

Table 18: Threat 6 DREAD Rating

D	R	E	A	D	Total	Rating
3	3	2	3	3	14	High

Table 19 lists details about threat 7.

Table 19: Logging Application Block Threat 7

Threat 7	Attackers can use some of the application block interfaces to send unsolicited e-mail messages.
Name	Unsolicited e-mail
Entry points	Public classes and static methods
Threat description	The Logging Application Block exposes interfaces that allow you to e-mail log messages. An attacker can use these interfaces to send unsolicited e-mail messages.
Countermeasures	The administrator should assign the appropriate ACLs to the SMTP server.
STRIDE classification	Information Disclosure, Denial of Services
Risk	High
Mitigation	No
Investigative notes	None

Table 20 lists the DREAD rating for threat 7.

Table 20: Threat 7 DREAD Rating

D	R	E	A	D	Total	Rating
3	3	2	3	3	14	High

Table 21 lists details about threat 8.

Table 21: Logging Application Block Threat 8

Threat 8	Attackers can flood the message queues with false messages.
Name	Flooding of message queues and distributor service
Entry points	Public classes and static methods
Threat description	The Logging Application Block exposes interfaces that allow you to send messages to the message queues so that the log messages can be asynchronously processed. An attacker can use these interfaces to send false messages to the message queue. The distributor service then processes these messages. These false messages can constitute a denial of service attack. In addition, messages in a message queue may contain sensitive information.
Countermeasures	The application should perform authentication and authorization before granting a user access to the message queue.
STRIDE classification	Tampering, Information Disclosure, Denial of Services, Elevation of Privileges
Risk	High
Mitigation	No
Investigative notes	None

Table 22 lists the DREAD rating for threat 8.

Table 22: Threat 8 DREAD Rating

D	R	E	A	D	Total	Rating
3	3	2	3	3	14	High

Table 23 lists details about threat 9.

Table 23: Logging Application Block Threat 9

Threat 9	Attackers can flood WMI event instrumentation with false events.
Name	Flooding WMI events
Entry points	Public classes and static methods
Threat description	The Logging Application Block exposes interfaces that allow you to raise WMI events. An attacker can use these interfaces to raise false instrumentation events. This can constitute a denial of service attack.
Countermeasures	Validate the input.
STRIDE classification	Denial of Services, Elevation of Privileges
Risk	High
Mitigation	No

Threat 9	Attackers can flood WMI event instrumentation with false events.

Investigative notes	The following code examples show how the application block should first validate the input to the **Logger.Write** method before it raises an event.
	`public static void Write(object message, ICollection<string> categories, int priority, int eventId, TraceEventType severity, string title, IDictionary properties)`
	`{`
	`  LogEntry log = new LogEntry();`
	`//input validation should have been done here for message`
	`  log.Message = message.ToString();`
	`  log.Categories = categories;`
	`  log.Priority = priority;`
	`  log.EventId = eventId;`
	`  log.Severity = severity;`
	`  log.Title = title;`
	`  log.ExtendedProperties = properties;`
	`  Write(log);`
	`}`
	The following examples show other places that require input validation.
	`Logger.Write(object,..) for message`
	`SoapLogFormatter. DeserializeLogEntry(string) for serializedLogEntry`
	`ContextItems.ProcessContextItems(LogEntry) for log`
	`LoggingSettings.GetLoggingSettings(IConfigurationSource) for configurationSource`
	`ComPlusInformationProvider.PopulateDictionary(IDictionary) for dict`
	`DebugInformationProvider.PopulateDictionary(IDictionary) for dict`
	`ManagedSecurityContextInformationProvider.PopulateDictionary(IDictionary) for dict`
	`UnmanagedSecurityContextInformationProvider.PopulateDictionary(IDictionary) for dict`
	`DebugUtils.GetStackTraceWithSourceInfo(StackTrace) for stackTrace`
	`PriorityFilter.Filter(LogEntry) for log`
	`DictionaryToken.FormatToken(String, LogEntry) for log`
	`KeyValueToken.FormatToken(String, LogEntry) for log`
	`TimeStampToken.FormatToken(String, LogEntry) for log`
	`TokenFunction.Format(StringBuilder, LogEntry) for messageBuilder`
	`FormatterDatabaseTraceListener.ValidateParameters(LogEntry) for logEntry`
	`MsmqListener.MsmqListener(DistributorService, Int32, String) for distributorService`
	`MsmqDistributorSettings.GetSettings(IConfigurationSource) for configurationSource`

Table 24 lists the DREAD rating for threat 9.

Table 24: Threat 9 DREAD Rating

D	R	E	A	D	Total	Rating
3	3	2	3	3	14	High

Table 25 lists details about threat 10.

Table 25: Logging Application Block Threat 10

Threat 10	Input validation is not performed
Name	The Logging Application Block does not perform input validation.
Entry points	Public classes and static methods
Threat description	The **Logger** class exposes the static **Write** method. This method accepts a log message as a parameter. However, it does not perform input validation on the message to check for **NULL** values. An invalid input can cause the application block to throw an unhandled exception to the application.
Countermeasures	The application block must perform input validation. For details, see the investigative notes for threat 9.
STRIDE classification	Information Disclosure
Risk	High
Mitigation	No
Investigative notes	See threat 9.

Table 26 lists the DREAD rating for threat 10.

Table 26: Threat 10 DREAD Rating

D	R	E	A	D	Total	Rating
3	3	2	3	3	14	High

Table 27 lists details about threat 11.

Table 27: Logging Application Block Threat 11

Threat 11	Altering the log file's directory path in the configuration file
Name	Flat file path canonical input validation
Entry points	Configuration files, custom configuration stores
Threat description	The Logging Application Block allows you to log messages to a flat file. The location of this file is included in the configuration file or custom configuration source. An attacker can change the file's directory path to point away from the flat file and to a system file. The attacker can then send harmful log messages to that system file.
Countermeasures	The application block should only log messages to a file in the current directory. All log files should have a .txt file name extension.

Threat 11	Altering the log file's directory path in the configuration file
STRIDE classification	Tampering, Denial of Services, Elevation of Privileges
Risk	High
Mitigation	No
Investigative notes	None

Table 28 lists the DREAD rating for threat 11.

Table 28: Threat 11 DREAD Rating

D	R	E	A	D	Total	Rating
3	3	2	3	3	14	High

Table 29 lists details about threat 12.

Table 29: Logging Application Block Threat 12

Threat 12	The Debug.Assert statements can cause a denial of service when the application is in debug mode
Name	**Debug.Assert** statements halt code while in debug mode
Entry points	Public classes and static methods
Threat description	The **LogEntry** public class has a static method named **GetProcessName**. The method uses the **NativeMethods** class to retrieve the name of the current process. (The **NativeMethods** class calls unmanaged APIs.) If an exception is thrown inside a call to unmanaged code and the process name is not properly returned, the **Debug.Assert** statement displays a message box. This message box requires a user to select **Abort**, **Retry**, or **Ignore** before the application can proceed. This can constitute a denial of service attack in the case of service applications.
Countermeasures	Remove all **Debug.Assert** statements from the Logging Application Block code before it is shipped.
STRIDE classification	Denial of Services
Risk	High
Mitigation	No

Investigative notes	The following code shows an example of a **Debug.Assert** statement that should be removed.
	`public static string GetProcessName()`
	`{`
	`  StringBuilder buffer = new StringBuilder(1024);`
	`  int length = NativeMethods.GetModuleFileName(NativeMethods .GetModuleHandle(null), buffer, buffer.Capacity);`
	`  Debug.Assert(length > 0);`
	`  //This line can halt the code when in debug mode.`
	`  return buffer.ToString();`
	`}`

Table 30 lists the DREAD rating for threat 12.

Table 30: Threat 12 DREAD Rating

D	R	E	A	D	Total	Rating
3	3	2	3	3	14	High

Performing Security Reviews

The security reviews are the final steps in ensuring that the application block follows security best practices. The security reviews for Enterprise Library focused on the code, access considerations, and design and deployment. All of the security reviews had the following common characteristics:

- Security reviews were done on small pieces of code and reviews had multiple iterations.
- Security reviews were performed in a timely manner to avoid backlogs and to ensure that problems were discovered as early as possible.
- The security reviews were performed by more than one tester and/or developer. The reviewers were people with expertise in the pertinent areas. For example, people with experience in cryptography performed the security reviews of cryptography and secrets.
- Security review checklists were used to make sure that all the relevant points were covered and to serve as documentation.
- FXCOP was used as the analysis tool.

Security Review Checklists

Checklists enumerate recommendations as itemized lists. The checklists included here were used in the Enterprise Library security reviews. You can use them as models or templates. The categories of checklists are:

- General code review
- Managed code review
- Resource access
- Code access
- Design and deployment

General Code Review Checklist

Table 31 lists the general code review recommendations.

Table 31: General Code Review Checklist

Check	Description
Yes	Clearly document potential threats and log them in a bug tracking database. (Threats are dependent on the specific scenario and application block.)
Yes	Develop code that is based on the .NET Framework design guidelines. For more information, see *Design Guidelines for Class Library Developers* on MSDN.
Yes	Run the FxCop analysis tool on assemblies and address all security warnings.

Managed Code Review Checklists

The checklists that are in the managed code category are:

- Assembly-level checklist
- Class-level checklist
- Cryptography checklist
- Secrets checklist
- Exception management checklist
- Delegates checklist
- Serialization checklist
- Threading checklist
- Reflection checklist
- Unmanaged code access checklist

Assembly-level Checklist

Table 32 lists the assembly-level recommendations.

Table 32: Assembly-level Checklist

Check	Description
No	Assemblies can have a strong name to guarantee that no one has tampered with them. The threat model specifies that customers should sign assemblies for this purpose. However, because the Enterprise Library ships as source code, the assemblies do not have strong names.
Yes	Consider delay signing the assemblies to reduce exposure of the private key that is used in the strong naming and signing processes.

continued

Check	Description

Yes	Assemblies can include declarative security attributes that are implemented with the .NET Framework **SecurityAction.RequestMinimum** enumeration. This enumeration requests the minimum permissions required for the code to run. The run time loads the assemblies only if the security policy can grant them the permissions they need. Additionally, specifying the necessary permission level in the code lets administrators know what the application needs to successfully load. The following example shows how the Logging Application Block's AssemblyInfo.cs file requests permissions.
	`[assembly: ReflectionPermission(SecurityAction.RequestMinimum, Flags = ReflectionPermissionFlag.MemberAccess)]`
	`[assembly: FileIOPermission(SecurityAction.RequestMinimum)]`
	`[assembly: EventLogPermission(SecurityAction.RequestMinimum)]`
	`[assembly: MessageQueuePermission(SecurityAction.RequestMinimum, Unrestricted = true)]`
	`[assembly: PerformanceCounterPermission(SecurityAction.RequestMinimum)]`

Class-level Checklist

Table 33 lists the class-level recommendations.

Table 33: Class-level Checklist

Check	Description
Yes	Restrict the visibility of classes and their members. Use the most restrictive access modifier you can. Use **private** where possible.
Yes	Seal non-base classes.

Yes	Validate all input that originates outside of the current trust boundary. Check the input to see that it is the proper type, length, format, and range. The following example shows how the Logging Application Block checks the input for non-NULL values and valid files.
	`Public FileConfigurationSource(string configurationFilepath)`
	`{`
	`  if (string.IsNullOrEmpty(configurationFilepath)) throw new ArgumentException(Resources.ExceptionStringNullOrEmpty, "configurationFilepath");`
	`  this.configurationFilepath = RootConfigurationFilePath(configurationFilepath);`
	
	`  if (!File.Exists(this.configurationFilepath)) throw new FileNotFoundException(string.Format(Resources.Culture, Resources.ExceptionConfigurationLoadFileNotFound, this.configurationFilepath));`
	`  EnsureImplementation(this.configurationFilepath);`
	`}`

Check	Description
Yes	Implement declarative checks for virtual internal methods. Derived classes can override virtual internal methods. This can change the behavior of the application block. (Public types do not have internal virtual members so they do not need these checks.)

Yes	Fields should be private. When necessary, expose field values with read-write or read-only public properties. The following example shows how to use the read-only property to prevent an attacker from changing the tokens that make up a database connection string.
	`public abstract class Database : IInstrumentationEventProvider`
	`{`
	`  private static readonly string VALID_USER_ID_TOKENS =`
	`  Resources.UserName;`
	`  private static readonly string VALID_PASSWORD_TOKENS =`
	`  Resources.Password;`
	(Code access security checks do not apply to fields.)

Yes	Use read-only properties where possible.
Yes	Review how the application block uses event handlers.

Cryptography Checklist

Table 34 lists the cryptography recommendations.

Table 34: Cryptography Checklist

Check	Description
Yes	Use the .NET Framework-provided cryptography providers instead of custom providers. The Cryptography Application Block wraps standard .NET providers. Customers can choose to use a custom provider but this is not a recommended practice.

continued

Check	Description

Yes	Use DPAPI to encrypt configuration secrets so as to eliminate the key management issue. (DPAPI uses the password of the user account associated with the code that calls the DPAPI functions in order to derive the encryption key. As a result, the operating system, and not the application, manages the key.) The Cryptography Application Block uses DPAPI to encrypt the key. It retrieves the key from the key file or cache each time it must encrypt or decrypt information. The following code example shows how the Cryptography Application Block stores and retrieves the key.
	`public static ProtectedKey Read(string protectedKeyFileName, DataProtectionScope dpapiProtectionScope)`
	`{`
	`  string completeFileName =`
	`  Path.GetFullPath(protectedKeyFileName);`
	`  if (cache[completeFileName] != null) return`
	`  cache[completeFileName];`
	
	`  using (FileStream stream = new`
	`  FileStream(protectedKeyFileName, FileMode.Open,`
	`  FileAccess.Read, FileShare.Read))`
	`  {`
	`  ProtectedKey protectedKey = Read(stream,`
	`  dpapiProtectionScope);`
	`  cache[completeFileName] = protectedKey;`
	
	`  return protectedKey;`
	`  }`
	`}`
	The following code example shows how to use DPAPI to decrypt the key.
	`public byte[] Encrypt(byte[] plaintext)`
	`{`
	`  byte[] output = null;`
	`  byte[] cipherText = null;`
	`  this.algorithm.Key = Key;`
	`  private byte[] Key`
	`  {`
	`  get`
	`  {`
	`  return key.DecryptedKey;`
	`  }`
	`  }`
	`}`
	
	`public byte[] DecryptedKey`
	`{`
	`  get { return Unprotect(); }`
	`}`
	
	`public virtual byte[] Unprotect()`
	`{`
	`  return ProtectedData.Unprotect(protectedKey, null,`
	`  protectionScope);`
	`}`

Check	Description
Yes	Use the appropriate key sizes for the chosen cryptography algorithm. Identify and document any reasons for not following this guideline.
Yes	Do not store keys in code and configuration files.
Yes	Restrict access to persisted keys.
Yes	Periodically cycle keys.

Secrets Checklist

Table 35 lists the secrets recommendations.

Table 35: Secrets Checklist

Check	Description
Yes	Do not hard-code secrets.
No	Do not store plaintext secrets in configuration files. The connection strings that the Data Access Application Block uses to connect to databases are stored in the configuration files. The Data Access Application Block threat model documents this issue. It recommends that you encrypt the configuration file. For more information on how to do this, see Configuring the Application Blocks in the Enterprise Library documentation.
No	Do not store plaintext secrets in memory for extended periods of time. Currently, the Caching Application Block cannot encrypt information.

continued

Check	Description

Yes	Clear sensitive data from memory as soon as possible. The Cryptography Application Block byte arrays contain unencrypted keys. The application block clears them from memory as soon as it uses them. The following example shows how the application block uses a key and then removes it.
	`public byte[] Encrypt(byte[] plaintext)`
	`{`
	`byte[] output = null;`
	`byte[] cipherText = null;`
	
	`this.algorithm.Key = Key;`
	
	`using (ICryptoTransform transform =`
	`this.algorithm.CreateEncryptor())`
	`{`
	`cipherText = Transform(transform, plaintext);`
	`}`
	
	`output = new byte[IVLength + cipherText.Length];`
	`Buffer.BlockCopy(this.algorithm.IV, 0, output, 0,`
	`IVLength);`
	`Buffer.BlockCopy(cipherText, 0, output, IVLength,`
	`cipherText.Length);`
	
	`CryptographyUtility.ZeroOutBytes(this.algorithm.Key);`
	
	`return output;`
	`}`
	
	`public static void ZeroOutBytes(byte[] bytes)`
	`{`
	`if (bytes == null)`
	`{`
	`return;`
	`}`
	`Array.Clear(bytes, 0, bytes.Length);`
	`}`

Exception Management Checklist

Table 36 lists the general exception management recommendations.

Table 36: Exception Management Checklist

Check	Description

Yes	Use exception handling. Catch only the exceptions that you have anticipated in your design. Exceptions to this rule are exceptions that are not Common Language Specification (CLS)-compliant. (These are exceptions that do not derive from the .NET Framework **System.Exception** namespace.) These exceptions can occur if the application block uses a customer-supplied extension of if there is a call to unmanaged code. The application block should include a generic catch handler for these situations. (In C#, this takes the form of **catch{ ... }**). The inability to handle non-CLS-compliant exceptions may leave the application block vulnerable to denial of service attacks.
	An example of this situation is when the Data Access Application Block attempts to use a custom database class that is included in the configuration source. The following code example shows how to handle any non-CLS-compliant exceptions that may occur.
	`private static void TryLogConfigurationError(ConfigurationErrorsException configurationException, string instanceName)`
	`{`
	`try`
	`{`
	`DefaultDataEventLogger eventLogger = EnterpriseLibraryFactory.BuildUp<DefaultDataEventLogger>();`
	`if (eventLogger != null)`
	`{`
	`eventLogger.LogConfigurationError(configurationException, instanceName);`
	`}`
	`}`
	`catch { }`
	`}`

Yes	Log the details of the exceptions to assist in diagnosing problems.

Yes	All code paths that can result in an exception should provide a way to first check for a successful outcome before throwing an exception. In the following code example, the code first checks to see that the configuration file exists and is valid before it throws an exception.
	`Public FileConfigurationSource(string configurationFilepath)`
	`{`
	`if (string.IsNullOrEmpty(configurationFilepath)) throw new ArgumentException(Resources.ExceptionStringNullOrEmpty, "configurationFilepath");`
	`this.configurationFilepath = RootConfigurationFilePath(configurationFilepath);`
	`if (!File.Exists(this.configurationFilepath)) throw new FileNotFoundException(string.Format(Resources.Culture, Resources.ExceptionConfigurationLoadFileNotFound, this.configurationFilepath));`
	`EnsureImplementation(this.configurationFilepath);`
	`}`

Yes	If there is a problem, an exception should occur as early as possible to avoid needlessly using any more resources.

Delegates Checklist

Table 37 lists the delegates recommendations.

Table 37: Delegates Checklist

Check	Description

Yes	Delegates should not be accepted from untrusted sources. In Enterprise Library, it is possible to use external configuration sources. This threat is documented in the code so that users are aware of it. The following code example shows this documentation.
	`LogEntry log = new LogEntry();`
	`log.Message = "memory leak";`
	`log.Categories.Add(DropDownList1.SelectedValue);`
	`log.Priority = 0;`
	`log.EventId = 100;`
	`log.Severity = TraceEventType.Information;`
	`FileConfigurationSource source = new`
	`// This configuration source comes from an external source.`
	`// Only use configuration sources that come from`
	`// trusted sources.`
	`FileConfigurationSource(@"c:\pag\FileSource.config");`
	`LogWriterFactory factory = new LogWriterFactory(source);`
	`LogWriter writer = factory.Create();`
	`writer.Write(log);`
	`writer.Dispose();`

Serialization Checklist

Table 38 lists the serialization recommendations.

Table 38: Serialization Checklist

Check	Description

Yes	Any type that implements the **ISerializable** interface or derives from such a type should protect the **GetObjectData** method with a serialization formatter security action. In the Security Application Block, the **SyntaxException** type implements the **GetObjectData** method and protects it with the **SecurityAction.Demand** enumeration. This means that all callers higher in the call stack must have been granted the permission specified by the current permission object. The following example shows how the Security Application Block protects the **GetObjectData** method.
	`public class SyntaxException : Exception`
	`{`
	`[SecurityPermission(SecurityAction.Demand, SerializationFormatter=true)]`
	`public override void GetObjectData(SerializationInfo info, StreamingContext context)`
	`{`
	`  base.GetObjectData(info, context);`
	`  info.AddValue(IndexKey, this.index);`
	`}`

Check	Description
Yes	Restrict serialization to privileged code.

Yes	Do not serialize sensitive data or else document it if you do. In the Logging Application Block, the **LogEntry** object contains the logging information. Users who log sensitive information should be aware that that the data is serialized, The following example shows that the **LogEntry** class has a **Serializable** attribute.
	`[Serializable]`
	`[InstrumentationClass(InstrumentationType.Event)]`
	`public class LogEntry : ICloneable`

Yes	Validate field data from serialized data streams.

Threading Checklist

Table 39 lists the threading recommendations.

Table 39: Threading Checklist

Check	Description
Yes	Stress test the application blocks to guarantee that they are not susceptible to denial of service attacks. For example, the Caching Application Block's background scheduler that is used for scavenging and expiration was tested to ensure that its threads do not deadlock or leak (this means that the threads are properly released), and that each request does not generate a new thread.
Yes	Synchronize **Dispose** methods.

Reflection Checklist

Table 40 lists the reflection recommendations.

Table 40: Reflection Checklist

Check	Description
Yes	Callers cannot influence dynamically generated code (for example, by passing assembly and type names as input arguments). It is possible to alter the configuration files to dynamically generate code. Threat 2 in Building the Threat Models documents this issue.
Yes	Use full assembly names when the .NET Framework's **Activator.CreateInstance** method creates an instance of the specified type.

Unmanaged Code Access Checklist

Table 41 lists the unmanaged code access recommendations.

Table 41: Unmanaged Code Access Checklist

Check	Description
Yes	Constrain and validate input and output strings that are passed between managed and unmanaged code.

Yes	Assemblies that call unmanaged code use declarative security (**SecurityAction.RequestMinimum**) to specify unmanaged security permissions. The application blocks use API wrappers to call unmanaged APIs. All of the application blocks' AssemblyInfo files contain the **SecurityAction.RequestMinimum** enumeration. The following code example shows how to use declarative security.
	`[assembly : SecurityPermission(SecurityAction.RequestMinimum, Flags=`
	`SecurityPermissionFlag.SerializationFormatter |`
	`SecurityPermissionFlag.ControlThread |`
	`SecurityPermissionFlag.UnmanagedCode`
	`)]`

Resource Access Checklist

Table 42 lists the resource access recommendations.

Table 42: Resource Access Checklist

Check	Description
Yes	Security decisions are not made based on file names.
Yes	Input file paths and file names are well formed.
Yes	Stress testing does not detect any memory leaks and concurrent scenarios do not cause deadlocks. This reduces the possibility of denial of service attacks.

Check	Description

Yes	Specify an assembly's file I/O requirements with declarative security attributes (this should be **SecurityAction.RequestMinimum**). For example, the Caching Application Block's **FileDependecy** method calls the **EnsureTargetFileAccessible** method. This method demands **Read** permission to access a file in case it is a protected resource. The following code example demonstrates this.
	`[assembly : FileIOPermission(SecurityAction.RequestMinimum)]`
	`In the Caching Application Block, the FileDependency class calls the EnsureTargetFileAccessible method to demand I/O permission to read a file.`
	
	`public FileDependency(string fullFileName)`
	`{`
	`  if (Object.Equals(fullFileName, null))`
	`  {`
	`  throw new ArgumentNullException("fullFileName",`
	`  SR.ExceptionNullFileName);`
	`  }`
	`  if (fullFileName.Length == 0)`
	`  {`
	`  throw new ArgumentOutOfRangeException("fullFileName",`
	`  SR.ExceptionEmptyFileName);`
	`  }`
	`  dependencyFileName = Path.GetFullPath(fullFileName);`
	`  EnsureTargetFileAccessible();`
	`  if (!File.Exists(dependencyFileName))`
	`  {`
	`  throw new ArgumentException(SR.ExceptionInvalidFileName,`
	`  "fullFileName");`
	`  }`
	
	`  this.lastModifiedTime =`
	`  File.GetLastWriteTime(fullFileName);`
	`}`
	
	`private void EnsureTargetFileAccessible()`
	`{`
	`  FileIOPermission permission = new`
	`  FileIOPermission(FileIOPermissionAccess.Read,`
	`  dependencyFileName);`
	`  permission.Demand();`
	`}`

Yes	Use the **EnvironmentPermission** class to restrict code that accesses environment variables. This is especially important if untrusted code can call the application block.
Yes	Declare environment permission requirements with declarative security attributes (use **SecurityAction.RequestMinimum**).

Code Access Security Checklist

Table 43 lists the general code review recommendations. An asterisk (*) next to an entry means that the analysis was performed with FxCop.

Table 43: Code Access Security Checklist

Check	Description

Yes	If a virtual method, property, or event with the **LinkDemand** security check overrides a base class method, the base class method must also have the same **LinkDemand** security check for the overridden method in order to be effective. For example, the **InstrumentationCategoryAttribute** class derives from the **System.Attribute** class that has a link demand. The following code example shows how the Logging Application Block uses the **LinkDemand** security check.
	`[System.Security.Permissions.PermissionSetAttribute`
	`(System.Security.Permissions.SecurityAction.LinkDemand,`
	`Name="FullTrust")]`
	`public sealed class InstrumentationCategoryAttribute : Attribute`
	`{`
	`private string name = null;`
	`private string description = null;`

Yes	*Members that call late-bound members should have declarative security checks. In the Enterprise Library, all provider types are late bound and use the **Activator.CreateInstance** method to create the specified type, which is in the configuration source. All of the application blocks' threat models document the need for trusted configuration files.
Yes	*Method-level declarative security should not mistakenly override class-level security checks.
Yes	None of the application blocks use the **AllowPartiallyTrustedCallerAttribute** (APTCA) attribute.

Check	Description

Yes	Do not expose methods protected by a **LinkDemand** security check. Some method calls in the .NET Framework are annotated with a **LinkDemand**. If they are called from within an application block's methods, the calling code is not checked for any security permissions. An example of this is the Logging Application Block's **CollectIntrinsicProperties** method. This method calls the **AppDomain.CurrentDomain** property, which issues a link demand to unmanaged code. Because of this, it is necessary to protect the **CollectIntrinsicProperties** method with the **SecurityPermissionFlag.UnmanagedCode** enumeration.
	`[SecurityPermission(SecurityAction.Demand, Flags = SecurityPermission-Flag.UnmanagedCode)]`
	`private void CollectIntrinsicProperties()`
	`{`
	`this.TimeStamp = DateTime.UtcNow;`
	`this.ActivityId = Trace.CorrelationManager.ActivityId;`
	`try`
	`{`
	`MachineName = Environment.MachineName;`
	`}`
	`catch (Exception e)`
	`{`
	`this.MachineName =`
	`String.Format(Properties.Resources.Culture,`
	`Properties.Resources.IntrinsicPropertyError, e.Message);`
	`}`
	`try`
	`{`
	`//AppDomain.CurentDomain issues a link demand to`
	`//unamanaged code. Link demands only check the immediate`
	`//caller. (In this case, this is the`
	`// CollectIntrinsicProperties method.)`
	`appDomainName = AppDomain.CurrentDomain.FriendlyName;`

Yes	None of the application block's methods should include **Assert** statements or **LinkDemand** security checks.

Design and Deployment Checklist

Table 44 lists the design and deployment recommendations.

Table 44: Design and Deployment Checklist

Check	Description
Yes	The design should address the scalability and performance criteria. Performance tests and stress tests demonstrate that the application block meets these criteria. The application block's availability and ability to handle concurrent users should also be tested.
Yes	Identify precautions that must be taken to satisfy the security requirements of the infrastructure and network (examples include operating system services, communication protocols, and firewalls). For example, the Logging Application Block should use a secure channel such as SSL or IPSEC, if it is logging sensitive data to a remote SQL Server or a remote message queue. The Caching Application Block does not encrypt data, so sensitive data logged to SQL store should be used over secured channel.
Yes	Application blocks do not save sensitive data in the registry or in text files during installation.
Yes	The application block respects the principle of least privilege. An application block does not need permissions from an administrator to run on ASP.NET, which requires only a network service account, or in Windows-based applications, which accepts any standard security context with the appropriate permissions to write to resources such as the event log and to use message queuing. The exact permissions depend on the application block.
Yes	Secure configuration stores with the appropriate ACLs.
No	Do not store sensitive information in plain text configuration files. An example of such information is a connection string that is used by the Data Access Application Block. Users should encrypt the configuration file. For more information, see Configuring the Application Blocks in the Enterprise Library documentation.
Yes	The design identifies application trust boundaries.
Yes	The design identifies the identities that are used to access resources across the trust boundaries.
Yes	The design identifies service account requirements.
Yes	The design identifies the mechanisms, such as SSL, IPSec, and encryption, to protect credentials when they are sent over a network.
Yes	If SQL authentication is used, credentials are adequately secured over the network (with SSL or IPSec) and in storage (with DPAPI).
Yes	The application blocks do not change the ACLs of the registry or of any files during installation or run time.
Yes	The application blocks do not listen to unknown ports except for their internal use. An example of where this is acceptable is when the Logging Application Block uses the **MSMQ Trace Listener**.

Additional Resources

There are many additional resources on MSDN to help you make your applications more secure. Here are some of them:

- For information about coding guidelines, see *Security Guidelines: .NET Framework 2.0*.
- For information about designing class libraries, see *Design Guidelines for Class Library Developers*.
- For information on building secure ASP.NET applications, see *Building Secure ASP.NET Applications*.
- For an index to a number of different checklist templates, see *Security Checklists Index*.

For information about FxCop security rules, see *FxCop Security Rules* on GotDotNet.

Testing for Globalization Best Practices

Globalization best practices ensure that the application code can support multiple cultures with little or no change to the code base. A culture is a combination of a language and a cultural environment. It includes information such as the format of dates and times, currencies, character classification, and sorting rules for strings. One globalization best practice is to separate the culture-dependent content from the culture-independent content. Another best practice is to write code that can process any culture-dependent content. This ensures that the application can be easily localized for a particular culture.

Tests for globalization best practices detect problems in the application's design that violate these practices. The tests should prove that the code can handle any cultural convention without incurring any difficulties that would either cause data loss or display problems.

This chapter discusses how the Enterprise Library application blocks were tested to see whether they followed globalization best practices.

The Test Approach

The test approach is the overall procedure that the test teams followed to ensure that the application blocks adhered to globalization best practices.

► **To test the application blocks**

1. Create a test plan that consists of test cases that show whether the application blocks follow globalization best practices.
2. Design pseudo-localization tests to find localizability bugs.
3. Choose a test environment by deciding on the operating system and the culture of the server that hosts the application blocks.
4. Execute the test cases and pseudo-localization tests and analyze the results to ensure that the application blocks do not lose the data or incorrectly display the data.

A tool that may help you prepare your code for globalization testing is Strgen. Strgen is a multilingual text generator tool that generates multi-lingual strings. The tool can be downloaded from *Multilingual Text Generator – STRGEN* on Microsoft.com.

Note: Strgen is distributed "as-is," with no obligations or technical support from Microsoft Corporation.

Creating a Test Plan

The test plan is made up of the test cases that determine whether the application blocks adhere to globalization best practices. Table 1 lists the test cases for the Enterprise Library application blocks.

Table 1: Test Plan for Enterprise Library Application Blocks

Test case	Example

Verify that the application block converts strings to Unicode characters from the managed format (Unicode) used within the application blocks.	In the following example, the platform uses the .NET Framework **CharSet.Unicode** attribute to marshal strings as Unicode characters.
	`[DllImport("kernel32.dll", CharSet =CharSet.Unicode)]`
	`public static extern IntPtr GetModuleHandle(string moduleName);`

Verify that any text that the application block sends to external resources, such as a file, a database, a message queue, and e-mail, are Unicode characters with UTF-8 encoding.	In the following example, messages that the application block sends in e-mail are set to UTF-8 encoding.
	`MailMessage message = new MailMessage();`
	`message.BodyEncoding = Encoding.UTF8;`

Verify that application blocks that use public APIs can accept culture-specific information such as addresses, currency, dates, and numerals. In addition, verify that the output is displayed in the appropriate, culture-specific format.	In the following example, the application block first converts a date and time that are in local time to universal time before processing them. It then converts them back to local time before displaying them.
	`// Converts the local time to universal`
	`// time before processing it.`
	`System.DateTime univDateTime = localTime.ToUniversalTime();`
	
	`// Do the processing using universal time.`
	`// Convert the universal time back to local`
	`// time to display it.`
	`System localTime = univTime.ToLocalTime();`

Test case	Example
Verify that text messages are not dynamically created at run time by concatenating multiple strings.	Concatenating multiple strings is not a globalization best practice. For example, assume the following: String 1 is "one after the other." String 2 is "The controls will be deleted" String 3 is "The forms will be deleted" If your code concatenates string 2 and string 1 or string 3 and string 1, the resulting English sentence is comprehensible. However, when the same strings are converted to a different language, such as German, the concatenation does not form a complete sentence.
Verify that all the localizable strings that the application block returns to the user, such as exception messages, are stored in external resource files.	In the following example, the application block validates the **writer** instance. If it is null, the application block throws an **ArgumentNullException** exception. The first parameter of the **ArgumentNullException** exception is the instance name that is null. The second parameter is the exception message. The application block uses the **Resources** class to retrieve the message from the resource file that stores strings for each culture. The application block retrieves the appropriate string based on the current culture setting. `if (writer == null) throw new ArgumentNullException("writer", Resources.ExceptionWriterShouldNotBeNull);` (Visual Studio generates the internal **Resources** class for every project.)
Verify that strings that the application block uses internally, such as the GUID and keys for collections, are not stored in external resource files. Because these strings are internal to the application block, they do not belong in resource files that store localized strings.	In the following example, the **CallContextSlotName** field "**EntlibLoggerContextItems**" is the key that retrieves the collection that stores additional context information that belongs to the calling application. The Logging Application Block uses this key name internally to retrieve information. `public const string CallContextSlotName = "EntLibLoggerContextItems";`

Verify that the application block uses the .NET Framework **DateTime.UtcNow** property to log the date and time to an external store such as a database, Windows Management Instrumentation (WMI), e-mail, or a message queue.	In the following example, the timestamp is logged in UTC format to WMI.
	`[InstrumentationClass(InstrumentationType.Event)]`
	`public abstract class BaseWmiEvent`
	`{`
	`private DateTime utcTimeStamp = DateTime.UtcNow;`
	`public DateTime UtcTimeStamp`
	`{`
	`get { return utcTimeStamp; }`
	`}`
	`}`

continued

Test case	Example

Verify that the application block handles strings as entire strings instead of as a series of characters. This is especially important when sorting or searching for substrings. When the application block must parse individual characters, verify that the application block uses the .NET Framework **StringInfo** class to parse the individual text elements.	The **StringInfo** class contains methods that retrieve individual text elements from a given string. A text element can be a base character, a surrogate pair, or a combined character sequence. In the following example, the application block retrieves individual text elements from a string and displays them on the console.
	`TextElementEnumerator charEnum = StringInfo.GetTextElementEnumerator(s);`
	`while (charEnum.MoveNext())`
	`{`
	`  Console.WriteLine(charEnum.GetTextElement());`
	`Console.WriteLine(Environment.NewLine);`
	`}`

Verify that strings that the application block stores internally (the end user cannot see them) are stored in a culture-independent format.	By storing strings in a culture-independent format, you can ensure that processes that require culture-independent results behave consistently, regardless of the actual culture. Use the .NET Framework **CultureInfo.InvariantCulture** property with strings that are culture-independent. This is shown in the following example.
	`CultureInfo Invc = CultureInfo.InvariantCulture;`
	`Thread.CurrentThread.CurrentCulture = Invc`

Verify that the application block's buffers are large enough to hold translated strings. Translated strings are often longer than the original strings. For example, when you translate strings from English to German, the German strings are longer than the English strings.	In the following example, the application block reads an array of 10 characters from a file. This may result in lost data if the number of characters in the file for a local language is larger than the number of characters used in English.
	`char[] buffer = new char[10];`
	`using (FileStream fs = new FileStream("", FileMode.Open))`
	`{`
	`using (StreamReader sr = new StreamReader(fs))`
	`{`
	`sr.Read(buffer, 0, 10);`
	`}`
	`}`

Verify that the application block uses culture-independent operations to compare strings such as file names, persistence formats, or symbolic information that the end user does not see.	If an application block uses a comparison operation to determine if a string is a recognized XML tag, that comparison should be culture-independent. If the application block bases a security decision on the result of a string comparison or case change operation, the operation should be culture-independent to ensure that the result is not affected by the value of the **CultureInfo.CurrentCulture** property. To make the comparison culture independent, use the overload of the .NET Framework **String.Compare** method that takes a **CultureInfo** object as a parameter. Pass in the **CultureInfo.InvariantCulture** property, This is shown in the following example. `int compareResult = String.Compare(string1, string2, false, CultureInfo.InvariantCulture);`

Test case	Example

Verify that the collection classes use culture-independent operations to sort the keys.	For example, the .NET Framework **SortedList** class represents a collection of key/value pairs that are sorted by the keys. A **SortedList** element can be accessed by its key or by its index. When you use a **SortedList** object whose keys are strings, the sorting and lookup operations can be affected by the **Thread.CurrentCulture** property. To obtain culture-independent behavior from a **SortedList** object, create it with one of the constructors that accepts an **IComparer** interface as a parameter. This parameter specifies the **IComparer** implementation to use when comparing keys. For the **IComparer** parameter, specify a custom comparer class that uses the **CultureInfo.InvariantCulture** property to compare keys. The following example illustrates a custom culture-insensitive implementation of the **IComparer** interface that you can specify as the **IComparer** parameter to a **SortedList** constructor.
	`internal class InvariantComparer : IComparer`
	`{`
	`  private CompareInfo m_compareInfo;`
	`  internal static readonly InvariantComparer Default = new`
	`  InvariantComparer();`
	
	`  internal InvariantComparer()`
	`  {`
	`  m_compareInfo = CultureInfo.Invariant-Culture.CompareInfo;`
	`  }`
	
	`  public int Compare(Object a, Object b)`
	`  {`
	`  String sa = a as String;`
	`  String sb = b as String;`
	`  return m_compareInfo.Compare(sa, sb);`
	`  }`
	`}`

Pseudo-Localization Testing

Pseudo-localization may be the most effective way of finding localizability bugs. This technique involves translating the application block's localizable resources into something readable but drastically different from normal text. For example, you could replace every "a" with an "â". Pseudo-localization also adds extra padding characters to the ends of strings. The types of errors that pseudo-localization helps you find are hard-coded strings that need internationalization, strings that should not be translated, non-Latin characters, and errors handling longer language strings. A pseudo-localized version of an application block should behave the same as the original version.

To use pseudo-localization, alter the application block's localizable resources in the following ways.

- Replace English text with text that contains non-English characters.
- Add extra characters to the resource strings. This ensures that the application block functions even if the translated text is longer than the English text.
- Add markers before and after the non-English strings. This test ensures that the application block displays the complete string and does not lose data. For example, you can enclose all strings within "[]" so that you can see where each string begins and ends.
- Use multi-lingual Unicode for all the substitutions and additions. This will help you find places where the application block uses ANSI functions to process or display text.

Creating the Test Environment

When you create a test environment, it is important to select the correct operating system. Windows XP is often a good choice for globalization testing because it supports a broad range of cultures. This means you can simulate many different regions on a single computer. Use the local build of Windows XP with a language group installed. For example, if you use the U.S. build of Windows XP, you could install the East Asian language group. Because German is installed by default, this combination gives you good test coverage without imposing requirements on the testers' language skills.

Another option is to use the local build of the target operating system. If the applications are meant for specific regions, the versions of the operating system that are tailored for these regions are the obvious choices for the builds. If the target operating system is Windows 2000 or Windows XP, you can use one language for the system UI and another for the actual application. By using this configuration, you can see how the application interacts with a localized system, where the names of the system folders, built-in accounts, fonts, and other system objects might be different from how they are represented in an English or Multilanguage User Interface (MUI) system. You can choose to use whatever language for the system UI that the testers understand.

The Enterprise Library application blocks do not have specific requirements for the cultures that they must support. To see if the application blocks could support diverse cultures, they were tested with a minimum of two language groups that belonged to linguistically unrelated regions, such as Japan and Germany.

Execute and Analyze the Results

Execute the tests that comprise the test plan in addition to the pseudo- localization tests. Here are some of the most common problems that may occur:

- Special characters, such as question marks, ANSI characters, vertical bars, boxes, and tildes, appear randomly on the display.
- The application block returns data, such as dates, times, and currency, that is incorrectly formatted.
- The error messages, or other hard-coded strings, are not in accordance with the current culture setting.
- The application block displays incomplete messages or strings (in other words, the application block loses data).

Usually, a simple code review reveals mistakes such as hard-coded strings, misuse of an overloaded method that has culture or culture-related parameters, or an incorrectly set culture-related property for the thread in which a call is executed. Here are some specific issues.

- If special characters appear on the display, there may be a problem with the Unicode-to-ANSI conversion process. For example, a question mark may mean that the Unicode-to-ANSI conversion is not using the correct conversion tables, which are determined by the **CultureInfo.LCID** property. This property gets the current culture identifier. If you are trying to convert Japanese Unicode strings to ANSI on an English system, and if you do not explicitly specify the code page to use, the system will use the default code page, which does not contain information on how to convert the Japanese Unicode strings.
- Incorrectly formatted data may indicate that the application block has methods that do not use the current culture setting when they retrieve information.
- If the application block displays incomplete messages or strings, the length of the message or string may be too large in a non-English language for the application block's buffers.

For a complete list of globalization-related issues, see *Globalization and Localization Issues* on MSDN.

Testing for Performance and Scalability

Testing the Enterprise Library application blocks for performance and scalability involves testing the library's set of application blocks under both normal and peak loads. The tests are performed without incorporating the application blocks into a full-fledged application. The goals of the performance tests are:

- **To verify that the application blocks meet the performance requirements while staying within the budgeted constraints on system resources**. The performance requirements can include different measurements, such as the time it takes to complete a particular scenario (this is known as the response time) or the number of concurrent or simultaneous requests that can be satisfied for a particular operation within a given response time. Examples of system resources are CPU time, memory, disk I/O, and network I/O.
- **To analyze the behavior of the application blocks at various load levels**. Performance tests that measure this behavior use metrics that relate to the performance objectives and other metrics that help to identify the bottlenecks in the application blocks. A bottleneck is the device or resource that constrains throughput. Most of the time, performance bottlenecks in your application relate to resource issues that may include server resources, such as CPU, memory, disk I/O, and network I/O or other external resources, such as available database connections or network bandwidth. Bottlenecks can be caused by various issues, such as memory leaks, slow response times, and contention for resources while under load.

Performance tests for the application blocks fall into two broad categories:

- **Load tests**. Load tests monitor and analyze the behavior of an application block under both normal and peak load conditions. Load tests enable you to verify that the application block meets the performance objectives.
- **Stress tests**. Stress tests identify problems that occur both when the application block operates under heavy load conditions and when it must operate under these conditions for lengthy periods of time. The application block may fail under these conditions because it has depleted some system resources. Another goal of stress testing is to see if the application block can recover when its load exceeds the specified limits and then returns to normal. In short, when you run stress tests, monitor the application block to see if its performance degrades under heavy loads and to see if its performance recovers after the load returns to normal.

Scalability tests are extensions of performance tests. The outstanding characteristic of a scalable application is that it only requires additional resources to operate under additional loads instead of requiring extensive modifications to its code. The goals of scalability testing are:

- To determine any performance gains the application block achieves with additional system resources, such as CPUs or computers, or any additional external resources, such as a SQL Server database or a disk array.
- To identify any locking and contention problems that may not be detected by performance tests.

Although performance affects how you determine the number of users that an application can support, scalability and performance are two separate requirements. Performance optimizations may reduce an application's scalability and scalability optimizations may reduce an application's performance.

This chapter contains the following sections:

- Defining Performance Criteria. This section defines the performance criteria that are used to test the application blocks. The performance criteria describe the performance goals for the Enterprise Library in terms of system metrics and transaction times. For example, a performance goal may be to have less than 10 locks per second. Another goal may be to have transaction times that are less than 250 milliseconds (ms).
- Setting Up the Test Environment. This section describes the Enterprise Library test environment.
- Building Test Harnesses. This section describes how to create a test project, Web tests, test harness script, and a load test.
- Testing the Application Blocks. This section describes how each application block was tested.
- Detecting Performance Issues. This section describes the performance parameters you should monitor to detect performance issues in the application blocks.
- Measuring Performance. The section describes the performance metrics for the application blocks. Examples include the amount of system resources each application block uses and the transaction times for API calls. It also describes performance counters you can use to measure performance.
- Testing for Scalability. This section describes the objectives of the scalability tests and analyzes the results for several application blocks.
- Measuring Initialization Costs. This section describes how to measure the amount of system resources and time necessary to prepare an application block so that it can begin to execute requests.
- Extrapolating Workload Profiles. This section describes how to use Little's Law to characterize workloads.
- Debugging Memory Leaks. This section explains how to use the WinDbg debugger to detect memory leaks.

Defining Performance Criteria

The most common approach to determining an application's performance is to measure its response times and the resources it uses against criteria, such as the application's budgets for resources, its business needs, and its service level agreements. However, it is not always possible to use these criteria with Enterprise Library, treated as a whole. The reasons for this are:

- There is no single application that encompasses all the application blocks.
- Different applications require different resources.
- Different applications have different goals.

Because the application blocks cannot be tested in the context of a specific application, another approach is to measure an application block's performance goals against a baseline. This is the approach used with Enterprise Library. The baselines are the .NET Framework and Enterprise Library 1.1. In this case, an example of a performance goal may be that an Enterprise Library method should have less than a 15 percent overhead when compared to the overhead of the equivalent .NET Framework method.

For example, the cost of adding an item to the ASP.NET cache serves as a baseline measurement for the Caching Application Block's equivalent **Add** method. By comparing the two costs, the overhead of the Enterprise Library application block becomes apparent.

Measurements of the performance criteria are collected with the Windows Performance Monitor (Perfmon.exe). Here is the complete list of performance criteria for Enterprise Library–January 2006:

- Overhead cost
- Initialization cost
- Consistency
- Availability

The next sections discuss these subjects.

Overhead Cost

The overhead cost is expressed as a percentage. It is the difference between the response time of an application block (or one of its system resources) when it performs a function and the response time of the .NET Framework and Enterprise Library 1.1 when it performs the same function. The overhead cost is expressed as a percentage of total transactions (this is also termed "total hits" for Web applications) and as a percentage of transactions per second. The next sections describe the calculations for determining the overhead costs.

Total Transactions

This section describes how to calculate the overhead cost in terms of percentage of total transactions. For Web applications, the term "total hits" is often used instead of "total transactions."

Overhead cost = ((Total transactions for .NET – Total transactions for application block)/Total transactions for .NET) * 100

Table 1 lists the data that was collected during a 7 minute performance test. The goal was to compare the cost of using the Caching Application Block in Enterprise Library–January 2006 to add an item to the cache against the ASP.NET and Enterprise Library 1.1 baselines.

Table 1: Overhead of Using Caching Application Block Add Method

Component	TPS	Time (sec)	Overhead	Total transactions
ASP.NET	1844.215	0.048801	baseline	608591
Enterprise Library 1.1 (instrumentation off)	1692.394	0.053179	8.23%	558490
Enterprise Library 1.1 (instrumentation on)	1626.352	0.055339	11.81%	536696
Enterprise Library – January 2006 (instrumentation off)	1725.536	0.052158	6.44%	569427
Enterprise Library – January 2006 (instrumentation on)	1611.703	0.055842	12.61%	531862

The overhead of using the Enterprise Library–January 2006 Caching Application Block is calculated as follows:

Total Transactions for ASP.NET = 608591

Total Transactions for Enterprise Library–January 2006 (instrumentation on) = 531862

Overhead = ((608591-531862)/608591*100) =12.61%

Transactions per Second

This section describes how to calculate the overhead cost in terms of percentage of transactions per second (TPS).

Overhead cost = ((TPS for .NET – TPS for application block)/TPS for .NET) * 100

The following is an example that uses the data in Table 1:

Transactions Per second for ASP.NET = 1844.21

Transactions Per Second for Enterprise Library – January 2006(instrumentation on) = 1611.7

Overhead = ((1844.21-1611.70)/1844.21)*100) = 12.60%

Initialization Cost

The initialization cost is the cost of preparing an application block so that it can begin to execute requests. This measurement is not relevant for Web applications because the first hit (this is the initialization cost) is not considered an important factor in terms of performance. However, the measurement does matter for smart client applications because users frequently stop and restart the application.

Consistency

The measurements for response times and use of system resources should be stable and consistent during performance tests and stress tests. This means that transactions should always return the expected results and there should be no interruptions of service. In addition, there should be no spikes in the use of system resources, such as memory or processors. Transactions that return data should show repeatable results.

Availability

The application block and the computer it runs on should be available. This means that if there are failures while the application block is running, they should be resolved before there are problems, such as service errors or corrupted data. For example, there should be no interruptions of service during the stress tests.

Setting Up the Test Environment

The test environment setup for Enterprise Library requires:

- A host engine for the application blocks.
- A network of computers to act as load agents.

Choosing the Host Engine

ASP.NET hosts the application blocks for the Enterprise Library tests. It has the following advantages:

- ASP.NET is well-documented and has readily available tools to measure performance and scalability.
- ASP.NET allows you to place the client and the server on separate computers.

Setting up the Test Environment

After you decide the host engine to use, you can set up the test environment. The test environment is where you run your test harnesses and make your performance measurements. Enterprise Library is tested with Visual Studio 2005 Team Edition for Software Testers. The test environment consists of the following components:

- 3 agents to run the load tests
- 1 controller to administer the agents and collect test results
- 1 computer with Visual Studio 2005 to develop the test code
- 1 Web server to host Enterprise Library
- 1 SQL Server database

The tests use dedicated client computers as load generators (these are also referred to as load agents) to ensure that there is no competition for system resources. In addition, it is good practice to use isolated networks with dedicated switches for the performance lab benches. This prevents any unrelated issues from contaminating the test results.

Load tests should use all of an agent's CPU cycles by simulating concurrent virtual users. This is because the Web server that hosts Enterprise Library is not used to its full capacity. The greater the capacity of the Web server, the greater the load the agents must put on the system to counteract the Web server's effects. Otherwise, the Web server may distort the measurements so the system's response times appear to be better than they actually are. You may have to add multiple load agents to accomplish this. The Enterprise Library test environment uses three load agents.

Figure 1 shows the Enterprise Library test environment.

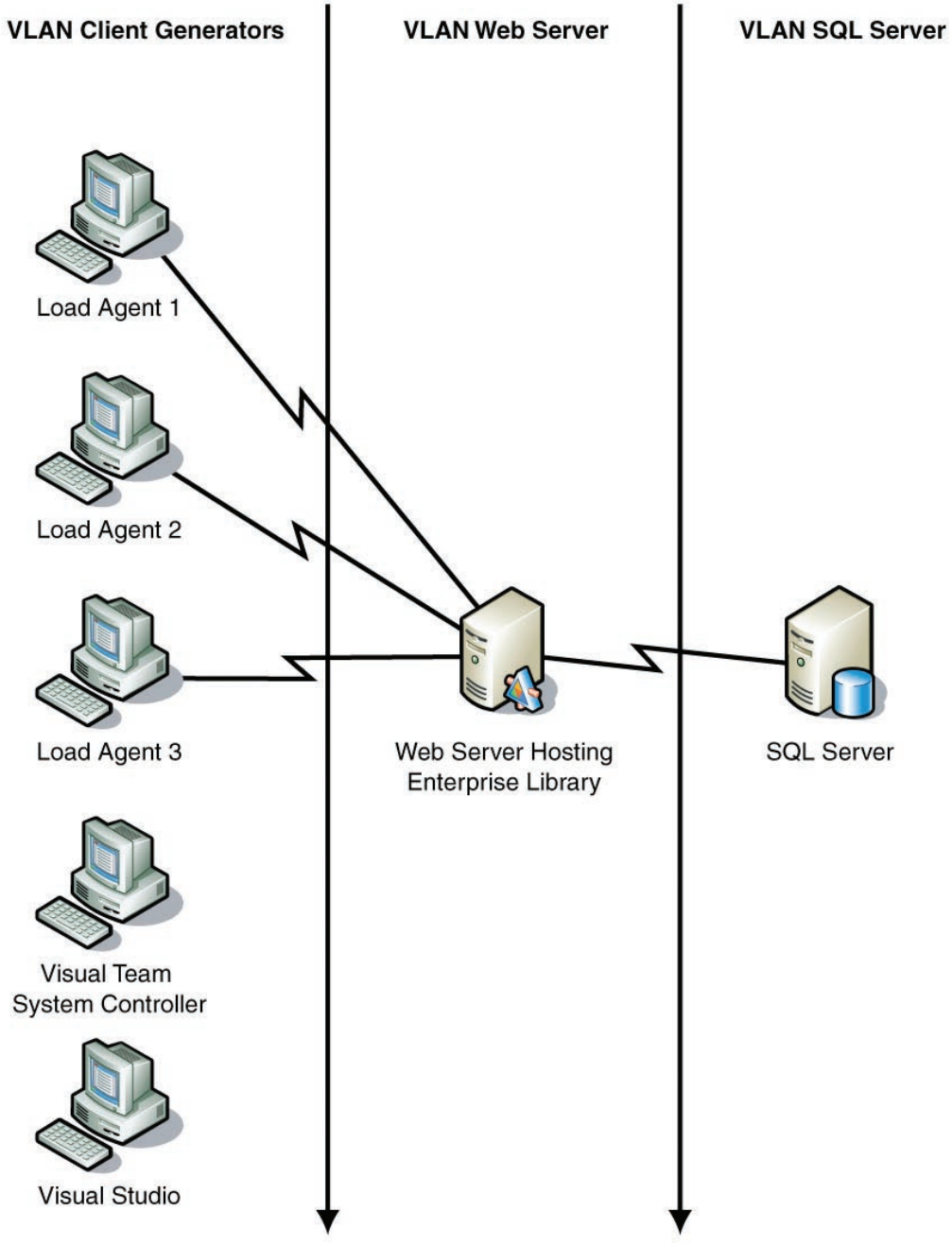

Figure 1 *Enterprise Library test environment*

After you set up the test environment, you should tune it to make sure that there are no inherent problems that may distort the measurements.

Tuning the Test Environment

To ensure that the test measurements are accurate, you should first make sure that the test environment is performing well and has no constraints or performance issues that could contaminate your measurements. The areas you should monitor are identical to those for the application blocks. For information, see Detecting Performance Issues.

Building Test Harnesses

A test harness is a Web application that loads the tests and runs them. The Enterprise Library test harnesses use Web controls on a Web page to configure and run the tests. Figure 2 illustrates the Caching Application Block test harness.

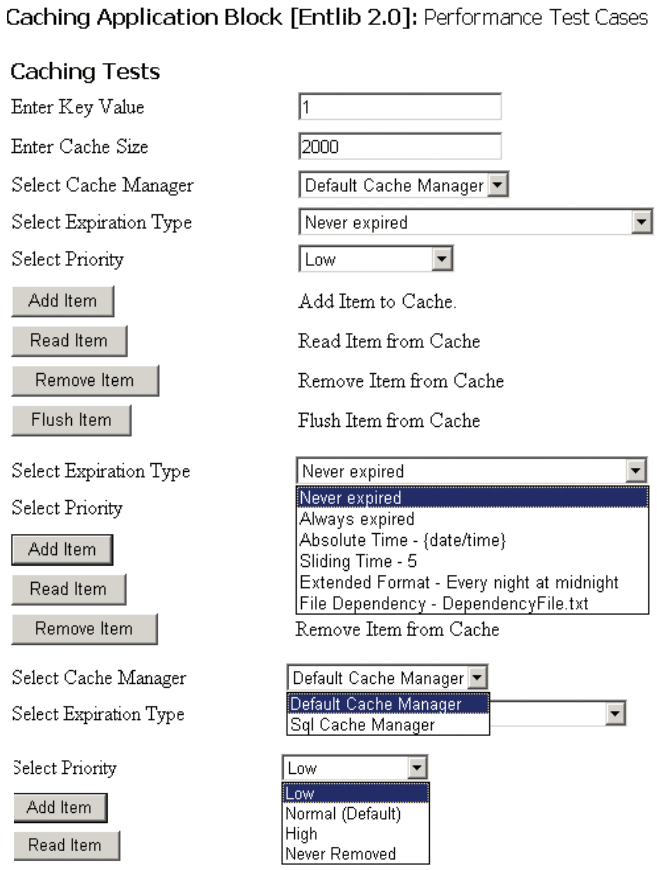

Figure 2 *Caching Application Block test harness*

To create a test harness, you need a test project and a Web test script in addition to the Web page.

Creating a Web Test Script

A Web test script simulates how an application might interact with an application block. Typically, you create a test script by recording HTTP requests using the Web Test Recorder in a browser session.

The following procedures generate a test script in Visual Studio 2005 Team Edition for Software Testers.

► To create a test project

1. Open Visual Studio Team Edition for Testers
2. Create a test project. Click **File**, point to **New** and click **Visual C# Test**.
3. Type a name and click **OK**.

After you have created a test project you can record a Web test.

► To record a Web test

1. If the test project is not open, open it. On the **File** menu, point to **New**, and then click the test project.
2. On the **Test** menu, click **New Test**. The **Add New Test** dialog box appears.
3. Click **Web Test**.
4. In the **Test Name** box, type a name. Do not use the .webtest extension. Click **OK**. The **Web Test Recorder** opens inside a new instance of Internet Explorer.
5. Enter the URL of the Web site you want to test.
6. Execute the actions the test will simulate by clicking the appropriate controls, such as buttons and drop-down list boxes.
7. Click **Stop** to stop recording.
8. On the **File** menu, click **Save** to save the test.

Repeat this procedure for each test that is contained in the test harness.

The tree of nodes that contains URLs in the Web Test Editor is named the *request tree*. You can select nodes in the request tree to view the properties associated with each request after the test is recorded. Figure 3 is an example of a request tree.

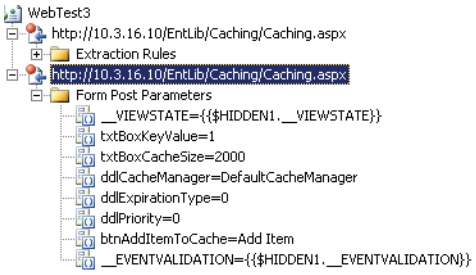

Figure 3 *Request tree*

After you create the test, you can convert the recorded Web test to a coded Web test.

► **To convert a recorded Web test**

1. Open a solution that contains a recorded Web test.
2. Open the Web test file.
3. In the Web Test Editor, click the **Generate Code** button. You are prompted for a name for the coded Web test.
4. Type a name, and then click **OK**.
5. Click **Build**, and then click **Build Solution**. Your code compiles. The compiled code is a test script.

When you build your test harnesses, remember two points:

- Do not cache an application block's internal domain objects over iterations of the tests to avoid the costs of creating them. Performance tests should assume the worst case scenario, which is the most expensive code path. For example, the Caching Application Block's most expensive code path is when the application block creates a **CacheManager** object. The most expensive path for the Logging Application Block is when the application block creates a **LogEntry** object. For an example of how to create a **CacheManager** object, see Using the Test Script.
- Do not distort the TPS measurements made with the performance monitor (ASP.NET: Requests/sec) and the Visual Studio 2005 Team Edition for Software Testers load test tool by reloading the Web page after the first request. Instead, read the **ViewState** and **EventValidation** variables (and any other static variables that the Web server returns).

There are two ways to implement these suggestions. You can either edit the test script or you can use data binding. For information about editing the test script, see Using the Test Script. For information about using data binding, see Using Data Binding.

Using the Test Script

You can add code to the test script to maintain the state of the Web page. This means that the entire page is loaded only once, when the first user accesses it. The values from the page and all of the controls are collected and formatted into a single encoded string and then saved in the **_VIEWSTATE** hidden field. In the example given here, the test script stores this string in the **ViewState** variable. The test script tests the **ViewState** variable to see if it is null. This is the equivalent of seeing if this is the first time the page has been loaded. If the value is not null, the test script reloads the page but executes only the **POST** request path, along with the required test actions.

The test script also uses the **__EVENTVALIDATION** hidden field. This field acts as security against fraudulent postbacks. It validates any events (such as clicking a button) that occur. The **EventValidation** variable stores the value of the field.

The test script also demonstrates how you can define test parameters such as the type of cache storage and the priority of a cached item by including them in a drop-down list box. The test script selects the correct parameters at run time and no user action is necessary. For example, the following line in the script selects the **CacheManager** type object, which is named **DefaultCacheManager**. This object is created for each test iteration.

```
request2Body.FormPostParameters.Add("ddlCacheManager","DefaultCacheManager");
```

The **CacheManager** object is created for each test iteration.

The following test script illustrates how to save the state of the Web page and how to create objects for each test iteration. Although this test script is for the Caching Application Block, the same script can be used with the other application blocks, except for the section that generates unique keys for cached items. For more information about this section of the test script, see Testing the Caching Application Block.

```
// Test script
// This script, except for the code that generates the unique keys,
// can be used with all application blocks. This code is specific to the
// Caching Application Block.
// Variables hold the ViewState and EventValidation hidden fields.
// Variables hold the KeyValue and lockn objects to generate the key item
// for the Caching Application Block.

      public class CachingBlockTestCoded : WebTest
       {
private static string ViewState;
private static string EventValidation;
private static int KeyValue=0;
private static object lockn = new object();
private const int MaxElements=2000
public Caching_Entlib20()
{
    this.PreAuthenticate = true;
}

public override IEnumerator<WebTestRequest> GetRequestEnumerator()
{
     // If it is the first hit, the value
     // is cached and used
     // on subsequent hits.
     if (ViewState == null)
     {
WebTestRequest request1 = new WebTestRequest("http://10.3.16.10/EntLib/Caching/
Caching.aspx");
ExtractHiddenFields rule1 = new ExtractHiddenFields();
            rule1.ContextParameterName = "1";
request1.ExtractValues += new EventHandler<ExtractionEventArgs>(rule1.Extract);
            yield return request1;
// The ViewState and EventValidation variables are extracted from
// the context of a Web test object.
```

```
viewState = this.Context["$HIDDEN1.__VIEWSTATE"].ToString();
EventValidation = this.Context["$HIDDEN1.__EVENTVALIDATION"].ToString();
      }

      String KeyValueString;
      // The object is locked and the keyvalue is incremented.
      // It is stored in a local variable
      // and used in the post statement.
      // This is only applicable to the Caching Application Block.
      lock (lockn)
      {
           KeyValueString = KeyValue.ToString();
           KeyValue++;
           if (KeyValue > MaxElements)
               KeyValue = 1;
      }
      // A timer is set to measure transaction times.

this.BeginTransaction("Caching_Entlib20");
WebTestRequest request2 = new WebTestRequest("http://10.3.16.10/EntLib/Caching/
Caching.aspx");
      request2.Method = "POST";
      FormPostHttpBody request2Body = new FormPostHttpBody();
      // Add the ViewState hidden filed to the body of the request.
request2Body.FormPostParameters.Add("__VIEWSTATE", ViewState);
request2Body.FormPostParameters.Add("txtBoxKeyValue", KeyValue);
request2Body.FormPostParameters.Add("txtBoxCacheSize", "2000");
request2Body.FormPostParameters.Add("ddlCacheManager","DefaultCacheManager");
request2Body.FormPostParameters.Add("ddlExpirationType", "0");
    request2Body.FormPostParameters.Add("ddlPriority", "0");
request2Body.FormPostParameters.Add("btnAddItemToCache", "Add Item");
// Add the EventValidation hidden field to the body of the request.
request2Body.FormPostParameters.Add("__EVENTVALIDATION", EventValidation);
      request2.Body = request2Body;
      yield return request2;
      this.EndTransaction("Caching_Entlib20");
}
```

Using Data Binding

With data binding, you can use a data source to provide data for a Web test. You can bind data from the data source to a part of a Web request that requires data, such as a form post parameter. For example, in the Caching Application Block Web test, you can set the value of the cache storage (this can be either **SqlCacheManager** or **DefaultCacheManager**) in the data source and then pass this value to the Web test at run time. The **Cache Manager** drop-down list will show the correct cache manager as selected. You can use any OLE DB data source for data binding, including .csv files, Microsoft Excel, Access and SQL Server databases.

To use data binding, first create a database table or a .csv file that contains the data that will be bound to the Web control. The first row in the column is a name and the subsequent rows are the values. For example, to data bind values to the drop-down

list that displays the cache manager, you could enter the name **Cache** in the first row, the value **DefaultCacheManager** in the second row, and the value **SqlCacheManager** in the third row, as shown here:

Cache

DefaultCacheManager

SqlCacheManager

After you create the data source, you can bind it to the correct form post parameter.

► **To data bind a form post parameter**

1. In Visual Studio, click the Web test. The Web test opens in the Web Test Editor.
2. In the Web Test Editor, right-click the top node of your Web test, and then click **Add Data Source**.
3. In the **OLE DB Provider** drop-down list, select your data provider.
4. In the Web test request tree, expand the **Form Post Parameters** node. Click the Web control that will be bound to the database or .csv file.
5. In the **Properties** window, click the **Value** property, and then click the down arrow that appears.
6. Click **DataSource1**. Expand the node that is named after the database table or .csv file. Click the name of the column that contains the values for the Web control. The Web control is now data bound. You should see this if you look at the Web control in the Web test request tree.

After the Web control is data bound, click the **Generate Code** button on the toolbar. This generates a new test script that includes the data binding. The following code is an example of a test script that uses data binding.

```
Test code with the data binding information. Note that the post contains the call
to request2Body.FormPostParameters.Add("TextBox1", this.Context["DataSource1.
cache#csv.cacheitem"].ToString());
  [DataSource("DataSource1", "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=c:\\
pag\\entlibcode;Extended Propertie" +
        "s=text", Microsoft.VisualStudio.TestTools.WebTesting.DataBindingAccess-
Method.Sequential, "cache#csv")]
    [DataBinding("DataSource1", "cache#csv", "cacheitem", "DataSource1.cache#csv.
cacheitem")]
    public class WebTest2Coded : WebTest
    {

        public WebTest2Coded()
        {
            this.PreAuthenticate = true;
        }

        public override IEnumerator<WebTestRequest> GetRequestEnumerator()
        {
            WebTestRequest request1 = new WebTestRequest("http://localhost/
entlibtests/caching.aspx");
```

```
            request1.ThinkTime = 13;
            ExtractHiddenFields rule1 = new ExtractHiddenFields();
            rule1.ContextParameterName = "1";
            request1.ExtractValues += new EventHandler<ExtractionEventArgs>(rule1.
Extract);
            yield return request1;

            WebTestRequest request2 = new WebTestRequest("http://localhost/
entlibtests/caching.aspx");
            request2.Method = "POST";
            FormPostHttpBody request2Body = new FormPostHttpBody();
            request2Body.FormPostParameters.Add("__VIEWSTATE", this.
Context["$HIDDEN1.__VIEWSTATE"].ToString());
            request2Body.FormPostParameters.Add("TextBox1", this.
Context["DataSource1.cache#csv.cacheitem"].ToString());
            request2Body.FormPostParameters.Add("TextBox2", "2000");
            request2Body.FormPostParameters.Add("DropDownList1", "0");
            request2Body.FormPostParameters.Add("DropDownList3", "DefaultCacheMan-
ager");
            request2Body.FormPostParameters.Add("DropDownList2", "0");
            request2Body.FormPostParameters.Add("Button1", "Add Item");
            request2Body.FormPostParameters.Add("__EVENTVALIDATION", this.
Context["$HIDDEN1.__EVENTVALIDATION"].ToString());
            request2.Body = request2Body;
            yield return request2;
        }
    }
}
```

Each test iteration uses a different cache store. If you want to use only one of the cache stores, remove the other entry from the data source. Data binding is particularly useful for stress tests, where you want to test all of the scenarios. Bind every Web control to a data source to create a Web test with full coverage.

Defining the Workload Profile

The workload profile consists of an aggregate mix of users simultaneously performing various operations. This profile should yield the same results over multiple test runs. A workload profile consists of an aggregate mix of users simultaneously performing various operations. To create a workload profile, decide the number of users you will simulate, how the operations the users perform will be distributed in terms of percentages, and how much think time you will allow between operations. Think times simulate behavior that causes people to wait between interactions with a Web site. They should occur between requests in a Web test and between test iterations in a load test. Including think times in a load test can be useful in creating more accurate load simulations. In Visual Studio 2005 Team Edition for Software Testers, think times are measured in seconds. (The Enterprise Library tests do not use think times because they do not have to simulate that type of user behavior.)

Create a workload profile when you have to prove that an application block can concurrently execute a variety of scenarios. The Caching Application Block performance

tests had to verify that the application block could support concurrent users accessing the same cache but for different reasons. For example, some of these users would add items to the cache while other users would remove items. To see the workload profile for the Caching Application Block, see Profiling the Workload in Testing the Caching Application Block.

You will use the workload profile when you create the load tests. The settings in Creating a Load Test were used in the Enterprise Library load tests.

Creating a Load Test

After you create the Web tests, you can create the load test. The load test simulates many clients accessing the application block at the same time.

► **To create a load test**

1. Open the solution that contains the Web test.
2. In Solution Explorer, right-click the test project node. Click **Add**, and then click **Load Test**. The **Load Test Wizard** appears.
3. Click **Next**.
4. Enter the name of the scenario.
5. Set the **Think Time Profile** setting to **Use normal distribution centered on recorded think times**.
6. Set the **Think time between test iterations** setting. The Enterprise Library tests set this value to **0** because they did not use think times.
7. Click **Next**. The **Edit load pattern settings for a load test scenario** dialog box appears.
8. Select **Constant Load**. Different iterations of the Enterprise Library tests used different user counts. Each application block defines these counts in its workload profile.
9. Click **Next**. The **Add tests to a load test scenario and edit the test mix** dialog box appears.
10. Click **Add** to select tests. Click the tests in the **Available tests** pane that you want to use in the load test. After you select all tests that you want to use, click the arrow to add them to the **Selected tests** pane.
11. Click **OK**.
12. The text mix appears. You can use the sliders to adjust the test distribution. When you are finished, click **Next**.
13. The **Select browser mix for test scenario** dialog box appears. In the drop-down list, click **IE6**. Click **Next**.
14. The **Select network mix for test scenario** dialog box appears. Select the LAN connection type you will use. Set the sliders to full bandwidth. Click **Next**.

15. The **Specify computers to monitor with counter sets during load test run** dialog box appears. Click **Add Computer** to select the computers to monitor during the load tests.
16. Click **Counter sets**. Select the counters that interest you. For information about the counters used in the Enterprise Library tests, see Detecting Performance Issues and Measuring Performance. Click **Next**.
17. The **Review and edit run settings for a load test** dialog box appears. Set the **Timing** values. For the Enterprise Library tests, the sampling rate was 120 samples for any particular time frame. Stress tests ran from 12 hours to 72 hours. Performance tests ran for 5 minutes to 7 minutes. Warm-up times varied from 30 seconds for 7 minute performance tests to 5 minutes for tests longer than 2 hours.
18. Click **Finish**.

To run a load test, open the load test in the Load Test Editor, and then click the green **Run** button.

Testing the Application Blocks

This section provides the specifics of the performance tests for each of the application blocks. It includes the following information:

- The scenarios used to test the application block
- A description of the test harness
- The test code for the application block
- The test code for the .NET Framework baseline
- The workload
- A template to compare the application block metrics with the baseline metrics

The application blocks were tested both with the instrumentation enabled and with it disabled. Use the Enterprise Library Configuration Console to enable or disable the instrumentation. For more information, see the Enterprise Library documentation.

Note: Only *Testing the Caching Application Block* contains information about how the application block was configured. This is the only application block where some of the configuration parameters could affect the performance and stress tests.

Testing the Caching Application Block

To run the performance tests, the Caching Application Block was configured as follows:

- The expiration poll frequency was 60 seconds.
- The maximum number of elements in the cache before scavenging would begin was 900.

- The number of elements to remove during scavenging was 10.
- The cache storage was both in-memory and a SQL Server database.
- The cache size was 2, 000 KB.
- The number of elements in the cache was between 900 items and 2,000 items.

Table 2 lists the scenarios for the Caching Application Block. Here is an explanation of each of the columns:

- **ID**. This column lists the ID number for each scenario.
- **Scenario**. This column lists the Caching Application Block scenario being tested.
- **Users**. This column lists the different numbers of users that the tests simulated.
- **Expiration policy**. This column lists the different expiration policies that were used during the tests.

Table 2: Caching Application Block Scenarios

ID	Scenario	Users	Expiration policy
1	Add an item.	1, 10, 50, 150, 300	Never/Absolute/FileDependency/ Extended Format
2	Read an item.	1, 10, 50, 150, 300	Never/Absolute/FileDependency/ Extended Format
3	Remove an item.	1, 10, 50, 150, 300	Never/Absolute/FileDependency/ Extended Format
4	Add and remove an item.	1, 10, 50, 150, 300	Never/Absolute/FileDependency/ Extended Format
5	Add and remove and read an item.	1, 10, 50, 150, 300	Never/Absolute/FileDependency/ Extended Format
6	Add and remove and read and flush an item.	1, 10, 50, 150, 300	Never/Absolute/FileDependency/ Extended Format
7	Flush an item.	1, 5	

Creating a Test Harness

Each iteration of the test should create the **CacheManager** object so that the cost of creating the object is included in the performance metrics. In addition, every item in the cache requires a unique key. For more information, see Generating Unique Keys. For an example of how to create a test harness, see Building Test Harnesses. This section also contains an example of a Web page for the Caching Application Block test harness.

Generating Unique Keys

Every item that is added, read, or removed from the cache requires a unique key. Another way to think of this is that each thread, which represents a virtual user, requires its own key. The Caching Application Block test increments the key value until

it reaches the maximum number of elements in the cache. (This number varied from 900 to 2,000.) When the maximum number was reached, the test resets the count and begins again.

To generate keys, you can either modify the test script or use data binding.

Modifying the Test Script to Generate Keys

You can modify the test script to include code that generates a unique key value for each cache item. In this approach, a static integer named **KeyValue** stores the key value. The maximum value that can be generated is controlled by another variable named **MaxElements**. If the key is higher than **MaxElements**, it is reset to **1**. Because the script runs in multithreaded scenarios, the code locks the steps that generate the keys. This is shown in the following example.

```
private static int KeyValue=0;
private static object lockn = new object();
private const int MaxElements=2000

String KeyValueString;
lock (lockn)
{
  KeyValueString = KeyValue.ToString();
  KeyValue++;
  if (KeyValue > MaxElements)
   KeyValue = 1;
}
```

For more information, see Using the Test Script.

Using Data Binding to Generate Keys

Data binding allows you to use either a database or a .csv file to serve as the data source for the keys. The first row of the table or file should contain the name of the items that will go into the cache. For example, "Key" would be an appropriate name. The other rows contain the values for the keys. For more information, see Using Data Binding.

Creating the Test Code

The test code implements the scenarios. There are two versions of the test code shown here. One version is for Enterprise Library and the other version is for ASP.NET. Only the Add and Read scenarios were used to establish a baseline, so the .NET Framework code implements only those two cases. During the baseline tests, the Enterprise Library tests were performed both with and without scavenging. The code for Enterprise Library 1.1 differs from the code for Enterprise Library–January 2006, but it is not shown here.

Table 3 lists the Caching Application Block test code.

Table 3: Caching Application Block Test Code

ID	Scenario	Test code

1	Add	This is the code to add items to the cache.
		`protected void AddItem_Click(object sender, EventArgs e)`
		`{`
		`// The cache element is created. The size is defined`
		`// by the TextBox2.Text.`
		`byte[] CacheObject = new byte[int.Parse(TextBox2.Text)];`
		`// The GetCacheManager method is called for each iteration.`
		`// The CacheManager is not cached. The call to GetCacheManager`
		`// passes it the type of caching store from`
		`// DropDownList3. SqlCacheManager uses SQL Server as the`
		`cache.`
		`// DefaultCacheManager is an in-memory cache.`
		`this.primitivesCache = CacheFactory.GetCacheManager(DropDownLi`
		`st3.SelectedItem.Value);`
		`this.absolutetime = DateTime.Now + TimeSpan.FromSeconds(60);`
		`switch (this.DropDownList1.SelectedIndex)`
		`{`
		`case 0:`
		`primitivesCache.Add(TextBox1.Text, CacheObject, this.Priority,`
		`new NullRefreshAction(), null);`
		`break;`
		`case 1:`
		`primitivesCache.Add(TextBox1.Text, CacheObject, this.Priority,`
		`new NullRefreshAction(),`
		`new AbsoluteTime(this.absolutetime));`
		`break;`
		`case 2:`
		`primitivesCache.Add(TextBox1.Text, CacheObject, this.Priority,`
		`new NullRefreshAction(),`
		`new SlidingTime(TimeSpan.FromSeconds(5)));`
		`break;`
		
		`case 3:`
		`primitivesCache.Add(TextBox1.Text, CacheObject, this.Priority,`
		`new NullRefreshAction(),`
		`new ExtendedFormatTime("0 0 * * *"));`
		`break;`
		`case 4:`
		`primitivesCache.Add(TextBox1.Text, CacheObject, this.Priority,`
		`new NullRefreshAction(),`
		`new FileDependency(@"c:\pag\entlibtests\web30.config"));`
		`break;`
		`}`
		`}`

ID	Scenario	Test code

2	Read	This is the code to read an item from the cache.
		`protected void ReadItem_Click(object sender, EventArgs e)`
		`{`
		`this.primitivesCache= CacheFactory.GetCacheManager(DropDownLis t3.SelectedItem.Value);`
		`byte[]b= (byte[])this.primitivesCache. GetData(TextBox1.Text);`
		`}`

3	Remove	This is the code to remove an item from the cache.
		`protected void RemovingItem_Click(object sender, EventArgs e)`
		`{`
		`this.primitivesCache= CacheFactory.GetCacheManager(DropDownLis t3.SelectedItem.Value);`
		`primitivesCache.Remove(TextBox1.Text);`
		`}`

7	Flush	This is the code to flush the cache.
		`protected void FlusghingItem_Click(object sender, EventArgs e)`
		`{`
		`this.primitivesCache= CacheFactory.GetCacheManager(DropDownLis t3.SelectedItem.Value);`
		`primitivesCache.Flush();`
		`}`

Table 4 lists the ASP.NET test code.

Table 4: ASP.NET Framework Test Code

ID	Scenario	Test code

1	Add	This is the code to add items to the cache.
		`protected void ASPNETadditem_Click(object sender, EventArgs e)`
		`{`
		`byte[] CacheObject = new byte[int.Parse(TextBox2.Text)];`
		`Cache[TextBox1.Text] = CacheObject`
		`}`

2	Read	This is the code to read items from the cache.
		`protected void ASPNETreaditem_Click(object sender, EventArgs e)`
		`{`
		`byte[] CacheObject= Cache[TextBox1.Text];`
		`}`

Profiling the Workload

The number of users increases over time. The number of users was 1, 10, 50, 150, and 300. There was no think time. Table 5 lists the distribution of operations.

Table 5: Caching Application Block Workload Profile

Test Case	Percentage
Add	80 percent
Read	15 percent
Remove	4 percent
Flush	1 percent
Total	100 percent

Setting Up the Load Test

The load test simulates many clients accessing the application block at the same time. Use the information in the workload profile to set the load test parameters. For more information, see Creating a Load Test.

Recording Baseline and Application Block Metrics

The baseline tests are a subset of the Caching Application Block performance tests. The application block is tested against ASP.NET and Enterprise Library 1.1. The two operations that are tested are adding an item to the cache and reading an item from the cache. These operations are tested with 1 user, 10 users, and 50 users.

Table 6 is the template to record the performance metrics both for the baseline and for the application block. The first column lists the configuration options for Enterprise Library 1.1, ASP.NET, and Enterprise Library–January 2006. The Setting column allows you to circle which option or operation you selected. Record the performance metrics in the other columns. Note that ASP.NET uses its default configuration. For information about these settings, see *ASP.NET Configuration Settings* on MSDN.

Table 6: Template to Record and Compare Metrics

Configuration for Enterprise Library 1.1	Setting	Users	TPS	Response times (ms)	Total transactions	Overhead (percent)
Instrumentation	On/Off	NA	NA	NA	NA	NA
Scavenging	On/Off	NA	NA	NA	NA	NA
Cache storage	In-memory	NA	NA	NA	NA	NA
Item size	2KB/50KB/100KB	NA	NA	NA	NA	NA
Operation	Add/Read	1				Baseline
Operation	Add/Read	10				Baseline
Operation	Add/Read	50				Baseline

Default configuration for ASP.NET	Setting	Users	TPS	Response times (ms)	Total transactions	Overhead (percent)
Instrumentation	Default	NA	NA	NA	NA	NA
Scavenging	Default	NA	NA	NA	NA	NA
Cache storage	Default	NA	NA	NA	NA	NA
Item size	Default	NA	NA	NA	NA	NA
Operation	Add/Read	1				Baseline
Operation	Add/Read	10				Baseline
Operation	Add/Read	50				Baseline
Configuration for application block	**Setting**	**Users**	**TPS**	**Response times (ms)**	**Total transactions**	**Overhead (percent)**
Instrumentation	On/Off	NA	NA	NA	NA	NA
Scavenging	On/Off	NA	NA	NA	NA	NA
Cache storage	In-memory	NA	NA	NA	NA	NA
Item size	2KB/50KB/100KB	NA	NA	NA	NA	NA
Operation	Add/Read	1				
Operation	Add/Read	10				
Operation	Add/Read	50				

For an explanation of these metrics, see *Measuring Performance*.

Testing the Logging Application Block

The tests for the Logging Application Block use a variety of trace listeners and formatters. Table 7 lists the scenarios for the Logging Application Block. Here is an explanation of each of the columns:

- **Logging mechanism**. This column lists either the trace listeners or the message queuing distributor service.
- **Action**. This column lists the action the test case performs.
- **Users**. This column lists the different numbers of users that the tests simulated.
- **Formatters**. This column lists the Logging Application Block formatters that were used.
- **Trace listeners**. This column lists the Logging Application Block trace listeners that were used.

Table 7: Logging Application Block Scenarios

Logging mechanism	Action	Users	Formatters	Trace listeners
Trace listeners	Log message	1, 10, 50, 150, 300	Text Formatter; Binary Formatter	Event Log; MSMQ
Place a log message in a message queue and use the distributor service to send the message to the trace listeners.	Log message	1, 10, 50, 150, 300	Text Formatter; Binary Formatter	Event Log; MSMQ; Database; WMI; Flat File

Creating the Test Harness

For an example of how to create a test harness, see *Building Test Harnesses*. Iterations of the test should create the **LogEntry** object so that the cost of creating the object is included in the performance metrics. For an example of how to create a domain object for each iteration, see *Using the Test Script*. Create a Web page to run the test harness. Figure 4 is an example of a test harness for the Logging Application Block.

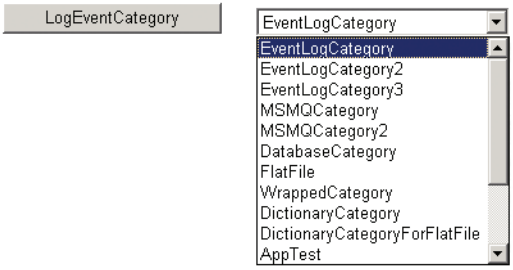

Figure 4 *Logging Application Block Test Harness*

Clicking the **LogEventCategory** button logs a message that is formatted with one of the formatters listed in Table 7. The application block sends the message to the trace listener that is associated with the selected category. For more information about using formatters, categories, and trace listeners, see the Enterprise Library documentation for the Logging Application Block.

Creating the Test Code

The test code implements the scenarios. There are two versions of the test code. One version is for Enterprise Library and the other version is for the .NET Framework.

Table 8 lists the test code for the Logging Application Block.

Table 8: Logging Application Block Test Code

ID	Scenario	Test code
1	Log to trace listener using the formatter associated with the selected category. The trace listener may be the Event Log trace listener, the MSMQ trace listener, or the WMI trace listener. This depends on the configuration of the application block.	protected void btn_LogEventCategory_ Click(object sender, EventArgs e) { // A LogEntry object is created for each // test iteration. This is to // simulate the worst case code path. LogEntry log = new LogEntry(); log.Message = "Logging To Event Log"; log.Categories.Add(DropDownList1.Se- lectedValue); log.Priority = 6; log.EventId = 100; log.Severity = TraceEventType.Informa- tion; Logger.Write(log); }

Table 9 lists the .NET Framework test code.

Table 9: .NET Framework Test Code

ID	Scenario	Test code
1	Log to Event Log trace listener.	protected void ButtonEventLog_Click(object sender, EventArgs e) { EventLog elog = new EventLog(); elog.Source = "Entlib Tests2"; elog.WriteEntry("Message"); elog.Close(); }
2	Log to MSMQ trace listener.	protected void ButtonMSMQ_Click(object sender, EventArgs e) { MessageQueue(@".\Private$\entlib",true)) { Message queueMessage = new Message(); queueMessage.Body = "Message"; queueMessage.Label = "label"; queueMessage.Priority = MessagePriority.High; messageQueue.Send(queueMessage); messageQueue.Close(); } }

continued

ID	Scenario	Test code

3	Log to WMI trace listener.	protected void WMITest_Click(object sender, EventArgs e)
		{
		LogEntry logEntry = new LogEntry();
		// Create a new event.
		EventDetails eventDetails = new EventDetails();
		// Set the event details.
		eventDetails.Message = "Test WMIin .NET Framework";
		eventDetails.Guid = Guid.NewGuid().ToString();
		eventDetails.Type = 2;
		Instrumentation.Fire(eventDetails);
		}
		
		EventDetails class
		[InstrumentationClass(InstrumentationType.Event)]
		public class EventDetails
		{
		public string Message;
		public string Guid;
		public int Type;
		}

Profiling the Workload

The number of users increases over time. The number of users was 1, 10, 50, 150, and 300. There was no test mix and there was no think time.

Setting Up the Load Test

The load test simulates many clients accessing the application block at the same time. Use the information in the workload profile to set the load test parameters. For more information, see Creating a Load Test.

Recording Baseline and Application Block Metrics

The baseline tests are a subset of the Logging Application Block performance tests. The objective is to compare the following overhead costs:

- The cost of using the Logging Application Block Event Log trace listener with the costs of using Enterprise Library 1.1 and the .NET Framework to write to the system event log.
- The cost of using the Logging Application Block WMI trace listener against the costs of using Enterprise Library 1.1 and the .NET Framework WMI events.

- The cost of using the Logging Application Block MSMQ trace listener against the costs of using Enterprise Library 1.1 and the .NET Framework Message Queuing service.

These operations were tested with 1 user, 10 users, and 50 users.

Table 10 is the template to record the performance metrics both for the baseline and for the application block.

Table 10: Template to Record and Compare Metrics

Operation	Users	TPS	Response time (ms)	Total transactions	Overhead (percent)
.NET Framework writes to event log.	1				Baseline
.NET Framework writes to event log.	10				Baseline
.NET Framework writes to event log.	50				Baseline
.NET Framework raises a WMI event.	1				Baseline
.NET Framework raises a WMI event.	10				Baseline
.NET Framework raises a WMI event.	50				Baseline
.NET Framework uses Message Queuing to write a message.	1				Baseline
.NET Framework uses Message Queuing to write a message.	10				Baseline
.NET Framework uses Message Queuing to write a message.	50				Baseline
Enterprise Library 1.1 uses Event Log sink.	1				Baseline
Enterprise Library 1.1 uses Event Log sink.	10				Baseline

continued

Operation	Users	TPS	Response time (ms)	Total transactions	Overhead (percent)
Enterprise Library 1.1 uses Event Log sink.	50				Baseline
Enterprise Library uses WMI sink.	1				Baseline
Enterprise Library 1.1 uses WMI sink.	10				Baseline
Enterprise Library 1.1 uses WMI sink.	50				Baseline
Enterprise Library 1.1 uses MSMQ sink.	1				Baseline
Enterprise Library 1.1 uses MSMQ sink.	10				Baseline
Enterprise Library 1.1 uses MSMQ sink.	50				Baseline
Logging Application Block uses Event Log trace listener.	1				
Logging Application Block uses Event Log trace listener.	10				
Logging Application Block uses Event Log trace listener.	50				
Logging Application Block uses WMI trace listener.	1				
Logging Application Block uses WMI trace listener.	10				
Logging Application Block uses WMI trace listener.	50				
Logging Application Block uses MSMQ trace listener.	1				
Logging Application Block uses MSMQ trace listener.	10				

Operation	Users	TPS	Response time (ms)	Total transactions	Overhead (percent)
Logging Application Block uses MSMQ trace listener.	50				

For an explanation of these metrics, see *Measuring Performance*.

Testing the Data Access Application Block

The tests for the Data Access Application Block use a variety of database access methods. To avoid external constraints, such as network I/O or disk I/O for SQL Server, all the queries to the database return small payloads. Most of the test cases read from the database. Some of the queries were stored procedures and Transact-SQL script files that were used with the Northwind sample database. Because the target is Enterprise Library, it was important to ensure that there were no deadlocks or SQL Server exceptions to contaminate the measurements. For more information, see Tuning the Test Environment.

Table 11 lists the scenarios for the Data Access Application Block. Here is an explanation of each of the columns:

- **ID**. This column lists the ID number for each scenario.
- **Scenario**. This column lists the Data Access Application Block scenario being tested.
- **Transaction**. This column lists that the scenarios are tested both with and without a transaction.
- **Users**. This column lists the different numbers of users that the tests simulated.
- **Database provider**. This column lists the database provider that is used.

Table 11: Data Access Application Block Scenarios

ID	Scenario	Transaction	Users	Database provider
1	Use a **DbDataReader** to read multiple rows	With/Without Transaction	1, 10, 50, 150, 300	SQL Server
2	Use a **DataSet** to read multiple rows	With/Without Transaction	1, 10, 50, 150, 300	SQL Server
3	Use **ExecuteDataSet** to execute a stored procedure that has parameter values and return the results in a **DataSet**.	With/Without Transaction	1, 10, 50, 150, 300	SQL Server
4	Use **ExecuteScalar** to retrieve a single item.	With/Without Transaction	1, 10, 50, 150, 300	SQL Server

continued

ID	Scenario	Transaction	Users	Database provider
5	Use **ExecuteNonQuery** to retrieve output parameters.	With/Without Transaction	1, 10, 50, 150, 300	SQL Server
6	Use a **DataSet** to update the database.	With/Without Transaction	1, 10, 50, 150, 300	SQL Server
7	Add a new **DataTable** to the existing **DataSet**.	With/Without Transaction	1, 10, 50, 150, 300	SQL Server
8	Retrieve multiple rows as XML	With/Without Transaction	1, 10, 50, 150, 300	SQL Server

Creating the Test Harness

For an example of how to create a test harness, see Building Test Harnesses. Iterations of the test should call the **CreateDatabase** method and create any other objects, such as a **DbCommandWrapper**, that the application block uses to execute a query. For an example of how to create a domain object for each iteration, see Using the Test Script. Create a Web page to run the test harness.

Creating the Test Code

The test code implements the scenarios. There are two versions of the test code shown here. The Enterprise Library uses one version and ADO.NET uses the other version. Enterprise Library 1.1 was also used to establish a baseline. Its database methods differ slightly from the methods in Enterprise Library–January 2006. The Enterprise Library 1.1 code is not shown. The only scenario used for baseline comparisons was scenario 5.

When the application block executes the scenarios within a transaction, it creates a connection object and retrieves a transaction object. Then, the application block uses the appropriate method overload for working with transactions.

Table 12 lists the Data Access Application Block test code.

Table 12: Data Access Application Block Test Code

ID	Scenario	Test code

1	Use a **DbDataReader** to read multiple rows.	protected void ButtonDataReader_Click(object sender, EventArgs e)
		{
		Database db = DatabaseFactory.CreateDatabase("DataSQLTest");
		string sqlCommand = "Select CategoryID from Categories where Description LIKE'Cheeses'";
		DbCommand dbCommandWrapper = db.GetSqlStringCommand(sqlCommand);
		IDataReader dataReader = null;
		// We pass a transaction object to ExecuteReader
		// along with transaction the test cases.
		dataReader = db.ExecuteReader(dbCommandWrapper);
		dataReader.Close();
		}

2	Use a **DataSet** to read multiple rows.	protected void ButtonExecuteDataSet_Click(object sender, EventArgs e)
		{
		Database db= DatabaseFactory.CreateDatabase("DataSQLTest");
		string sqlCommand = "Select * from products where productid=1";
		DbCommand dbCommandWrapper = db.GetSqlStringCommand(sqlCommand);
		// We pass a transaction object to ExecuteDataSet
		// with the transaction test cases.
		DataSet dsActualResult= db.ExecuteDataSet(dbCommandWrapper);
		}

3	Use **ExecuteDataSet** to execute a stored procedure that has parameter values and return the results in a **DataSet**.	protected void Button4_Click(object sender, EventArgs e)
		{
		Database db = DatabaseFactory.CreateDatabase("DataSQLTest");
		
		using (DataSet dsActualResult = db.ExecuteDataSet("CustOrdersOrders", new object[] {"ALFKI"}))
		{
		
		}
		}

continued

ID	Scenario	Test code

4	Use **ExecuteScalar** to retrieve a single item.	`protected void ButtonExecuteScalarClick(object sender, EventArgs e)`
		`{`
		`  Database db= DatabaseFactory.CreateDatabase("DataSQLTest");`
		`  string sqlCommand = "Select CategoryID from Categories  where Description LIKE'Cheeses'";`
		`  DbCommand dbCommandWrapper =  db.GetSqlStringCommand(sqlCommand);`
		`// We pass a transaction object to ExecuteScalar`
		`// with transaction test cases.`
		`object actualResult = db.ExecuteScalar(dbCommandWrapper);`
		`dbCommandWrapper.Dispose();`
		`}`

5	Use **ExecuteNonQuery** to retrieve output parameters.	`protected void ExecuteSP_Click(object sender, EventArgs e)`
		`{`
		`  // The Database object is created over`
		`  // load test iterations.`
		`  Database db= DatabaseFactory.CreateDatabase("DataSQLTest");`
		`  string spName = "TenMostExpensiveProducts";`
		`  db.ExecuteNonQuery(CommandType.StoredProcedure, spName);`
		`}`

6	Use a **DataSet** to update the database.	

7	Add a new **DataTable** to the existing **DataSet**.	`protected void ButtonLoadDataSet_Click(object sender, EventArgs e)`
		`{`
		`  Database db= DatabaseFactory.CreateDatabase("DataSQLTest");`
		`  DataSet ItemDataSet = new DataSet();`
		`  string sqlCommand = "Select top 20 * from products";`
		`  DbCommand  dbCommandWrapper=db.GetSqlStringCommand(sqlCommand);`
		`  string ItemsTable = "Products";`
		`  // We pass a transaction object to LoadDataSet`
		`  // with transaction test cases.`
		`  db.LoadDataSet(dbCommandWrapper, ItemDataSet, ItemsTable);`
		`  dbCommandWrapper.Dispose();`
		`}`

ID	Scenario	Test code

8	Retrieve multiple rows as XML.	protected void Button_Click(object sender, EventArgs e)
		{
		SqlDatabase dbSQL =
		DatabaseFactory.CreateDatabase("DataSQLTest") as SqlDatabase;
		// Use "FOR XML AUTO" to have SQL return XML data.
		string sqlCommand = "Select CategoryID from Categories where Description LIKE'Cheeses'FOR XML AUTO";
		System.Data.Common.DbCommand dbCommandWrapper =
		dbSQL.GetSqlStringCommand(sqlCommand) as DbCommand;
		XmlReader productsReader = null
		// We pass a transaction object to
		// ExecuteXmlReader with transaction
		// test cases.
		productsReader = dbSQL.ExecuteXmlReader(dbCommandWrapper);
		dbCommandWrapper.Connection.Close();
		productsReader.Close();
		dbCommandWrapper.Dispose();
		}

Table 13 lists the ADO.NET Framework test code.

Table 13: ADO.NET Framework Test Code

ID	Scenario	Test code

5	Use the equivalent of the **ExecuteNonQuery** method to retrieve output parameters.	protected void ButtonAdoNet_Click(object sender, EventArgs e)
		{
		SqlConnection sqlcon = new SqlConnection("server=****;database=Northwind;User Id=****;password=****");
		sqlcon.Open();
		SqlCommand sqlcommand = new SqlCommand();
		sqlcommand.Connection = sqlcon;
		sqlcommand.CommandType = CommandType.StoredProcedure;
		sqlcommand.CommandText = "TenMostExpensiveProducts";
		sqlcommand.ExecuteNonQuery();
		sqlcommand.Dispose();
		sqlcon.Close();
		}

Profiling the Workload

The number of users increases over time. The number of users was 1, 10, 50, 150, and 300. There was no test mix and there was no think time.

Setting Up the Load Test

The load test simulates many clients accessing the application block at the same time. Use the information in the workload to set the load test parameters. For more information, see Creating a Load Test.

Recording Baseline and Application Block Metrics

The baseline tests are a subset of the Data Access Application Block performance tests. The objective is to compare the overhead costs of using the Data Access Application Block's **ExecuteNonQuery** method with the equivalent ADO.NET method and with Enterprise Library 1.1.

Table 14 is the template to record the performance metrics for both the baseline and the application block.

Table 14: Template to Record and Compare Metrics

Operation	Users	TPS	Response time (ms)	Total transactions	Overhead (percent)
ADO.NET equivalent of **ExecuteNonQuery** method	1				Baseline
ADO.NET equivalent of **ExecuteNonQuery** method	10				Baseline
ADO.NET equivalent of **ExecuteNonQuery** method	50				Baseline
Enterprise Library 1.1 **ExecuteNonQuery** method	1				Baseline
Enterprise Library 1.1 **ExecuteNonQuery** method	10				Baseline
Enterprise Library 1.1 **ExecuteNonQuery** method	50				Baseline

Operation	Users	TPS	Response time (ms)	Total transactions	Overhead (percent)
Data Access Application Block **ExecuteNonQuery** method	1				
Data Access Application Block **ExecuteNonQuery** method	10				
Data Access Application Block **ExecuteNonQuery** method	50				

For an explanation of these metrics, see *Measuring Performance*.

Testing the Exception Handling Application Block

The tests for the Exception Handling Block use different handlers. Table 15 lists the scenarios for the Exception Handling Application Block. Here is an explanation of each of the columns:

- **ID**. This column lists the ID number for each scenario.
- **Scenario**. This column lists the Exception Handling Application Block scenario being tested.
- **Users**. This column lists the different numbers of users that the tests simulated.

Table 15: Exception Handling Application Block Scenarios

ID	Scenario	Users
1	Process an exception without handlers.	1, 5, 10, 50
2	Process an exception with the **Wrap** handler.	1, 5, 10, 50
3	Process an exception with the **Replace** handler.	1, 5, 10, 50
4	Process an exception with the **Logging** handler.	1, 5, 10, 50
5	Process an exception with the **Replace** and **Logging** handlers.	1, 5, 10, 50

Creating the Test Harness

For an example of how to create a test harness, see Building Test Harnesses. Iterations of the test should create an **Exception** object. For an example of how to create a domain object for each iteration, see Using the Test Script. In addition, the application block should handle and rethrow the exception for each iteration. Create a Web page to run the test harness.

Creating the Test Code

The test code implements the scenarios. There are two versions of the test code. The Enterprise Library uses one version and the .NET Framework uses the other version. Only the first scenario is used in the baseline measurements.

Table 16 lists the Exception Handling Application Block test code. The code is the same for all the scenarios. Only the exception policies vary from one scenario to another.

Table 16: Exception Handling Application Block Test Code

ID	Scenario	Test code

1	Process an exception with the **Wrap** handler.	`protected void ButtonException_Click(object sender, EventArgs e)`
		`{`
		`  try`
		`  {`
		`Exception originalException = new System.FormatException("Original Exception: format exception");`
		`bool rethrow`
		`  =ExceptionPolicy.HandleException(originalException,`
		`"WrapThrowNewDifferentAssembly");`
		`if (rethrow) throw originalException;`
		`  }`
		`  catch (Exception ex){ }`
		`}`

Table 17 lists the .NET Framework test code.

Table 17: .NET Framework Test Code

ID	Scenario	Test code

1	Throw a wrapped exception.	`protected void ExceptionHandling_Click(object sender, EventArgs e)`
		`{`
		`  try`
		`  {`
		`  Exception originalException = new`
		`  System.FormatException("Original Exception: format`
		`  exception");`
		`  throw (new System.FormatException("originalException",`
		`  originalException));`
		`  }`
		`  catch {}`
		`}`

Profiling the Workload

The number of users increases over time. The number of users was 1, 10, 50, 150, and 300. There was no test mix and there was no think time.

Setting Up the Load Test

The load test simulates many clients accessing the application block at the same time. Use the information in the workload to set the load test parameters. For more information, see Creating a Load Test.

Recording Baseline and Application Block Metrics

The baseline tests are a subset of the Exception Handling Application Block performance tests. The objective is to compare the overhead costs of using the Exception Handling Application Block's wrap handler with the equivalent .NET Framework method and with Enterprise Library 1.1.

Table 18 is the template to record the performance metrics both for the baseline and for the application block.

Table 18: Template to Record and Compare Metrics

Operation	Users	TPS	Response time (ms)	Total transactions	Overhead (percent)
.NET Framework wraps and throws an exception. Enterprise Library 1.1 does the same.	1				Baseline
.NET Framework wraps and throws an exception. Enterprise Library 1.1 does the same.	10				Baseline
.NET Framework wraps and throws an exception. Enterprise Library 1.1 does the same.	50				Baseline

continued

Operation	Users	TPS	Response time (ms)	Total transactions	Overhead (percent)
Enterprise Library 1.1 wraps and throws an exception.	1				Baseline
Enterprise Library 1.1 wraps and throws an exception.	10				Baseline
Enterprise Library 1.1 wraps and throws an exception.	50				Baseline
Exception Handling Application Block wraps and throws an exception.	1				
Exception Handling Application Block wraps and throws an exception.	10				
Exception Handling Application Block wraps and throws an exception.	50				

For an explanation of these metrics, see *Measuring Performance*.

Testing the Cryptography Application Block

The tests for the Cryptography Application Block use a variety of symmetric algorithm providers and hash providers. Table 19 lists the scenarios for the Cryptography Application Block. Here is an explanation of each of the columns.

- **ID**. This column lists the ID number for each scenario.
- **Scenario**. This column lists the Cryptography Application Block scenario being tested.

- **Users**. This column lists the different numbers of users that the tests simulated.
- **Cryptography provider**. This column lists the symmetric algorithm provider or hash provider used in the scenario.

Table 19: Cryptography Application Block Scenarios

ID	Scenario	Users	Cryptography provider
1	Hash plaintext and compare hashed value with plaintext.	1, 10, 50, 150	**HMACSHA1**
2	Hash plaintext and compare hashed value with plaintext.	1, 10, 50, 150	**HMACSHA1NoSalt**
3	Hash plaintext and compare hashed value with plaintext.	1, 10, 50, 150	**SHA512Managed**
4	Hash plaintext and compare hashed value with plaintext.	1, 10, 50, 150	**SHA512ManagedNoSalt**
5	Hash plaintext and compare hashed value with plaintext.	1, 10, 50, 150	**SHA384Managed**
6	Hash plaintext and compare hashed value with plaintext.	1, 10, 50, 150	**SHA384ManagedNoSalt**
7	Hash plaintext and compare hashed value with plaintext.	1, 10, 50, 150	**SHA256Managed**
8	Hash plaintext and compare hashed value with plaintext.	1, 10, 50, 150	**SHA256ManagedNoSalt**
9	Hash plaintext and compare hashed value with plaintext.	1, 10, 50, 150	**SHA1Managed**
10	Hash plaintext and compare hashed value with plaintext.	1, 10, 50, 150	**SHA1ManagedNoSalt**
11	Hash plaintext and compare hashed value with plaintext.	1, 10, 50, 150	**SHA1CryptoServiceProvider**
12	Hash plaintext and compare hashed value with plaintext.	1, 10, 50, 150	**SHA1CryptoServiceProviderNoSalt**
13	Encrypt and decrypt plaintext with a symmetric algorithm provider.	1, 10, 50, 150	**MD5CryptoServiceProvider**
14	Encrypt and decrypt plaintext with a symmetric algorithm provider.	1, 10, 50, 150	**MD5CryptoServiceProviderNoSalt**
15	Encrypt and decrypt plaintext with a symmetric algorithm provider.	1, 10, 50, 150	**MD5CryptoServiceProvider**
16	Encrypt and decrypt plaintext with a symmetric algorithm provider.	1, 10, 50, 150	**MD5CryptoServiceProviderNoSalt**
17	Encrypt and decrypt plaintext with a symmetric algorithm provider.	1, 10, 50, 150	**MACTripleDES**

continued

ID	Scenario	Users	Cryptography provider
18	Encrypt and decrypt plaintext with a symmetric algorithm provider.	1, 10, 50, 150	**MACTripleDESNoSalt**
19	Encrypt and decrypt plaintext with a symmetric algorithm provider.	1, 10, 50, 150	**DESCryptoServiceProvider**
20	Encrypt and decrypt plaintext with a symmetric algorithm provider.	1, 10, 50, 150	DPAPI Symmetric Cryptography Provider
21	Encrypt and decrypt plaintext with a symmetric algorithm provider.	1, 10, 50, 150	DPAPI Symmetric Cryptography Provider1
22	Encrypt and decrypt plaintext with a symmetric algorithm provider.	1, 10, 50, 150	**RC2CryptoServiceProvider**
23	Encrypt and decrypt plaintext with a symmetric algorithm provider.	1, 10, 50, 150	**RijndaelManaged**
24	Encrypt and decrypt plaintext with a symmetric algorithm provider.	1, 10, 50, 150	**TripleDESCryptoServiceProvider**

Creating the Test Harness

For an example of how to create a test harness, see Building Test Harnesses. Iterations of the test should create the symmetric algorithm and hash providers. For an example of how to create a domain object for each iteration of the test, see Using the Test Script. Create a Web page to run the test harness. Figure 5 illustrates an example of a Web page for the Cryptography Application Block test harness.

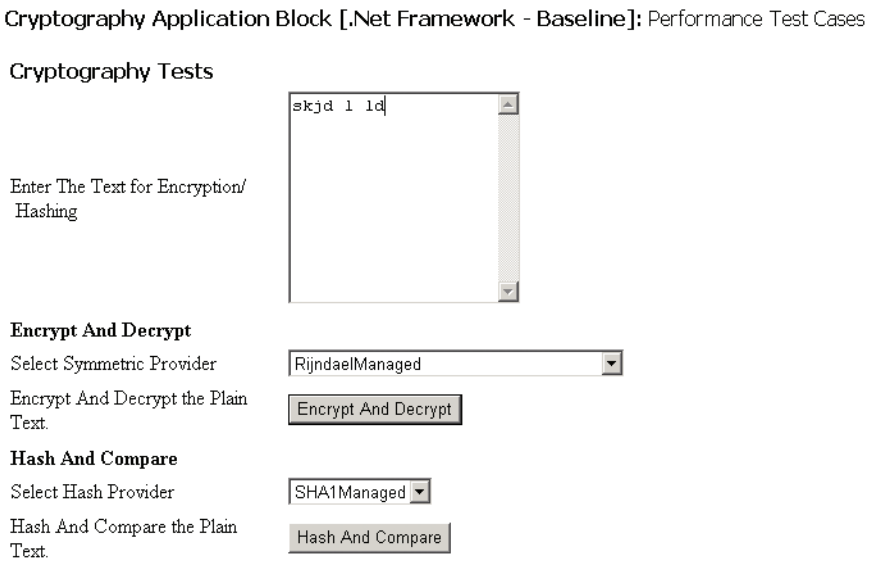

Figure 5 *Cryptography Application Block test harness*

Creating the Test Code

The test code implements the scenarios. There are two versions of the test code. The Enterprise Library uses one version and the .NET Framework uses the other version. Enterprise Library 1.1 was not used as a baseline for the Cryptography Application Block.

Table 20 lists the Cryptography Application Block test code.

Table 20: Cryptography Application Block Test Code

ID	Scenario	Test code

1	Hash plain-text and compare hashed value with plaintext.	`protected void ButtonHash_Click(object sender, EventArgs e)`
		`{`
		`  byte[] b = new byte[Int32.Parse(TextBox1.Text)];`
		`  byte[] hs=Cryptographer.CreateHash(ddlSymmetricProvider.Select-edValue, strEncryptedText);`
		`  Cryptographer.CompareHash("ddlSymmetricProvider.SelectedValue, b, hs);`
		`  // Optionally, the factory is created`
		`  // instead of using the static CreateHash CompareHash.`
		`  HashProviderFactory factory = new HashProviderFactory();`
		`  IHashProvider hashProvider = factory.Create(ddlSymmetricProvide r.SelectedValue);`
		`  hashProvider.CreateHash(b);`
		`  HashProviderFactory factory = new HashProviderFactory();`
		`  IHashProvider hashProvider = factory.Create(ddlSymmetricProvide r.SelectedValue);`
		`  hashProvider.CompareHash(b);`
		`}`

23	Encrypt and decrypt plaintext with a symmetric algorithm provider.	`protected void ButtonCrypto_Click(object sender, EventArgs e)`
		`{`
		`  // The size of the byte array is read`
		`  // from the text box.`
		`  byte[] b = new byte[long.Parse(TextBox1.Text)];`
		`  // The factory is created for each iteration.`
		`  // The static EncryptSymmetric method creates the factory.`
		`  byte[] enc = =Cryptographer.EncryptSymmetric(ddlSymmetric Provider.SelectedValue b);`
		`  ISymmetricCryptoProvider symmprovider =`
		`  factory.Create(ddlSymmetricProvider.SelectedValue);`
		`  Cryptographer.DecryptSymmetric(ddlSymmetricProvider.Selected-Value, enc);`
		`  // Or, optionally, the factory is created.`
		`  SymmetricCryptoProviderFactory factoryEncrypt = new`
		`  SymmetricCryptoProviderFactory();`
		`  byte[] result = symmprovider.Encrypt(b);`
		`  SymmetricCryptoProviderFactory factoryDecrypt = new`
		`  SymmetricCryptoProviderFactory();`
		`  byte[] result = symmprovider.Decrypt(b);`
		`}`

Table 21 lists the .NET Framework test code.

Table 21: .NET Framework Test Code

ID	Scenario	Test code

1	Hash plaintext with **HMACSAH1** hash provider and compare hashed value with plaintext.	protected void btnHashAndCompare_Click(object sender, EventArgs e)
		{
		string strKey = "EC14B8D88ABBD892F8736193DD35103B4F83AF46C39C27457412150EAA2E840B82C67CDCD7-2ACDD457DB0D91E8EDA504A6F06B0AF98DA7E2BD7C44CCE3CD3907";
		byte[] byteArrayKey = new byte[64];
		byte[] byteArrayPlainText;
		byte[] byteArrayHashedText;
		byte[] byteArrayHashedTextToCompare;
		UnicodeEncoding textConverter = new UnicodeEncoding();
		
		// Convert the key in string format to byte array.
		byteArrayKey = textConverter.GetBytes(strKey);
		
		using (HMACSHA1 myHMACSHA1 = new HMACSHA1(byteArrayKey))
		{
		// Convert the data to a byte array.
		byteArrayPlainText = textConverter.GetBytes(txtBoxPlainText.Text);
		byteArrayHashedText = myHMACSHA1.ComputeHash(byteArrayPlainText);
		
		byteArrayHashedTextToCompare = myHMACSHA1.ComputeHash(byteArrayPlainText);
		// Compare the computed hash with
		// the stored value.
		for (int i = 0; i < byteArrayHashedText.Length; i++)
		{
		if (byteArrayHashedTextToCompare[i] != byteArrayHashedText[i])
		{
		Response.Write("Hash values differ! Encoded file has been tampered with!");
		break;
		}
		}
		myHMACSHA1.Clear();
		}

continued

ID	Scenario	Test code

23	Encrypt and decrypt plaintext with **RijndaelManaged** provider.	protected void btnEncryptAndDecrypt_Click(object sender, EventArgs e)
		{
		  byte[] key1;
		  byte[] byteArrayEncryptedText;
		  byte[] byteArrayDecryptedText;
		  byte[] byteArrayPlainText;
		  UnicodeEncoding textConverter = new UnicodeEncoding();
		  ICryptoTransform encryptor, decryptor;
		  RijndaelManaged myRijndealManaged = new RijndaelManaged();
		
		  key1 =
		  Convert.FromBase64String
		  ("zKqffFs392WuxGMhFkfhYj/HtDDIjSiXnYqMlMMzrNc=");
		  myRijndealManaged.GenerateIV();
		
		  // Convert the data to a byte array.
		  byteArrayPlainText =
		  textConverter.GetBytes(txtBoxPlainText.Text);
		
		  encryptor = myRijndealManaged.CreateEncryptor(key1,
		  myRijndealManaged.IV);
		  byteArrayEncryptedText =
		  encryptor.TransformFinalBlock(byteArrayPlainText, 0,
		  byteArrayPlainText.Length);
		
		  // Get a decryptor that uses the same key
		  // and IV as the encryptor.
		  decryptor = myRijndealManaged.CreateDecryptor(key1,
		  myRijndealManaged.IV);
		
		  // Now decrypt the previously encrypted
		  // message using the decryptor
		  // obtained in the above step.
		  byteArrayDecryptedText = new
		  byte[byteArrayEncryptedText.Length];
		  byteArrayDecryptedText =
		  decryptor.TransformFinalBlock(byteArrayEncryptedText, 0,
		  byteArrayEncryptedText.Length);
		
		  // Convert the byte array back into a string.
		  txtBoxPlainText.Text =
		  textConverter.GetString(byteArrayDecryptedText);
		}

Profiling the Workload

The number of users increases over time. The number of users was 1, 5, 10, and 50. There was no test mix and there was no think time.

Setting Up the Load Test

The load test simulates many clients accessing the application block at the same time. Use the information in the workload profile to set the load test parameters. For more information, see Creating a Load Test.

Recording Baseline and Application Block Metrics

The baseline tests are a subset of the Cryptography Application Block performance tests. The first objective is to compare the cost of using the Cryptography Application Block to encrypt and decrypt plaintext with the **RijndaelManaged** provider against of the cost of using the .NET Framework to do the same thing. The second objective is to compare the cost of using the Cryptography Application to hash plaintext with the **HMACSAH1** provider and then compare the hashed value with the plaintext against of the cost of using the .NET Framework to do the same thing. For information about acceptable performance limits, see Measuring Performance.

Table 22 is the template to record the performance metrics both for the baseline and for the application block.

Table 22: Template to Record and Compare Metrics

Operation	Users	TPS	Response time (ms)	Total transactions	Overhead (percent)
.NET Framework uses **RijndaelManaged** provider to encrypt and decrypt plaintext.	1				Baseline
.NET Framework uses **RijndaelManaged** provider to encrypt and decrypt plaintext.	10				Baseline
.NET Framework uses **RijndaelManaged** provider to encrypt and decrypt plaintext.	50				Baseline
.NET Framework uses **HMACSAH1** provider to hash and compare plaintext.	1				Baseline

continued

Operation	Users	TPS	Response time (ms)	Total transactions	Overhead (percent)
.NET Framework uses **HMACSAH1** provider to hash and compare plaintext.	10				Baseline
.NET Framework uses **HMACSAH1** provider to hash and compare plaintext.	50				Baseline
Cryptography Application Block uses **RijndaelManaged** provider to encrypt and decrypt plaintext.	1				
Cryptography Application Block uses **RijndaelManaged** provider to encrypt and decrypt plaintext.	10				
Cryptography Application Block uses **RijndaelManaged** provider to encrypt and decrypt plaintext.	50				
Cryptography Application Block uses **HMACSAH1** provider to hash and compare plaintext.	1				
Cryptography Application Block uses **HMACSAH1** provider to hash and compare plaintext.	10				
Cryptography Application Block uses **HMACSAH1** provider to hash and compare plaintext.	50				

Testing the Security Application Block

The tests for the Security Application Block tested the application block's authorization methods. Table 23 lists the scenarios for the Security Application Block. Here is an explanation of each of the columns.

- **ID**. This column lists the ID number for each scenario.
- **Scenario**. This column lists the Security Application Block scenario being tested.
- **Users**. This column lists the different numbers of users that the tests simulated.
- **Configuration setting**. This column lists the cache that stores the security information or the authorization provider.

Table 23: Security Application Block Scenarios

ID	Scenario	Users	Configuration setting
1	Save or read the **Identity** property.	1, 10, 50, 150	Property is cached in SQL Server.
2	Save or read the **Principal** property.	1, 10, 50, 150	Property is cached in SQL Server.
3	Save or read the **Profile** property.	1, 10, 50, 150	Property is cached in SQL Server.
4	Authorize the user.	1, 10, 50, 150	Use AzMan to authorize the user.
5	Authorize the user.	1, 10, 50, 150	Use the Authorization Rule Provider to authorize the user.

Creating a Test Harness

For an example of how to create a test harness, see Building Test Harnesses. Iterations of the test should create the security provider objects. For an example of how to create a domain object for each iteration of the test, see Using the Test Script. Create a Web page to run the test harness.

Creating the Test Code

Because of time constraints, the Security Application Block was not measured against a baseline. Table 24 lists the Security Application Block test code.

Table 24: Security Application Block Test Code

ID	Scenario	Test code

1	Save or read the **Identity** property.	`protected void SQLSaveReadIdentity_Click(object sender, EventArgs e)`
		`{`
		`ISecurityCacheProvider securityCache = SecurityCacheFactory.GetSecurityCacheProvider("CacheProvidersDB");`
		`IToken token = securityCache.SaveIdentity(identity);`
		`securityCache.ExpireIdentity(token);`
		`IIdentity cachedIdentity = securityCache.GetIdentity(token);`
		`}`

2	Save or read the **Principal** property.	`protected void SQLSaveReadPrincipal_Click(object sender,EventArgs e)`
		`{`
		`WindowsIdentity wi = WindowsIdentity.GetCurrent();`
		`WindowsPrincipal wp = new WindowsPrincipal(wi);`
		`ISecurityCacheProvider securityCache = SecurityCacheFactory.GetSecurityCacheProvider("CacheProvidersDB");`
		`IToken token = securityCache.SavePrincipal(wp);`
		`IPrincipal Principal=securityCache.GetPrincipal(token);`
		`}`

3	Save or read the **Profile** property.	`protected void SQLSaveReadProfile_Click(object sender, EventArgs e)`
		`{`
		`WindowsIdentity wi = WindowsIdentity.GetCurrent();`
		`WindowsPrincipal wp = new WindowsPrincipal(wi);`
		`ISecurityCacheProvider securityCache = SecurityCacheFactory.GetSecurityCacheProvider("CacheProvidersDB");`
		`IToken token = securityCache.SaveProfile(new object());`
		`IPrincipal Principal = securityCache.GetProfile(token);`
		`}`

4	Authorize the user with AzMan.	`protected void AzmanAuthProvider_Click(object sender, EventArgs e)`
		`{`
		`IAuthorizationProvider azManProvider = AuthorizationFactory.GetAuthorizationProvider("DefaultAzManProvider");`
		`WindowsIdentity identity = WindowsIdentity.GetAnonymous();`
		`bool isAuthorized = azManProvider.Authorize(new WindowsPrincipal(identity), "Authorize Purchase");`
		`}`

ID	Scenario	Test code

5	Authorize the user with the Authorization Rule Provider.	protected void AuthRuleProvider_Click(object sender, EventArgs e)
		{
		AuthorizationRuleProvider authRuleProvider = AuthorizationFactory.GetAuthorizationProvider("DefaultRuleP
		rovider") as AuthorizationRuleProvider;
		boolauthorized = authRuleProvider.Authorize(principal, "TestIdentityAndRoleRuleORMe");
		}

Profiling the Workload

The number of users increases over time. The number of users was 1, 10, 50, 150, and 300. There was no test mix and there was no think time.

Setting Up the Load Test

The load test simulates many clients accessing the application block at the same time. Use the information in the workload profile to set the load test parameters. For more information, see Creating a Load Test.

Detecting Performance Issues

This section discusses the performance counters you should monitor to detect performance issues. These counters pertain to the following areas:

- Disk I/O
- The network
- The load agents
- Locking and contention within the application

The following sections explain how to monitor these components.

Monitoring Disk I/O

Ideally, the CPU on the Web server should use close to 100 percent of its cycles for the tests. When this does not occur, it may indicate that an external resource is constraining the server. For example, SQL Server may become I/O-bound when it writes to the database during the Data Access Application Block tests. This can cause delays. In this situation, you have to remove the constraint because it may cause you to miss performance and contention problems. Possible solutions are to add more physical disk drives (also commonly referred to as "spindles") to the database server disk subsystem in order to provide more disk I/O processing power, change the query so that it does not write to the database, or only write the number of bytes that the disk subsystem can handle without causing delays.

You should monitor I/O performance on the computer that hosts the application block, the computer that hosts SQL Server, and on any other computer that hosts a resource that the application block uses. Table 25 lists the performance counters you can use to analyze disk I/O activity.

Table 25: Disk I/O Performance Counters

Performance monitor counter	Description and recommendations
% Disk Time	This is the percentage of elapsed time that the selected disk drive is busy servicing I/O requests. This should be approximately 5 percent.
% Idle Time	This is the percentage of time during the sample interval that the disk was idle. This number should be approximately 95 percent on both the Web server and the SQL Server.
Disk Reads/Sec and **Disk Writes/Sec**	Together, these counters represent the number of I/O operations issued against a particular disk. Generally, there is a practical limit of 100 to140 operations per second per spindle. Consult with your hardware vendor for a more accurate estimation.
Avg. Disk sec/Read and **Avg. Disk sec/Write**	Together, these counters measure disk latency. Lower values are better than higher values, but this value can vary and is dependent on the size of the I/O operations and the workload characteristics. Numbers also vary across different storage configurations. For example, the storage area network (SAN) cache size and how often the cache is used can greatly impact this metric. On well-tuned online transaction processing (OLTP) systems that are deployed on high performance SANs, the ideal values vary between less than 2 ms for the log files and 4 ms to 10 ms for data. Decision support system (DSS) workloads may have higher latencies of 30 ms or more. For Internet Information Services (IIS), the number should be between 8 ms and 10 ms. Persistent values of greater than 100 ms can indicate I/O problems. However, this value is dependent on workload characteristics and the system hardware. When considering this measurement, keep in mind the normal values for your system.

Performance monitor counter	Description and recommendations
Avg. Disk Bytes/Read and **Avg. Disk Bytes/ Write**	Together, these counters represent the size of the I/O operations. Large I/O sizes can cause slightly higher disk latency. When you use this counter to measure SQL Server I/O operations during the Enterprise Library performance tests, this value tells you the average size of the I/O operations that SQL Server issues to fill query requests.
Avg. Disk Queue Length	This counter represents the average number of read and write requests that are queued for the selected disk during the sample interval. The general rule is to ideally have no more than two requests per spindle. However, this is difficult to measure because most SANs use storage virtualization. In reality, this value may be between 4 requests per spindle and 8 requests per spindle. In general, to detect problems, look for a higher than average disk queue length in combination with a higher than average disk latency. This combination can indicate that the SAN's cache is overused.

Monitoring the Network

The primary point of monitoring the network is to check for bottlenecks. To do this, use the **Network Interface\Bytes Total/sec** performance counter that belongs to the **Win32_PerfRawData_Tcpip_NetworkInterface** class. This counter indicates the rate at which bytes are sent and received over a network adapter.

Ideally, in a 100 MB network, the approximate value should be13,107,200 bytes/sec. However, packet control transfers and connection handshakes cause loss, so not all the bandwidth is available. Usually, this number does not exceed 70 percent of the total bandwidth. Another often overlooked problem is that the network interface card (NIC) is set to half duplex instead of full duplex. To find the card's setting, perform the following steps.

► **To determine NIC mode**

1. On the taskbar, click Start, and then click Control Panel.
2. Click Network Connections.
3. Right-click the name of your network connection.
4. Click Properties.
5. Click Configure.
6. Click the Advanced tab.

Also, a NIC's autonegotiation feature may not properly set the duplex value. You may have to disable the feature and manually set the NIC to full duplex.

Another problem that can increase network traffic is to host multiple Web controls on the Web test page. Remove some of the controls to decrease the Web page's use of network resources.

Monitoring the Load Agents

A common problem during performance tests is that the load agents use all of the system resources. Typically, the CPU is the problem. This may be because the transaction response times are very short. Another reason may be that it is expensive to create the domain objects that the application block uses. To correct this problem, increase the number of load agents. An agent should use no more than 75 percent to 80 percent of the CPU.

Monitoring for Locking and Contention

Locking and contention in an application are major performance issues because they affect the application's scalability. Use a tool. such as the profiler that is in Visual Studio Team System. to pinpoint problems that are caused by locked data structures. To identify contention problems, examine the **.NET CLR LocksAndThreads\Contention Rate/sec** performance counter. This counter displays the rate at which the common language runtime (CLR) unsuccessfully attempts to acquire a managed lock. Sustained higher values may be a cause of concern, particularly if the application block uses only a small percentage of the CPU.

Measuring Performance

Each Enterprise Library performance test ran for 5 minutes to 7 minutes. There were no think times. The warm-up times were 30 seconds. (The warm-up time is used in a test script to ensure that an application reaches a steady state before the test tool starts to record results.) Results were recorded every 15 seconds.

Each Enterprise Library stress test ran for 12 hours to 72 hours. There were no think times. The warm-up time was 5 minutes. Results were recorded every 60 seconds. Usually, 72 hours with no think times is a good simulation of 2 weeks in a production environment.

Data collection samples should contain a minimum of 120 samples. Adjust the sampling interval according to the duration of the test. This is true for both performance and stress tests. Longer tests should have longer intervals between samples to avoid too many measurements and filling up the media store.

Important concepts to keep in mind when you measure performance are:

- Utilization
- Idle time
- Saturation

Utilization is (Resource Busy Time/Total Time of service) *100. Many performance counters measure utilization. They have names that include percentages such as Processor: % Processor Time.

Idle time is 100 percent – Utilization. The Enterprise Library tests use the Logical Disk: % Idle Time counter to measure utilization.

Saturation means that a resource is used to 100 percent of its capacity, or close to it.

There are two sets of performance counters you can use to measure an application block's performance. The first set records the use of system resources. System resources include memory, disk I/O, CPU usage, network I/O, and specific counters associated with the CLR memory and with contention. Also, where applicable, there are performance counters to measure SQL Server performance. For more information about these counters, see Detecting Performance Issues.

The second set of performance counters measure end-to-end transaction times and transactions per second. Here is a summary of the goals for these metrics:

- The **Process(w3wp_.exe)\Private Bytes** and the **Process(w3wp.exe)\Virtual Bytes** performance counters should not increase during the stress and performance tests.
- The number of CLR garbage collection handles or process handles should not increase during stress and performance tests.
- There should be no CLR lock contention coupled with a low measurement for CPU usage. The CPU usage should be at close to 100 percent for all test cases, except where there is dependency on an external resource such as during a SQL Server query.
- The ratio of context switches to system threads should not be more than 20.
- The thread count should not increase during the stress and performance tests.
- The transaction execution times and the transactions per second should remain stable during stress and performance tests.
- There should be no deadlocks or w3p3.exe process restarts.
- The system memory and memory pools should be stable.
- The latency times for reads and writes to logical drives should not be more than 7 ms.
- Only the Exception Handling Application Block should have CLR exceptions.
- Contention should not cause poor CPU usage.

- The elapsed time spent performing garbage collection since the last garbage collection recycle should be less than 10 percent during stress and performance tests.
- A Gen 2 collection should be one-tenth the value of the number of Gen 1 collections and a Gen 1 collection should be one-tenth the value of the number of Gen 0 collections.
- Where applicable, there should be no SQL Server deadlocks.
- Where applicable, the SQL Server **Lock Waits/sec** performance counter should be zero.
- Where applicable, there should be no SQL Server exceptions.

Table 26 lists the relevant performance counters and their respective thresholds.

Table 26: Performance Counters Used in Performance and Stress Testing

Area	Performance counter name	Threshold
Processor	**% Processor Time**	This value should be close to 100 percent when the load simulation saturates the application block. This usually happens with 10 virtual clients because there are no think times. Any contention or external dependency may cause this value to be too low. For scalability tests, this metric is used in conjunction with end-to-end transactions per second.
Processor	**% Privileged Time**	This value should be no higher than 20 percent to 25 percent.
Process (w3wp)	**Handle Count**	This value should remain stable during stress tests.

Area	Performance counter name	Threshold
Process (w3wp)	**Private Bytes**	This value should remain stable during stress tests. Small amounts of memory, less than 50 MB to 60 MB, are reserved for the Caching Application Block when the test adds many items to the cache. In the Caching Application Block, the number of private bytes can add up to almost all the available memory if items are added to the cache and there is no expiration policy and no scavenging. If there is a memory leak, this performance counter increases.
Process (w3wp)	**Virtual Bytes**	The number of virtual bytes should not exceed the number of private bytes by more than 800 MB. The difference between the number of virtual bytes and private bytes is because it is possible to commit more memory than the process actually owns. Virtual bytes represent virtual address space. When the number of virtual bytes is close to the 2 GB limit, memory fragmentation causes the system to slow down. Eventually, an **OutofMemoryException** exception occurs.

continued

Area	Performance counter name	Threshold
Process (w3wp)	**Thread Count**	This value should remain stable during stress tests. The thread count of a worker process should be no more than 75 + ((**maxWorkerThreads** + **maxIoThreads**) * # of CPUs), where # of CPUs is the value in the Machine.config file. The Caching Application Block uses its own background scheduler for expiration and scavenging. The value of this performance counter should be constant while the scavenging and expiration scenarios are being stress tested. This means that the scheduler does not have any memory leaks.
Process (w3wp)	**Elapsed Time**	This performance counter keeps track of the elapsed time, in seconds, that the process has been running. In other words, it measures an application block's availability. The value should remain constant during the stress tests. If the value is less than the length of the test, an ASP.NET recycle occurred, which may indicate a problem.
Memory	**Pool Paged Bytes**	This value should be less than 90 MB and remain stable during stress tests. Larger values can indicate that the application block is unnecessarily allocating system objects, such as security tokens or thread handles. The Enterprise Library uses instrumentation that allocates unmanaged handles and performance counters objects. In particular, you should monitor this performance counter during the Cryptography Application Block stress tests.

Area	Performance counter name	Threshold
Memory	**Pool Nonpaged Bytes**	This is the number of bytes in the nonpaged pool, an area of system memory for objects that cannot be written to disk, but must remain in physical memory as long as they are allocated. This value should remain stable during stress tests.
Memory	**Committed Bytes**	This value is the amount of virtual memory that is supplied either by RAM or by a secondary storage device. You can use it to calculate the contention index. This index is the ratio of **CommittedBytes**/Total RAM. This ratio should be less that 2:1. Higher ratios mean that there is too much paging. The only scenario where a larger ratio is acceptable is when the Caching Application Block adds unique keys to the cache without using any scavenging or expiration policies.
Memory	**Pages/sec**	This value should be zero. It represents hard page faults, which indicates that there is memory starvation because of a memory leak.
Memory	**Available MBytes**	This value should be stable during the stress tests. Low values for SQL Server is normal because it allocates as much memory as it can for caching. In general, the value should be at least 20 MB for IIS when it hosts Enterprise Library. The exception is when the Caching Application Block adds unique keys to the cache without using scavenging or expiration policies. In this case, the value is dependent on the number of items you add to the cache.

continued

Area	Performance counter name	Threshold
.NET CLR Memory (w3wp)	**% Time in GC**	This is the percentage of time spent performing the last garbage collection. An average value of 5 percent or less is ideal because all threads are suspended during a garbage collection. For Enterprise Library, this number should be from 5 percent to 9 percent. The most common cause of a high value is that the application block is allocating memory for too many objects for each request.
.NET CLR Memory (w3wp)	**# Gen 0 Collections** **# Gen 1 Collections** **# Gen 2 Collections**	These counters indicate the number of times the generation (Gen) *n* objects are garbage-collected from the start of the application. The number of Gen 0 collections should be 10 times higher than Gen 1 collections, which should be 10 times higher than Generation 2 collections. In other words, a Gen 1 collection should be one-tenth the value of the number of Gen 0 collections, and so on.
.NET CLR Memory (w3wp)	**# GC Handles**	This value should be stable during stress tests. It is the current number of garbage collection handles in use.

Area	Performance counter name	Threshold
.NET CLR Memory (w3wp)	**# Bytes in all Heaps**	This value should be stable during the stress tests. It indicates the current memory allocated in bytes on the garbage collection heaps. If there is a memory leak within managed memory, this counter increases along with the **Process (w3wp)\ Private Bytes** performance counter. If there is a memory leak within unmanaged memory, the **# Bytes in all Heaps** performance counter remains stable, while the **Private Bytes** performance counter increases.
.NET CLR Locks and Threads	**Contention Rate / sec**	This number should be less than 9/sec. For Enterprise Library, higher numbers indicate poor scalability, particularly when running the code on computers with multiple processors or to 64-bit computers. A high contention rate indicates poor scalability and low throughput on multi-processor computers.
Logical Disk (drive)	**Current Disk Queue Length**	This value should be no more than 2 requests per spindle, unless there is paging.
Logical Disk (drive)	**Avg. Disk Read Queue Length**	This value should be no more than 2 requests per spindle, unless there is paging.
Logical Disk (_Total)	**Avg. Disk Write Queue Length**	This value should be no more than 2 requests per spindle, unless there is paging.
Logical Disk (drive)	**Avg. Disk/Sec Write** **Avg. Disk/Sec Read**	Both of these counters should have values of less than 6 ms.

continued

Area	Performance counter name	Threshold
Logical Disk (drive)	**% Idle Time**	This value should be higher than 90 percent. Values lower indicate that either there is paging or the application block writes to the SQL Server database. This value should be high for computers that host Enterprise Library.
System	**Threads**	This value should be stable and read in conjunction with the **Process\ThreadCount** performance counter.
System	**Context Switches/sec**	This is the rate of switches from one thread to another. The value should not be higher than System\Threads * 20. Higher values indicate that there is either contention or context switching from kernel mode to user mode. The **ContextSwitchesPerSec** performance counter should linearly increase as the throughput, load, and the number of CPUs increases. If it increases exponentially, there is a problem. For Enterprise Library, this number should be low in all test cases for all application blocks.
.NET CLR Security	**% Time in RT Checks**	This value should be less than 7 percent. It is the percentage of elapsed time spent performing runtime code access security checks since the last sample.
.NET CLR Exceptions (w3wp)	**# Exceptions Thrown**	This value should be zero, except for the Exception Handling Application Block, when it should be ignored. This application block causes exceptions that are recorded in the performance monitor.
SqlServer:AccessMethods	**Full Scans/sec**	This value should be less than 3.

Area	Performance counter name	Threshold
SqlServer:AccessMethods	**Full Page Splits/sec**	This value should be less than 10.
SqlServer:Buffer Manager	**Buffer Cache Hit Ratio**	This value should be higher than 80 percent.
SqlServer:Buffer Manager	**Lazy Writes/sec**	This value should be close to zero.
SqlServer:Cache Manager	**Cache Hit Ratio**	This value should be higher than 80 percent.
SqlServer:Locks	**Average Wait Time**	This value should be less than 2 ms and remain stable during the stress tests.
SqlServer:Locks	**Number of Deadlocks/sec**	This value should be zero.
SqlServer:Memory Manager	**Total Server Memory**	This value should be less than 75 percent.
SqlServer:Memory Manager	**Lock Blocks**	This value should be stable during the stress tests.

Understanding and Measuring Transaction Times

This section discusses the threshold levels for the different transaction metrics. The threshold defines the highest value that the transaction metric can be and still be acceptable. This section also demonstrates how to measure transactions. Here are the acceptable threshold values:

- **Transaction Times Threshold**. The application block's threshold should be no more than 12 percent to 32 percent higher than the baseline transaction times metric.
- **TransactionsPerSecond**. The application block's threshold metric should be no more than 12 percent to 32 percent higher than the baseline's transactions per second metric. The total transactions or total hits can be proved by reading **Web Service \Total Anonymous Users** performance counter after recycling IIS in between test cases.
- **Total Transactions**. The application block's threshold metric should be no more than 12 percent to 32 percent higher than the baseline's total transactions metric. To find the total transactions (or total hits), examine the **Web Service\Total Anonymous Users** performance counter. You must stop and restart IIS between test cases because the counter starts to count when the service starts.

A transaction in a Web test is like a timer. You can encapsulate a set of actions in a transaction. You can think of a typical transaction as starting a timer, requesting a page, requesting another page, and then ending the timer. This series of actions, from start to end, constitutes a transaction. You can make more granular measurements of

transactions than are possible with performance counters. To do this, place a **BeginTransaction** method before the first action. Place an **EndTransaction** method after the last action. After you save the script, you will see an icon in the Web Test Editor.

The following code example shows how to add transactions to the Logging Application Block test harness.

```
this.BeginTransaction("LogWriter");
            yield return request1;
WebTestRequest request2 = new WebTestRequest("http://10.3.15.50/EntLibtests/Log-
ging.asp);
            request2.Method = "POST";
            FormPostHttpBody request2Body = new FormPostHttpBody();
            request2Body.FormPostParameters.Add("__VIEWSTATE",
 ViewState);            request2Body.FormPostParameters.Add("DropDownList1","Even
tlogCatetory" );
request2Body.FormPostParameters.Add("Button6", "LogCategory");
request2Body.FormPostParameters.Add("__EVENTVALIDATION", EventValidation);
            request2.Body = request2Body;
            this.EndTransaction("LogWriter");
            yield return request2;
```

Testing for Scalability

Scalability tests determine whether adding more system resources, such as CPUs or computers, or adding any external resources, such as SQL Server or disk arrays, can increase throughput without degrading performance. They also identify any locking problems and contention for resources that may not be found during the performance tests. Contentions problems are also called bottlenecks.

Identifying Bottlenecks

There are two approaches to test for bottlenecks:

- You can increase the load beyond the saturation point to see whether the throughput remains stable.
- You can perform the tests with computers that have 2 to 4 processors. You should do this with both 32-bit computers and 64-bit computers.

The Enterprise Library scalability tests use the second approach. The main goal is to determine that contention in the code does not interfere with the application block's throughput. Test this to see if the appropriate metrics remain stable when the loads exceed the specified constraints on the system resources.

Hardware Configurations

Table 27 lists the hardware configuration for the Enterprise Library scalability tests.

Table 27: Hardware Configurations for Scalability Tests

	Server A	Server B	Server C	Server D	Server E
Architecture	32 bit	32 bit	64 bit	64 bit	64 bit
Number of processors	2	4	2	4	4
Processor speed	2,790 Mhz	1,903 Mhz	1,804 Mhz	1,804 Mhz	1,804 Mhz
Physical Memory	1024 MB	4048 MB	3072 MB	3072 MB	5098 MB
Manufacturer/ Model	HP DL 380 PROLIANT G3	HP DL 560 PROLIANT G1	HP DL 385 PROLIANT G1	HP DL 385 PROLIANT G1	HP DL 385 PROLIANT G1
Characteristics	Intel Xeon Prestonia processors (2.8 GHz) with a 512 KB level 3 processor cache and Hyper Threading Technology	Intel (1.8 GHz) with a 2 MB level 3 processor cache and Hyper Threading Technology	1 MB level 3 processor cache and Hyper Threading Technology	1 MB level 3 processor cache and Hyper Threading Technology	1 MB level 3 processor cache and Hyper Threading Technology

Scalability Test Scenarios and Results

The scalability tests use scenarios for the Logging Application Block, the Caching Application Block, and the Data Application Block. The scenarios are:

- **Logging Application Block**. Write a message to the event log.
- **Caching Application Block**. Add an item to the in-memory cache. (Each execution thread generates a unique key for every item.)
- **Data Application Block**. Execute a stored procedure that returns one row from the Northwind sample database.

Each scenario simulated 1, 10, 50, and 150 users.

Analysis of Logging Application Block Scalability Test

Figure 6 is a graph of the test results. It shows the results for both the 32-bit computers and 64-bit computers, each with 2 processors and then 4 processors.

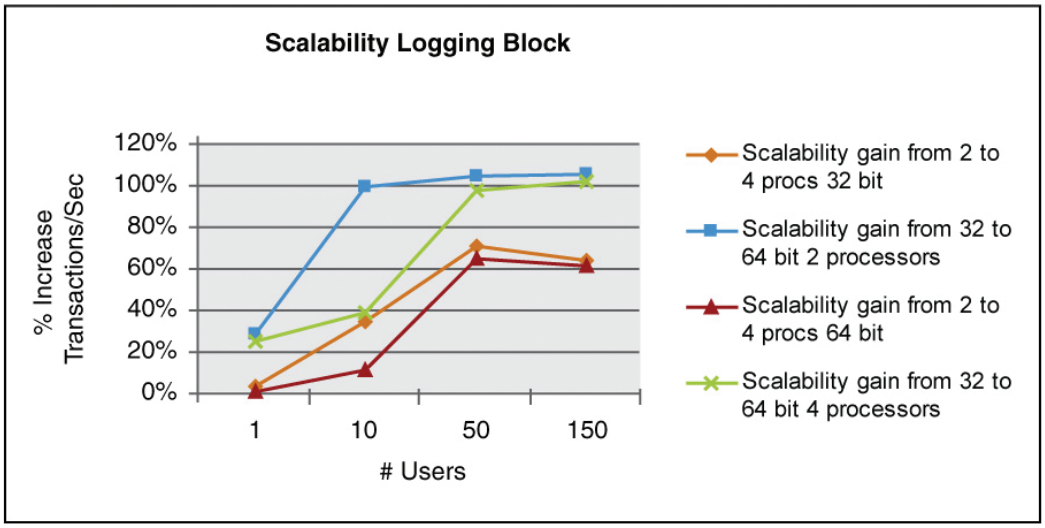

Figure 6 *Scalability results for Logging Application Block*

The servers are always CPU bound. Other than the processors, there are no other dependencies that can cause bottlenecks. The change in memory size between the 2-processor configuration and 4-processor configurations (for both the 32-bit computers and 64-bit computers) did not affect the relevant measurements.

Here is a summary of the results:

- The 64-bit computers outperform the 32-bit computers by 45 percent for the transactions per second measurement and for the total transactions measurement.
- For all cases shown in Figure 6, there should be no I/O activity when the **% Disk Time** performance counter is less than 7 percent.
- The 32-bit computers with 2 processors could not support more than 10 concurrent users. The 32-bit computers with 4 processors could not support more than 50 concurrent users. The 64-bit computers with 2 processors could not support more than 10 concurrent users. The 64-bit computers with 4 processors could not support more than 50 concurrent users. Scalability increases as more concurrent users are added to the test mix. Scalability increases are greatest when moving from a 2-processor computer to a 4-processor computer. Moving from a 32-bit computer to a 64-bit computer yields smaller increases.
- The servers' working sets always require less than 40 MB for all configurations. The working set of an application is the set of memory pages currently available in RAM.

- Measurements of the memory required by the working sets and the values of the **CLR % GC** performance counter indicate that memory size did not cause any bottlenecks during the scalability tests. All bottlenecks were caused by the limit to the number of users the CPUs could support.
- Processor queue lengths began to lengthen as the number of concurrent users increased and the CPUs reached their limits. Values for processor queue lengths were initially zero and increased to between 7 ms and 10 ms after the CPUs could no longer support any more users.
- The Logging Application Block has good scalability. This is largely due to the low number of contentions per second. The value of the **CLR Contention rate/sec** performance counter was approximately 8 per second when the CPU was used at its full capacity (close to 100 percent) and 2 percent to 3 percent when the CPU was below its saturation levels. These are low numbers that demonstrate that there were very few contentions during the scalability tests.

Analysis of Data Access Application Block Scalability Test

Figure 7 is a graph of the test results. It shows the results for both the 32-bit computers and 64-bit computers, each with 2 processors and then 4 processors.

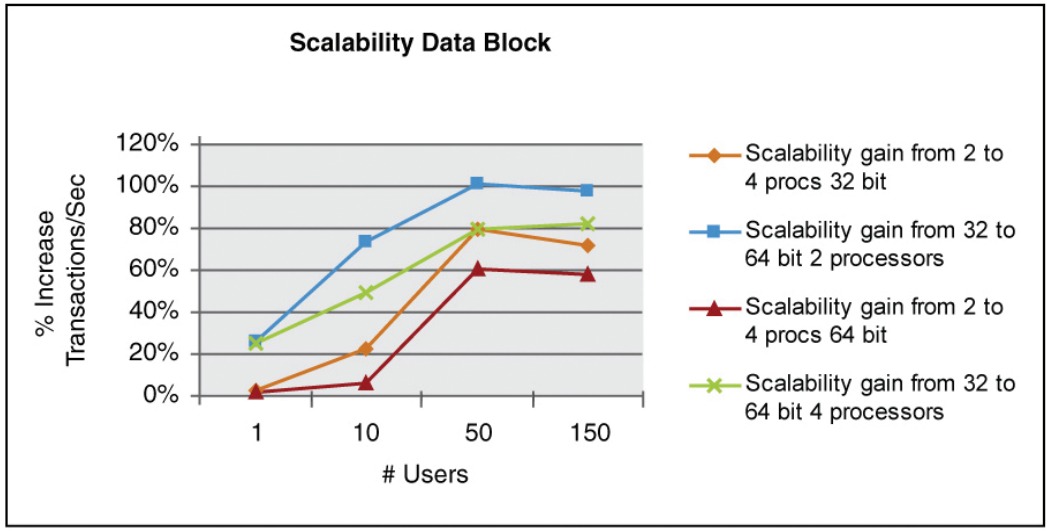

Figure 7 *Scalability results for Data Access Application Block*

The tests read one row of a database for each test cycle. There are no writes because write operations can cause an increase in SQL Server I/O activity. In turn, this dependency may affect the percentage of processor time that the application block can use. The test should be performed without any dependencies that would prevent the CPU from being used to its full capacity.

In the scalability tests for the Data Access Application Block, the processor queue lengths began to lengthen as the number of concurrent users increased and the CPUs reached their limits. Values for processor queue lengths were initially zero and increased to between 7 ms and 10 ms after the CPUs could no longer support any more users.

The servers are always CPU bound. Other than the processors, there are no other dependencies that can cause bottlenecks. The change in memory size between the 2 processor configurations and 4 processor configurations (for both the 32-bit computers and 64-bit computers) did not affect the relevant measurements.

Here is a summary of the results:

- The 64-bit computers outperform the 32-bit computers by 45 percent for the transactions per second measurement and for the total transactions measurement.
- For all cases shown in Figure 7, there should be no I/O activity when the **% Disk Time** counter is less than 7 percent.
- The 32-bit computers with 2 processors could not support more than 10 concurrent users. The 32-bit computers with 4 processors could not support more than 50 concurrent users. The 64-bit computers with 2 processors could not support more than 10 concurrent users. The 64-bit computers with 4 processors could not support more than 50 concurrent users. Scalability increases as the number of concurrent users were added to the test mix when bigger deltas were observed between 2 processors and 4 processors (32-bit computers and 64-bit computers).
- The servers' working sets always require less than 40 MB for all configurations.
- Measurements of the memory required by the working sets and the values of the **CLR % GC** performance counter indicate that memory size did not cause any bottlenecks during the scalability tests. All bottlenecks were caused by the limit to the number of users the CPUs could support.
- The Logging Application Block has good scalability. This is largely due to the low number of contentions per second. The value of the **% Contention rate/sec** performance counter was approximately 8 per second with saturation of CPU in all cases and between 2 percent and 3 percent below saturation levels.

Analysis of Caching Application Block Scalability Test

Figure 8 is a graph of the test results. It shows the results for both the 32-bit computers and 64-bit computers, each with 2 processors and then 4 processors.

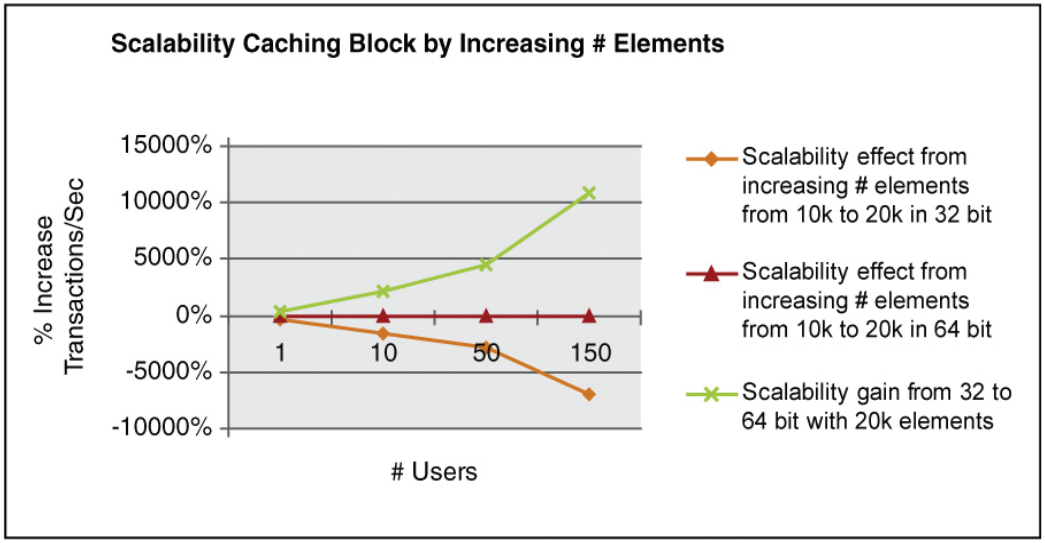

Figure 8 *Scalability results for Caching Application Block*

The scalability test for the Caching Application Block test is to insert items into the in-memory cache. A cached item is defined as a key/value pair. The test harness generates a random key value between 1 and 20,000. Each key is 100 KB. The 64-bit CLR can only allocate up to1 GB for a given object but it can allocate as many objects smaller than 1 GB as there is available memory.

The test cases are divided into those tests that add 10 KB values to the cache and those that add 20 KB items to the cache. In either case, the key is 100 KB. A test may use the same key in different transactions.

The tests ran on both the 32-bit computers and 64-bit computers with 4 processors. The purpose was to test the scalability of the address space for both types of computers. The reason that there were no tests on computers with 2 processors is that the number of processors does not affect how well the Caching Application Block uses memory.

Results showed that, for the tests that used10 KB values, the number of transactions on the 64-bit computers were 30 percent to 60 percent higher than on the 32-bit computers. The actual percentage depends on the number of concurrent users consuming processor cycles.

Before the 64-bit processor reached its limits, the percentages increased both for the total number of transactions and throughput. The **% GC** performance counter was less than 15 percent and the **Contention/Sec** performance counter was less than 9/sec. At the heaviest load, it was possible to use approximately 98 percent of both the 32-bit computers and 64-bit computers.

Results showed that, for the tests that used 20 KB values, the 32-bit computer started paging because there was insufficient memory to hold all the data. The **Pages/Sec** performance counter went up between 160 pages/sec and 170 pages/sec. The **% Processor Time** performance counter fell far below capacity. Generally, it was no higher than 34 percent. The **Private Bytes** performance counter for the ASP.NET w3wp.exe worker process was close to 1.7 GB.

For the 64-bit computer, it was possible to use values that were as large as 30 KB while still maintaining acceptable levels for both response times and transactions per second. During the heaviest load, the **Private Bytes** performance counter was at approximately 5 GB. There was little contention. The CLR contention rate/sec indicated values less than 8 and the **Pages/sec** performance counter was zero.

Measuring Initialization Costs

The initialization cost is the amount of system resources and time necessary to prepare an application block so that it can begin to execute requests. Subsequent costs will be much less because the necessary code and data structures are now cached. This measurement is not relevant for Web applications because the first hit (the initialization process) is not considered an important factor in terms of performance. However, the measurement does matter for smart client applications because clients frequently stop and restart the application. This means that clients will repeatedly initialize the code variables and data structures.

The .NET Framework **XmlSerializer** class, which Enterprise Library uses, illustrates this. The initialization cost is high because the class must be compiled. Afterward, the compiled version is in the cache.

For situations where the initialization cost is important, you should measure the times it takes to execute the most expensive execution paths. To do this, write a test console application that measures and displays the initialization costs. The following code shows how to do this for the **XmlSerializer** class.

```
public class OrderedItem
{
    public string Nationality;
    public string Description;
    public decimal UnitPrice;
    public int Quantity;
}

class Class1
{
    [STAThread]
    static void Main(string[] args)
    {

        for(int i=0;i<=10;i++)
```

```
        {

            int start = Environment.TickCount;
serializer = new XmlSerializer(typeof(OrderedItem),"ns");
            int stop = Environment.TickCount;
        Console.WriteLine(" Time create serializer: {0}
milliseconds", stop - start);
        }
}
}
```

If you examine the Visual Studio Team System profiler, you see that although the code creates the **XmlSerializer** object 11 times, the program incurs the largest costs when it first compiles the types inside the **XmlSerializer** object. After this, subsequent calls use the compiled version in the cache. Table 28 lists the information from the profiler.

Table 28: Profiler Information

Method name	Time in ms	Number of calls
System.Xml.Serialization.XmlSerializer..ctor	795.67	11
System.Xml.Serialization.XmlSerializer.GenerateTempAssembly	645.37	1
System.Xml.Serialization.TempAssembly..ctor	644.42	1
System.Xml.Serialization.TempAssembly.GenerateAssembly	414.6	1
System.Xml.Serialization.Compiler.Compile	281.72	1

Extrapolating Workload Profiles

It is possible to extrapolate workload characterizations for the load tests, even though you do not know the actual application that will host the application blocks or the systems on which the application will run.

Defining a load test is one of the steps in writing a performance test plan specification. The load test runs concurrent scenarios and records a system's behavior. You generate a test load by creating virtual users and a simulation that accurately reflects how actual users use the application.

To characterize a performance test workload, you should associate the number of concurrent users with the think times that occur between iterations.

When you create a load test for an application, you often have no data about how many concurrent users there actually will be or the type of computer or network you need to consider. For example, your load test may be for an application that will actually run on a single computer with a processor and a disk subsystem or it may run over an intranet. However, you can use Little's Law to extrapolate from a given performance specification or goal and calculate a realistic number of concurrent users.

Little's Law says that the average number of users (which is U) in a stable system (over some time interval) is equal to their average arrival rate (which is R), multiplied by their average time in the system (which is T). Expressed as an equation, this is U = R * T.

For a given system (whether a single computer or a complex network), you can measure the throughput of a system, which is the same as the average arrival rate, by dividing the number of users by the time they spend using the system This can be expressed as R = U / T.

Assume that users will wait some typical interval of time, named T_o, between requests. This is the think time. Because U = R * T, the number of users in think time, U_o, will be $U_o = R * T_o$. Therefore, the total number of users (U_t) is $U_t = U + U_0 = R * T + R * T_o = R(T + T_o)$. So $U_t = R(T + T_o)$. Finally, $R = U_t / (T + T_o)$.

As a concrete example, assume that there is an application that uses Enterprise Library. The application has 200 users who generate 1,600 requests in 15 minutes. The average response time is 2 seconds. If $T_0 = (U / R) – T$, the average think time is (200 / 1.77) – 2) or 110.2 seconds, where 1.77 is the average arrival rate, or the number of requests per second.

To calculate the number of users without including think times, you can apply the equivalent ways of specifying R, where R = U / T and $R = U_t / (T+T_0)$. Therefore, $U / T = U_t / T + T_0$. In the example, this means that U / 2 = 200/ (2+110). Simplifying, 2U +110U = 400. This means that there are 4 users when there are no think times. You can use the same technique to extrapolate any workload as long as you have a production profile for the application or a performance test goal. (A production profile models the types and numbers of concurrent operations an application should be able to support.)

Debugging Memory Leaks

This section discusses how to find and troubleshoot memory leaks. It assumes that you have the WinDbg debugger and the Son of Strike (SOS) extension.

After you use the performance counters to decide that a particular application block has a memory leak, modify the test harness to call **GC.Collect**; which forces a garbage collection. This prevents the debugger from producing false positives. In this case, false positives mean memory that seems to be allocated when it really is not because the garbage collector has already reclaimed it. The following example shows the Cryptography Application Block's modified test harness.

```
protected void CryptoHash_Click(object sender, EventArgs e)
    {
        byte[] b = new byte[Int32.Parse(TextBox1.Text)];
        byte[] hs=Cryptographer.CreateHash("HMACSHA1", b);
       Cryptographer.CompareHash("HMACSHA1", b, hs);
```

```
        HashProviderFactory factory = new HashProviderFactory();
        IHashProvider hashProvider = factory.Create("hashprovider");
        hashProvider.CreateHash(b);
        System.GC.Collect();

    }
```

Next, run the test again until the performance counters register the suspicious behavior. You can now use WinDbg to find and troubleshoot the memory leak.

► To use WinDbg

1. On the taskbar, click Start, point to All Programs, point to Debugging Tools for Windows, and then click WinDbg. WinDbg starts.
2. Because ASP.NET hosts the application block, you must attach the debugger to w3wp.exe. On the File menu, click Attach to Process. The Attach to Process list box appears. Click the By Executable option button. Click w3wp.exe. Click OK.

After performing these steps, load the SOS extension to WinDbg. Open a command window in the debugger and type:

!loadby sos mscorwks

Next, look at the heap. Type:

!dumpheap -stat

This command reports all the managed objects that are currently in the heap, sorted by number of allocations and size. The MT column, where MT stands for Method Table, lists the pointers to the tables that describe the objects. The Count column lists the number of objects that exist in the heap for each type of object. The TotalSize column lists the amount of memory that is being used by any one type of object. The last column is the fully typed name of the object.

```
MT          Count          TotalSize     Class Name
029b0800    11502          138024        Microsoft.Practices.EnterpriseLibrary.
Caching.Expirations.NeverExpired
79124670      193          229708       System.Char[]
79124228    12721          340908       System.Object[]
029b06d4    11501          460040 Microsoft.Practices.EnterpriseLibrary.Caching.
CacheItem
790fa3e0 20210000        93908800       System.String
791242ec      140        14656992       System.Collections.Hashtable+bucket[]
```

The suspicious entries have the largest counts and the largest sizes.

Assume that the number of **System.String** objects appears suspicious. The memory dump says that there are 20210000 of those objects. This appears too high.

You need to look at MT location 790fa3e0. Type:

!dumpheap -mt 790fa3e0

The following information appears in the command window:

```
1e321b38 791242ec      144
1e322194 791242ec      144
1e322374 791242ec      144
1e322578 791242ec      144
1e322710 791242ec      144
.
.
.
```

The first column reports the memory location of the object. The second column lists the method table. The third column lists the size in bytes of the object. Scroll down to view all the objects.

The problem you are investigating is that objects that should be released are not being released as expected. This means that there must be reference to the objects somewhere. The following command will help you to see what objects hold references to the objects in question. Type:

!gcroot 1e322710

The screen displays the following information:

```
eax:Root:1e3226d8(System.Collections.Hashtable)->
1e322710(System.Collections.Hashtable+bucket[])
ecx:Root:1e3226d8(System.Collections.Hashtable)->
1e322710(System.Collections.Hashtable+bucket[])
```

The root trace points to objects that allocate the suspicious strings. You should examine the methods in your code that perform this function and see if the resources are not being released when they should be.

If you need to investigate memory leaks that involve **GCHandles** objects, type:

!gchandles

This command reports the handle statistics and those managed objects that are held in the GCHandles table. The information is in the same format as the heap display.

Using the Test Cases

This chapter discusses some of the test cases that uncovered bugs and issues in Enterprise Library – January 2006. The test cases cover three areas:

- Performance
- Security
- Functionality

For each of these test cases, there is:

- A problem, which is the problem the test case uncovered.
- A proposed solution, which is the proposed solution to the problem.
- A verification, which is how the solution was tested to see whether it solved the problem.

The test cases are representative of the sorts of bugs and issues you might see in your own code.

Performance Testing

The goals of performance testing are:

- **To verify that the application block meets the performance requirements while staying within the budgeted constraints on system resources**. The performance requirements can include different measurements, such as the time it takes to complete a particular scenario (this is known as the response time) or the number of concurrent or simultaneous requests that can be satisfied for a particular operation within a specific response time. Examples of system resources are CPU time, memory, disk I/O, and network I/O.
- **To identify the bottlenecks in the application block's code**. The bottlenecks can be caused by issues such as memory leaks, slow response times, or resource contention under load.

The performance tests measure cost in terms of system resources. The unit of measurement is transactions per second. The baseline is the performance of Enterprise Library version 1.1 and .NET Framework code that performs similar functions. The cost is expressed as the percentage of overhead. This is the difference, in transactions per second, between the baseline and Enterprise Library – January 2006, divided by the baseline transactions per second.

General Performance Tests

All the application blocks were tested to confirm that they meet the performance requirements. Some tests were specific for particular application blocks and some tests applied to all the application blocks. A good example of a performance test that applied to all the application blocks was determining the cost of creating and binding instrumentation providers and listeners. The following example is for the Cryptography Application Block.

Determining the Cost of Creating and Binding Instrumentation Objects

This performance test determines the cost, in terms of system resources, of creating the instrumentation objects. The instrumentation objects include WMI (Windows Management Instrumentation), performance counters, and event logs. The cost of creating instrumentation objects for the Cryptography Application Block in Enterprise Library – January 2006 was compared with the cost of creating instrumentation objects for the Cryptography Application Block in Enterprise Library version 1.1. The goal was that the overhead in Enterprise Library – January 2006 be within acceptable limits when compared with Enterprise Library version 1.1.

Test Cases

Two test cases were used to measure the cost of creating the instrumentation objects for the Cryptography Application Block. One case used the **SymmetricCryptoProviderFactory** class to create a symmetric algorithm provider with the instrumentation enabled. The other test case used the **SymmetricCryptoProviderFactory** class to create a symmetric algorithm provider with the instrumentation disabled. The difference between the two cases provided the cost of creating and binding the instrumentation objects. Similar test cases were used with the hash providers.

Problems

The results from the performance tests showed that the cost of creating instrumentation providers for the Cryptography Application Block was higher than the allowable limits in Enterprise Library – January 2006 than in Enterprise Library version 1.1. An analysis of the application block revealed that the use of reflection to create the instrumentation objects was responsible for the performance overhead. To properly bind the classes, reflection-based binding requires that the **SymmetricCryptoProviderFactory** class reflect through all the methods in the **SymmetricAlgorithmInstrumentationListener** class and all the events in **SymmetricAlgorithmInstrumentationProvider** class.

Solution

The solution is to create an explicit binder that implements the **IExplicitInstrumentationBinder.Bind** method. The **Bind** method explicitly binds the instrumentation provider's events with the instrumentation listener's methods, which avoids the need to

use reflection. The following code shows the **SymmetricAlgorithmInstrumentationBinder** class, which implements the **IExplicitInstrumentationBinder** interface.

```
public class SymmetricAlgorithmInstrumentationBinder : IExplicitInstrumentation-
Binder
{
    public void Bind(object source, object listener)
    {
        SymmetricAlgorithmInstrumentationListener castedListener = (SymmetricAlgor
ithmInstrumentationListener)listener;
        SymmetricAlgorithmInstrumentationProvider castedProvider = (SymmetricAlgor
ithmInstrumentationProvider)source;
        castedProvider.cyptographicOperationFailed += castedListener.Cyptographic-
OperationFailed;
        castedProvider.symmetricDecryptionPerformed+= castedListener.SymmetricDe-
cryptionPerformed;
        castedProvider.symmetricEncryptionPerformed += castedListener.SymmetricEn-
cryptionPerformed;
        }
}
```

Verification

Running the two test cases again after applying the preceding code verified that the cost of creating the instrumentation objects was within acceptable limits.

Determining the Cost of Creating Objects Using the FileConfigurationSource Class

Another example of a general performance test is the cost of creating objects with data contained in a configuration file as opposed to data that is in memory. The following example uses the Data Access Application Block to determine the cost of creating a domain object from data in the file configuration source. To pass the test, the cost of creating domain objects with the **FileConfigurationSource** class could be slightly more than when using the **SystemConfigurationSource** class.

Test Cases

Two test cases determined the cost of creating domain objects with the **FileConfigurationSource** class. One test created a **Database** object with the **DatabaseProviderFactory** class and used the **FileConfigurationSource** class to retrieve the configuration data from a file. The other test did the same but used the **SystemConfigurationSource** class to retrieve the data from memory. Comparing the results showed whether the application block's performance was acceptable.

Problems

The results from the performance tests showed that it was more expensive to create a **Database** object with the **FileConfigurationSource** class than to create one with the **SystemConfigurationSource** class. An analysis of the application block revealed that the **FileConfigurationSource** class read the appropriate sections from the configuration file each time it created a **Database** object. This increased the number of I/O operations and impacted the application block's performance.

Solution

The **SystemConfigurationSource** class uses the .NET Framework **System.Configuration.ConfigurationManager** class, which caches the **Configuration** object. Caching the object reduces the number of I/O operations. Similarly, the **FileConfigurationSource** class could also cache the **Configuration** object, which should substantially improve the application block's performance. However, if the file changes, the data in the cache is out of date. To address this issue, the **FileConfigurationSource** class was modified to implement a notification handler that watches the configuration file to see if it changes. If it does, the handler notifies the configuration source to update the cache.

The following code shows the configuration object is cached and retrieved from the cache.

```
private System.Configuration.Configuration GetConfiguration()
{
    if (cachedConfiguration == null)
    {
        lock (cachedConfigurationLock)
        {
            if (cachedConfiguration == null)
            {
                cachedConfiguration = ConfigurationManager.OpenMappedExeConfiguration
(fileMap,ConfigurationUserLevel.None);
            }
        }
    }
return cachedConfiguration;
}
```

The following code shows the method that the notification handler invokes to update the cache.

```
internal void UpdateCache()
{
    System.Configuration.Configuration newConfiguration
    = ConfigurationManager.OpenMappedExeConfiguration(fileMap,ConfigurationUserLevel.
None);
    lock (cachedConfigurationLock)
    {
        cachedConfiguration = newConfiguration;
    }
}
```

Verification

Running the two test cases again after applying the preceding code examples verified that the cost of creating a domain object with the **FileConfigurationSource** class was within acceptable limits.

Specific Performance Tests

Specific aspects of each application block were also tested to see if their performance was acceptable. This section discusses performance issues that were found in the Data Access Application Block, the Logging Application Block, and the Cryptography Application Block.

Testing the Logging Application Block

The Logging Application Block was tested to determine the cost of logging messages to a database trace listener. To pass the test, the cost of logging to a database trace listener should be scalable. If the number of transactions per second decreases as the number of requests increases, there is a bottleneck.

Test Case

To run the test case, the application block was first configured to use a database trace listener to log messages to a database. The test case logged messages to the database with the **Logger.Write** method. A performance test tool was used to analyze the results.

Problem

The results from the performance test showed that the transactions per second decreased when the numbers of users increased. This decrease occurred because, with a large number of users, there was significant contention for access to the database trace listener.

This contention exists because the **LogSource** class enumerates through all the trace listeners that the application block is configured to use and calls each trace listener's **TraceData** method. Before calling the method, the **LogSource** class first checks to see whether the trace listeners are thread safe by calling each trace listener's **IsThreadSafe** property. It locks a trace listener if it is not thread safe. The **FormattedDatabaseTraceListener** class derives from **FormattedTraceListenerBase**, which has the **IsThreadSafe** property set to **false**. This means that the **FormattedDatabaseTraceListener** class is locked for every call to its **TraceData** method. This is why there is contention for access to the database trace listener.

Solution

The **FormattedDatabaseTraceListener** class is already thread safe, so there is no need to lock the database trace listener. The solution is to set the **IsThreadSafe** property to **true** in the **FormattedTraceListenerBase** class. This is the base class of the **FormattedDatabaseTraceListener** class. The following is the modified code.

```
public abstract class FormattedTraceListenerBase : TraceListener, IInstrumenta-
tionEventProvider
{
  public override bool IsThreadSafe
  {
   get
   {
    return true;
   }
  }
}
```

Verification

Running the test case again after applying the preceding code verified that the database trace listener was scalable.

Testing the Cryptography Application Block

The Cryptography Application Block was tested to determine the cost of reading a symmetric key from a key file. To pass the test, the cost could be slightly more than when using Enterprise Library version 1.1 or when using one of the symmetric algorithm classes in the .NET Framework 2.0 **System.Security.Cryptography** namespace.

Test Case

To run the test case, a key file was first generated with the Enterprise Library Configuration Console. The data protection scope could be either **current user** or **local machine**. The test case called the application block's **KeyManager.Read** method to read the symmetric key.

Problems

The test case revealed two problems. The first problem was that reading the key from the file for each request was an expensive I/O operation that caused poor performance. The second problem was that to read keys from an input stream, the application block used the .NET Framework **FileStream** class. It constructed **FileStream** objects with read/write access to the file. When concurrent users tried to access the file, the .NET framework threw a **System.IO.IOException** exception. The following error message displayed:

The process cannot access the file because it is being used by another process.

This is the code that caused the problems.

```
public static ProtectedKey Read(string protectedKeyFileName, DataProtectionScope
dpapiProtectionScope)
{
    using (FileStream stream = new FileStream(protectedKeyFileName, FileMode.
Open))
    {
        return Read(stream, dpapiProtectionScope);
    }
}
```

Solution

To solve the first problem, the **Read** method was modified so that it read the symmetric key only once from the file and then cached the key in a static collection. Subsequent reads retrieved the file from the cache, which greatly reduced the performance overhead.

To solve the second problem, the **Read** method was modified to use constructors that included the **FileShare.Read** enumeration. This allowed multiple users to concurrently read the file. The following is the code that solved both the performance problem and the concurrency problem.

```
public static ProtectedKey Read(string protectedKeyFileName, DataProtectionScope
dpapiProtectionScope)
{
    string completeFileName = Path.GetFullPath(protectedKeyFileName);
    if (cache[completeFileName] != null)
return cache[completeFileName];
    using (FileStream stream = new FileStream(protectedKeyFileName, FileMode.Open,
FileAccess.Read, FileShare.Read))
    {
        ProtectedKey protectedKey = Read(stream, dpapiProtectionScope);
        cache[completeFileName] = protectedKey;
        return protectedKey;
    }
}
```

Verification

Running the test case again after applying the preceding code verified that the cost of reading a key from a file was within acceptable limits. In addition, concurrent users could now read the file.

Security Testing

The goals of security testing are the following:

- Identify the potential threats to the application blocks.
- Identify the vulnerabilities of the application blocks.
- Provide counter measures to these threats and vulnerabilities.

In general, threats can be classified as spoofing identity, tampering with data, repudiation, information disclosure, denial of service, and elevation of privileges. To learn more about these threats, see *Threat Modeling Web Applications* on MSDN in the Microsoft patterns & practices Developer Center.

General Security Tests

All the application blocks are required to request code access security permissions for the appropriate assemblies. Code access security allows code to be trusted to varying degrees depending on where the code originates and on other aspects of the code's identity.

Test Case

The application blocks use reflection in multiple places to create domain objects. The test case was to review the code in the **AssemblyInfo** file to determine whether the application block requests permission to use reflection. This permission is necessary for the application block to run in low-trust environments. (The **AssemblyInfo** file contains information such as attributes, files, resources, types, versioning information, and signing information for modifying an assembly's metadata.)

Problem

The test case revealed that the **ReflectionPermission** class that controls access to the metadata was not defined in the **AssemblyInfo** file. This class defines the set of permissions that are required for application block assemblies to run in low-trust environments.

When the application block runs in a low-trust environment, the system administrator must explicitly grant those permissions that allow the application block to run. The explicit permission to use reflection allows the **ObjectBuilder** subsystem to use reflection and access private class members and metadata.

Solution

The solution was to add a **ReflectionPermission** object with the necessary permissions to the AssemblyInfo file. The following code shows how to do this.

```
   [assembly: ReflectionPermission(SecurityAction.RequestMinimum, MemberAccess =
true)]
```

Verification

Examining the **AssemblyInfo** file after applying the preceding code showed that it included a **ReflectionPermission** object.

Specific Security Tests

Specific aspects of each application block were also tested to see if they followed security best practices. This section discusses security issues that were found in the Data Access Application Block and the Cryptography Application Block.

Testing the Data Access Application Block

This test case determined whether the Data Access Application Block violated security best practices by returning the password that is stored in the **ConnectionString** property to the user. It is important to ensure that this does not happen because it can give malicious users access to sensitive information.

Test Case

The test case checked to see if the application block returned the password to the user. If it did, the test case failed. The following code implemented the test case.

```
 [TestMethod]
 public void TestPasswordInConnectionStringWithDAAB()
 {
   Database db =DatabaseFactory.CreateDatabase("PasswordProtectedConnectionString
Instance");
   db.Connection.Open();
    string connectionString = db.ConnectionString;
    db.Connection.Close();
    if (connectionString.Contains("password=test"))
    {
       Assert.Fail();
    }
 }
```

Problem

The test case failed because the **Database** class exposed the **ConnectionString** property as a public member. This allowed a user to read the password. The following code caused the problem.

```
public string ConnectionString
{
 get
  {
   return this.connectionString.ToString();
  }
}
```

Solution

The **Database** class was modified so that the **ConnectionString** property was changed from a public member to a protected internal member. A new public property named **ConnectionStringWithoutCredentials** was added. This property returns

the connection string without including sensitive information such as the password. The following is the modified code.

```
protected internal string ConnectionString
{
 get
  {
   return this.connectionString.ToString();
  }
}

public string ConnectionStringWithoutCredentials
{
 get
 {
  return ConnectionStringNoCredentials;
 }
}
```

Verification

Running the test case again after modifying the code verified that the **ConnectionStringWithoutCredentials** property did not return the password.

Testing the Cryptography Application Block

This test case determined whether the Cryptography Application Block cleared a decrypted key from memory after using it to encrypt and decrypt data.

Test Case

During a code review, the **SymmetricCryptographer** and **HashCryptographer** classes were examined to see if they cleared decrypted keys from memory after using them.

Problem

The code review revealed that the **SymmeticCryptographer** and **HashCryptographer** classes did not clear decrypted keys from memory before returning the data. The following code shows the **SymmetricCryptographer** code. The code for the **HashCryptographer** class was similar.

```
public byte[] Decrypt(byte[] encryptedText)
  {
    byte[] output = null;
    byte[] data = ExtractIV(encryptedText);
        this.algorithm.Key = Key;
    using (ICryptoTransform transform = this.algorithm.CreateDecryptor())
    {
       output = Transform(transform, data);
    }
    return output;
 }
```

Solution

The solution was to zero out the unencrypted algorithm key before returning the data. The application block code was modified by adding the **CryptographyUtility.ZeroOutBytes** method. The following is the modified code.

```
public byte[] Decrypt(byte[] encryptedText)
{
byte[] output = null;
byte[] data = ExtractIV(encryptedText);
this.algorithm.Key = Key;
using (ICryptoTransform transform = this.algorithm.CreateDecryptor())
{
    output = Transform(transform, data);
}
CryptographyUtility.ZeroOutBytes(this.algorithm.Key);
return output;
    }
```

Verification

The **CryptographyUtility.ZeroOutBytes** method was examined to verify that it cleared the algorithm key, which is contained in a byte array. The method incorporates the .NET Framework **Array.Clear** method, which clears byte arrays.

```
public static void ZeroOutBytes(byte[] bytes)
{
     if (bytes == null)
     {
        return;
     }
     Array.Clear(bytes, 0, bytes.Length);
}
```

Functional Testing

The objective of the functional tests is to verify that the application block meets all of its functional requirements. This section discussions functional issues that were found in the Cryptography Application Block, the Data Access Application Block, the Exception Handling Application Block, the Logging Application Block, and the Security Application Block.

Testing the Cryptography Application Block

This test case determined whether the Cryptography Application Block would encrypt data when the data protection scope was set to **LocalMachine** and there was no entropy.

Test Case

The test case created an instance of the **DpapiCryptographer** class and passed the **DataProtectionScope** mode as **LocalMachine**. It called the **DpapiCryptographer.Encrypt** method with null entropy. It then checked to verify that the data was encrypted.

Problem

The **Encrypt** method checked to see whether the **DataProtectionScope** enumeration was set to **LocalMachine**. If it was, and there was no entropy, the method threw an exception. This is incorrect. Entropy is not a requirement for local stores in highly trusted computers that are used for server-side applications. The **DataProtectionScope.LocalMachine** setting with no entropy is actually valid. The following code caused the problem.

```
public byte[] Encrypt(byte[] plaintext)
{
  if (DataProtectionScope.LocalMachine == storeScope)
  {
     throw new InvalidOperationException(Resources.DpapiMustHaveEntropyForMa-
chineMode);
   }
   return Encrypt(plaintext, null);
}
```

Solution

To solve the problem, the code that checked for the **LocalMachine** setting when there was no entropy was removed. The following is the modified code.

```
public byte[] Encrypt(byte[] plaintext)
{

  return Encrypt(plaintext, null);
}
```

Verification

Running the modified code verified that the **Encrypt** method no longer threw an exception.

Testing the Data Access Application Block

The test case determined whether the Data Access Application Block could read configuration information from a configuration source other than a file. For the test, the configuration source was an in-memory dictionary and the configuration information was a connection string.

Test Case

The test case first created a dictionary configuration source and then added the connection string to the <connectionStrings> section. It then used the **DatabaseProviderFactory** class to create a **Database** instance. The test case passed the dictionary configuration source as a parameter.

Problem

The test case failed with the following exception:

System.Configuration.ConfigurationErrorsException: The requested database instance is not defined in configuration.

The application block used the **DatabaseConfigurationView.GetConnectionStringSettings** method to read the relevant configuration section. This class relies on the. NET Framework **ConfigurationManager** class, which only reads configuration files. The following is the code that caused the problem.

```
public ConnectionStringSettings GetConnectionStringSettings(string name)
{
    ConnectionStringSettings connectionStringSettings = ConfigurationManager.
ConnectionStrings[name];
…
}
```

Solution

The application block should be able to read configuration information from any configuration source. If the configuration information is not defined in the specified configuration source, the application block should use the **ConfigurationManager** class to look in the configuration file. This is shown in the following code.

```
public ConnectionStringSettings GetConnectionStringSettings(string name)
{
…
ConnectionStringSettings connectionStringSettings;
ConfigurationSection configSection = configurationSource.GetSection("connectionStrin
gs");
if ((configSection != null) && (configSection is ConnectionStringsSection))
{
  ConnectionStringsSection connectionStringsSection = configSection as Connection-
StringsSection;
 connectionStringSettings = connectionStringsSection.ConnectionStrings[name];
}
else
 connectionStringSettings = ConfigurationManager.ConnectionStrings[name];
 …
}
```

Verification

Running the test case again after applying the preceding code verified that the application block could retrieve configuration information from the dictionary configuration source.

Testing the Exception Handling Application Block

The test case determined whether the Exception Handling Application Block logged the information retrieved by the .NET Framework **Exception.Data** property. This property gets a collection of key/value pairs that provide additional, user-defined information about an exception.

Test Case

The test case created an **Exception** object and added additional information to the **Data** collection. The test case used the **ExceptionPolicy.HandleException** method and checked to see whether the additional information, in addition to the exception, was logged.

Problem

The test case failed because the additional information was not logged. The **LoggingExceptionHandler**.**HandlerException** method passes the exception message and the stack trace but not the additional information that is stored in the **Data** collection. Therefore, this information is not logged. The following is the code that caused the failure.

```
public Exception HandleException(Exception exception, Guid handlingInstanceId)
{
    WriteToLog(CreateMessage(exception, handlingInstanceId));
    return exception;
}
```

Solution

The **WriteToLog** method was modified. A new parameter named **exception.Data** was added to the method signature. This parameter passes the additional information. Also, within the **WriteToLog** method, code was added that enumerated the **Data** collection and added that information to the **ExtendedProperties** collection of the **LogEntry** instance. The following is the modified code.

```
public Exception HandleException(Exception exception, Guid handlingInstanceId)
{
    WriteToLog(CreateMessage(exception, handlingInstanceId), exception.Data);
    return exception;
}

protected virtual void WriteToLog(string logMessage, IDictionary exceptionData)
{
    …
    foreach (DictionaryEntry dataEntry in exceptionData)
```

```
    {
        if (dataEntry.Key is string)
        {
            entry.ExtendedProperties.Add(dataEntry.Key as string, dataEntry.
Value);
        }
    }
    this.logWriter.Write(entry);
}
```

Verification

Running the test case again after applying the preceding code verified that the application block logged the additional information.

Testing the Logging Application Block

The test case checked to see whether multiple traces sources that were configured to use the same trace listener used the same instance of that trace listener to log messages. For example, if two trace sources were configured to use the same trace listener, they should both use the same instance of that trace listener.

Test Case

The test case configured multiple trace sources to log messages to a single instance of the **FlatFileTraceListener**. It then added the relevant categories to a **LogEntry** object. These categories route the messages to the trace listener. It logged the messages with the **Logger.Write** method.

Problem

The test case failed because the application block created one instance of the trace listener for every trace source. For example, if two trace sources were configured to use the same trace listener, two instances of the same trace listener were created and the message was logged twice. The trace listener that acquired the lock wrote to the log. The second trace listener could not write to the log, but the message was logged to a file created by the .NET Framework in response to the problem. Because the file was locked by whichever instance of the trace listener acquired the lock first, the other instances that tried to log the message threw exceptions. The .NET Framework handled the exceptions and created a new file with a new GUID for every trace listener instance that unsuccessfully tried to access the locked file.

Solution

The solution was to create a cache of all the configured trace listeners. The cache should contain only one instance of each trace listener.

Verification

Running the test case after the cache was added verified that the message was logged only once to the file and that the .NET Framework did not create multiple files with different GUIDs.

Testing the Caching Application Block

The test case checked to see that the Caching Application Block did not try to scavenge items in the cache when the number of items to scavenge was set to zero.

Test Case

The test case checked to see that the **ScavengerTask** class looked at the number of times to scavenge before it began scavenging.

Problem

The test case failed because the **DoScavenging** method did not check the number of items to be scavenged before it began to scavenge. As a result, each time an item was added, the method would unnecessarily scavenge the cache. The following is the code that caused the problem.

```
public void DoScavenging()
{
  Hashtable liveCacheRepresentation = cacheOperations.CurrentCacheState;
  int currentNumberItemsInCache = liveCacheRepresentation.Count;
  if (scavengingPolicy.IsScavengingNeeded(currentNumberItemsInCache))
  {
    ResetScavengingFlagInCacheItems(liveCacheRepresentation);
    SortedList scavengableItems = SortItemsForScavenging(liveCacheRepresentation);
    RemoveScavengableItems(scavengableItems);
  }
}
```

Solution

The **ScavengerTask.DoScavenging** method was modified so that it first checked to see if the number of items to be scavenged was set to zero. If it was, it did not scavenge the cache. The following is the modified code.

```
public void DoScavenging()
{
  if (NumberOfItemsToBeScavenged == 0) return;
         Hashtable liveCacheRepresentation = cacheOperations.CurrentCacheState;
  int currentNumberItemsInCache = liveCacheRepresentation.Count;
  if (scavengingPolicy.IsScavengingNeeded(currentNumberItemsInCache))
  {
    ResetScavengingFlagInCacheItems(liveCacheRepresentation);
    SortedList scavengableItems = SortItemsForScavenging(liveCacheRepresentation);
    RemoveScavengableItems(scavengableItems);
  }
}
```

Verification

Running the test case again after applying the preceding code verified that no scavenging took place when the number of items to scavenge was set to zero.

Testing the Security Application Block

The test case checked to see that the Security Application Block's secure cache did not generate valid tokens for an invalid identity, principal, or profile. Instead, the cache should throw an exception.

Test Case

The test case used the **SecurityCacheFactory** to create an in-memory cache store. It passed null values to the **SaveIdentity**, **SavePrincipal**, and **SaveProfile** methods. The following code shows the test case for the **SaveProfile** method.

```
 [TestMethod]
 [ExpectedException(typeof(ArgumentNullException))]
public void InMemorySaveWithNullProfileTestFixture()
{
   ISecurityCacheProvider securityCache = SecurityCacheFactory.GetSecurityCachePro
vider("CacheProvider");
    IToken token = securityCache.SaveProfile(null);
     if (token != null)
     {
                Assert.Fail();
      }
}
```

Problem

The test case failed because the cache issued a valid token for an invalid value. The problem was that the **CachingStoreProvider** class did not validate the input values. The following is the code that caused the problem.

```
public override IToken SaveProfile(object profile)
{
    GuidToken guidToken = new GuidToken();
    SaveProfile(profile, guidToken);
    return guidToken;
}
```

Solution

The **SaveProfile**, **SaveIdentity**, and **SavePrincipal** methods on the **SecurityCacheProvider** class were modified to check the input values and to throw exceptions if they were null. The following is the modified **SaveProfile** method.

```
public override IToken SaveProfile(object profile)
{
    if (profile == null)
    {
        throw new ArgumentNullException("profile");
    }
…
}
```

Verification

Running the test case again after applying the preceding code verified that the cache threw the **ArgumentNullException** exception when the input value was null.

Index

E

F

G

H

I

R

S

T

U

V

W

X