Microsoft

# Testing .NET Application Blocks

**Version 1.0**

patterns & practices

# Contents

Chapter 9

# 1

# Introduction

## Overview

*Testing .NET Application Blocks* provides a process for testing .NET application blocks. The guide is designed to be used as a reference or to be read from beginning to end. It consists of three parts:

- **Part I,** "**Overview,**" gives an overview of the testing process and describes a process for test driven development, which is a core practice of Extreme Programming.
- **Part II,** "**Testing a .NET Application Block,**" presents the process for testing an application block and has a separate process for each step.
- **Part III,** "**Integration Testing for .NET Application Blocks,**" presents the process for testing customizations of the application block and for integration testing an application with an application block.

## Why We Wrote This Guide

The *patterns & practices* group at Microsoft® offers a number of application blocks that are available for free download from MSDN®. These application blocks are reusable, configurable, and customizable blocks of code. They are rigorously tested to ensure that they meet all the stated requirements and the best practices for design and implementation. You can easily integrate these application blocks with other applications, either using the blocks without modification or customizing them according to your requirements.

---

**Note:** The application blocks are not stand-alone executable files.

---

For a list of applications blocks that includes a link to each block, see the Appendix later in this chapter.

This guide shows developers and testers how to:

- Test the application blocks during their development/customization life cycle.
- Test the integration of an application block with an application.

.NET application blocks can be tested during the following stages:

- **Development life cycle of the application block.** You must test the application block to ensure that the block meets the requirements and objectives for which it is being developed. You must also test the application block to ensure that the design and the implementation follow best practices for performance, security, maintainability, and extensibility.
- **Customization of the application block.** You must test the changes made to the application block to ensure that the changes meet the objectives for which the block has been customized. You must also unit test the original functionality to ensure that customization has not broken the existing features.
- **Integration of the application block with an application.** You must test the integration of the application with the application block to ensure that the application block can process the application input and return the output accurately and consistently. You may also need to verify that the integration has not introduced bottlenecks.

## Scope of This Guide

This guide presents recommendations on how to test .NET application blocks during their development life cycle, during customization, and during their integration with an application. The recommended process can be easily used no matter what software development life cycle you use to develop, customize, and integrate these blocks.

Because this guide focuses on testing NET application blocks, its scope does not include user interface (UI) testing or database testing. Although the guidance is developed within the context of Microsoft *patterns & practices* application blocks, it applies to testing .NET code in general.

## Audience

This guide is intended for the following audience:

- Testers and test leads who are involved in the development and test life cycle of an application block.

- Testers and test leads who are involved in testing an application that requires integrating an application block in "off-the-shelf" manner or after customizing it.
- Developers who are involved in customizing an application block or integrating a block with other applications.

# How to Use This Guide

You can read this guide from beginning to end, or you can read just the parts or chapters that apply to your role.

## Applying the Guidance to Your Role

The guidance breaks down the testing processes into discrete steps such that each tester can follow the process for his or her area of specialization. The following parts apply to the tester's role:

- **Part II.** Part II presents an overall process for testing the application blocks. Each step in this testing process is further broken down into processes. Testers can selectively choose among these processes to perform tests depending on their area of specialization.
- **Part III.** Part III shows testers how to unit test customizations of the application block and how to test the integration of an application with the application block.

Test leads can use this guidance to enforce a standardized process within their organization. Specifically, they can:

- Efficiently track the progress of testing, because the process has been broken down into discrete steps.
- Conduct a risk analysis and distribute the effort among various steps. In this way the testing effort can be focused on high risk areas.
- Ensure that, by following all the steps in the process, the application block is tested rigorously.

The following parts apply to the test lead role:

- **Part I.** Part I helps the test lead to implement test-driven development, which is a core programming practice for Extreme Programming.
- **Part II.** Chapter 3, "Testing Process for Application Blocks," presents an overall process for testing the application blocks.
- **Part III.** Part III helps the test lead to enforce a standard process for testing any customizations of the application block and describes how to integration test the block with an application.

Developers can use this guidance to conduct design and code reviews of their own code.

The following part applies to the developer role:

- **Part II.** Chapter 4, "Design Review for Application Blocks" and Chapter 5, "Code Review for Application Blocks" provide the processes for design and code review. The developer can use these processes to identify various issues early in the development life cycle.

## Applying the Guidance to Your Application's Life Cycle

Figure 1.1 shows how the guidance for testing application blocks applies to the broad categories associated with an application's life cycle, regardless of the development methodology used. The recommended process for testing .NET application blocks runs as a separate but parallel set of activities throughout the development life cycle.

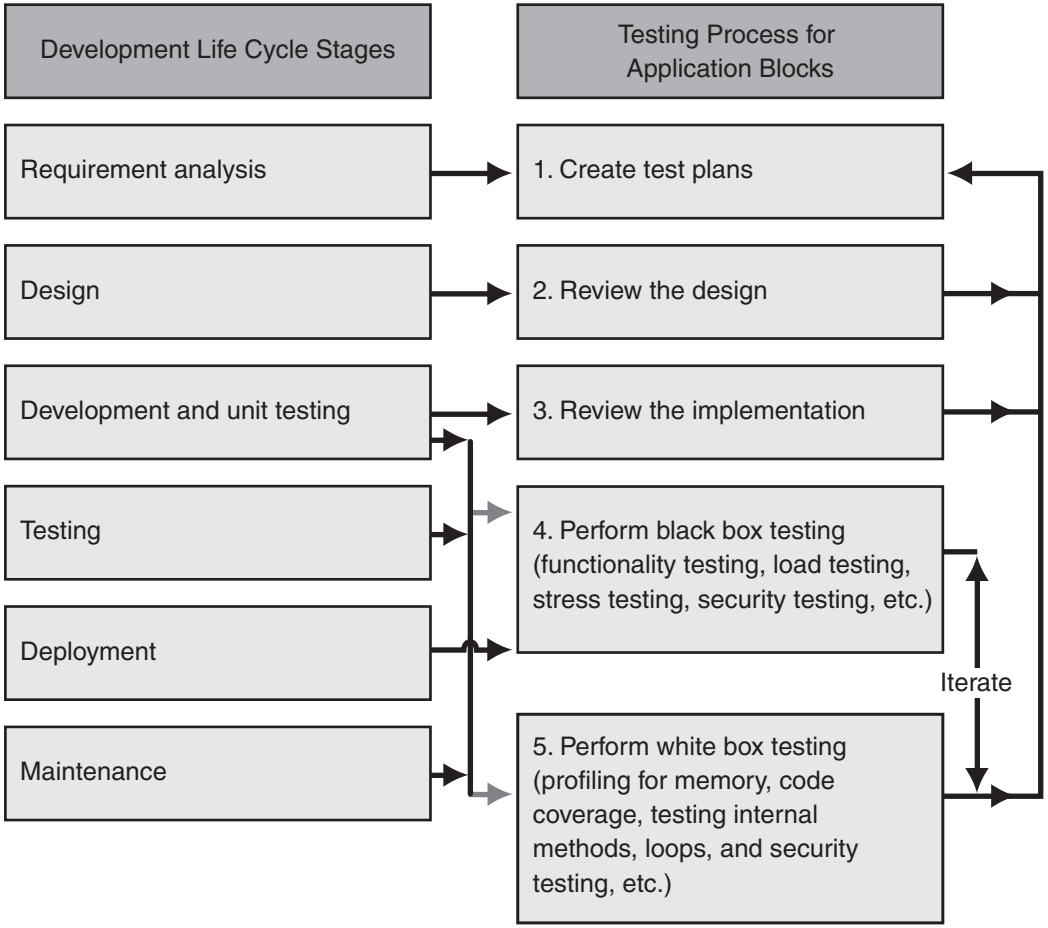


**Figure 1.1**
*Life cycle mapping*

## The Team Who Brought You This Guide

The following people created this guide:

- **Mohammad Al-Sabt,** Microsoft Corporation
- **Ken Perilman,** Microsoft Corporation
- **Ashish Babbar**, Infosys Technologies Ltd.
- **Sameer Tarey**, Infosys Technologies Ltd.
- **Ramprasad Ramamurthy**, Infosys Technologies Ltd.
- **Terrence J. Cyril**, Infosys Technologies Ltd.

## Contributors and Reviewers

Many thanks to the following contributors and reviewers:

- **Special thanks to key contributors:** Prashant Bansode, Nitai Utkarsh, Sankaranarayanan Milerangam Gopalan, Venkata Narayana Varun, and Hariharasudan T, Infosys Technologies Ltd.
- **Special thanks to key reviewers:** Guru Sundaram, Mrinal Bhao, and Ravi Prasad VV, Infosys Technologies Ltd.
- **Thanks to our *patterns & practices* team members for technical feedback and input:** Edward Lafferty, Carlos Farre, Chris Sfanos, J. D. Meier, Jim Newkirk, and Edward Jezierski (Microsoft Corporation)
- **Thanks to external reviewers:** Amit Bahree, Technologist at Capgemini; Matt Hessinger (Hessinger Consulting, Inc); Software Engineering and Technology Labs (SETLabs) and Independent Validation Services (IVS), Infosys Technologies Ltd.
- **Thanks to our editors:** Sharon Smith (Microsoft Corporation), Susan Filkins (Entirenet), and Tina Burden McGrayne (Entirenet)
- **Thanks to our graphic designer:** Claudette Iebbiano (Wadeware LLC)
- **Thanks to our release manager:** Sanjeev Garg (Satyam Computer Services)

## Summary

In this introduction, you were shown the structure of the guide and the basic approach used by it to test the .NET application blocks. You were also shown how to apply the guidance to your role or to specific phases of your product development life cycle.

# Appendix

The *patterns & practices* group at Microsoft offers the following application blocks. For continuous updates to this list, see *http://www.microsoft.com/resources/practices/code.mspx*.

- Aggregation Application Block for .NET: (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/ServiceAgg.asp*)
- Asynchronous Invocation Application Block for .NET: (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/PAIBlock.asp*)
- Authorization and Profile Application Bloc: (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/authpro.asp*)
- Caching Application Block for .NET: (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/CachingBlock.asp*)
- Configuration Management Application Block for .NET: (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/cmab.asp*)
- Data Access Application Block for .NET: (*http://msdn.microsoft.com/library/en-us/dnbda/html/daab-rm.asp*)
- Exception Management Application Block for .NET: (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/emab-rm.asp*)
- Logging Application Block: (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/Logging.asp*)
- Smart Client Offline Application Block: (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/offline.asp*)
- Updater Application Block for .NET: (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/updater.asp*)
- User Interface Process Application Block - Version 2.0: (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/uipab.asp*)
- User Interface Process Application Block for .NET: (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/uip.asp*)

# 2

# Testing Methodologies

## Objectives

- Look at the various methodologies for testing software.
- Learn about test-driven development.

## Overview

There are numerous methodologies available for developing and testing software. The methodology you choose depends on factors such as the nature of project, the project schedule, and resource availability. Although most software development projects involve periodic testing, some methodologies focus on getting the input from testing early in the cycle rather than waiting for input when a working model of the system is ready. Those methodologies that require early test involvement have several advantages, but also involve tradeoffs in terms of project management, schedule, customer interaction, budget, and communication among team members. This chapter discusses how testing fits into the various software development life cycles and then discusses the test-driven development practice in detail.

The Configuration Management Application Block (CMAB) is used to illustrate concepts in this chapter. The requirements for the CMAB are as follows:

- It provides the functionality to read and store configuration information transparently in persistent storage media. The storage media are the Microsoft® SQL Server™ database system, the registry, and XML files.
- It provides a configurable option to store the information in encrypted form and in plain text using XML notation.
- It can be used with desktop applications and with Web applications that are deployed in a Web farm.

- It caches configuration information in memory to reduce cross-process communication such as reading from any persistent medium. This caching reduces the response time of the request for configuration information. The expiration and scavenging mechanism for the data cached in memory is similar to the cron algorithm in UNIX.

# Software Development Methodologies and Testing

Various generic software development life cycle methodologies are available for executing software development projects. Although each methodology is designed for a specific purpose and has its own advantages and disadvantages, most methodologies divide the life cycle into phases and share tasks across these phases. This section briefly summarizes common methodologies used for software development and describes their relationship to testing.

## Waterfall Model

The waterfall model is one of the earliest structured models for software development. It consists of the following sequential phases through which the development life cycle progresses:

- **System feasibility.** In this phase, you consider the various aspects of the targeted business process, find out which aspects are worth incorporating into a system, and evaluate various approaches to building the required software.
- **Requirement analysis.** In this phase, you capture software requirements in such a way that they can be translated into actual use cases for the system. The requirements can derive from use cases, performance goals, target deployment, and so on.
- **System design.** In this phase, you identify the interacting components that make up the system. You define the exposed interfaces, the communication between the interfaces, key algorithms used, and the sequence of interaction. An architecture and design review is conducted at the end of this phase to ensure that the design conforms to the previously defined requirements.
- **Coding and unit testing.** In this phase, you write code for the modules that make up the system. You also review the code and individually test the functionality of each module.
- **Integration and system testing.** In this phase, you integrate all of the modules in the system and test them as a single system for all of the use cases, making sure that the modules meet the requirements.
- **Deployment and maintenance.** In this phase, you deploy the software system in the production environment. You then correct any errors that are identified in this phase, and add or modify functionality based on the updated requirements.

The waterfall model has the following advantages:

- It allows you to compartmentalize the life cycle into various phases, which allows you to plan the resources and effort required through the development process.
- It enforces testing in every stage in the form of reviews and unit testing. You conduct design reviews, code reviews, unit testing, and integration testing during the stages of the life cycle.
- It allows you to set expectations for deliverables after each phase.

The waterfall model has the following disadvantages:

- You do not see a working version of the software until late in the life cycle. For this reason, you can fail to detect problems until the system testing phase. Problems may be more costly to fix in this phase than they would have been earlier in the life cycle.
- When an application is in the system testing phase, it is difficult to change something that was not carefully considered in the system design phase. The emphasis on early planning tends to delay or restrict the amount of change that the testing effort can instigate, which is not the case when a working model is tested for immediate feedback.
- For a phase to begin, the preceding phase must be complete; for example, the system design phase cannot begin until the requirement analysis phase is complete and the requirements are frozen. As a result, the waterfall model is not able to accommodate uncertainties that may persist after a phase is completed. These uncertainties may lead to delays and extended project schedules.

## Incremental or Iterative Development

The incremental, or *iterative*, development model breaks the project into small parts. Each part is subjected to multiple iterations of the waterfall model. At the end of each iteration, a new module is completed or an existing one is improved on, the module is integrated into the structure, and the structure is then tested as a whole.

For example, using the iterative development model, a project can be divided into 12 one- to four-week iterations. The system is tested at the end of each iteration, and the test feedback is immediately incorporated at the end of each test cycle. The time required for successive iterations can be reduced based on the experience gained from past iterations. The system grows by adding new functions during the development portion of each iteration. Each cycle tackles a relatively small set of requirements; therefore, testing evolves as the system evolves. In contrast, in a classic waterfall life cycle, each phase (requirement analysis, system design, and so on) occurs once in the development cycle for the entire set of system requirements.

The main advantage of the iterative development model is that corrective actions can be taken at the end of each iteration. The corrective actions can be changes to the specification because of incorrect interpretation of the requirements, changes to the

requirements themselves, and other design or code-related changes based on the system testing conducted at the end of each cycle.

The main disadvantages of the iterative development model are as follows:

- The communication overhead for the project team is significant, because each iteration involves giving feedback about deliverables, effort, timelines, and so on.
- It is difficult to freeze requirements, and they may continue to change in later iterations because of increasing customer demands. As a result, more iterations may be added to the project, leading to project delays and cost overruns.
- The project requires a very efficient change control mechanism to manage changes made to the system during each iteration.

## Prototyping Model

The prototyping model assumes that you do not have clear requirements at the beginning of the project. Often, customers have a vague idea of the requirements in the form of objectives that they want the system to address. With the prototyping model, you build a simplified version of the system and seek feedback from the parties who have a stake in the project. The next iteration incorporates the feedback and improves on the requirements specification. The prototypes that are built during the iterations can be any of the following:

- A simple user interface without any actual data processing logic
- A few subsystems with functionality that is partially or completely implemented
- Existing components that demonstrate the functionality that will be incorporated into the system

The prototyping model consists of the following steps.

1. **Capture requirements.** This step involves collecting the requirements over a period of time as they become available.
2. **Design the system.** After capturing the requirements, a new design is made or an existing one is modified to address the new requirements.
3. **Create or modify the prototype.** A prototype is created or an existing prototype is modified based on the design from the previous step.
4. **Assess based on feedback.** The prototype is sent to the stakeholders for review. Based on their feedback, an impact analysis is conducted for the requirements, the design, and the prototype. The role of testing at this step is to ensure that customer feedback is incorporated in the next version of the prototype.
5. **Refine the prototype.** The prototype is refined based on the impact analysis conducted in the previous step.
6. **Implement the system.** After the requirements are understood, the system is rewritten either from scratch or by reusing the prototypes. The testing effort consists of the following:

- Ensuring that the system meets the refined requirements
- Code review
- Unit testing
- System testing

The main advantage of the prototyping model is that it allows you to start with requirements that are not clearly defined.

The main disadvantage of the prototyping model is that it can lead to poorly designed systems. The prototypes are usually built without regard to how they might be used later, so attempts to reuse them may result in inefficient systems. This model emphasizes refining the requirements based on customer feedback, rather than ensuring a better product through quick change based on test feedback.

## Agile Methodology

Most software development life cycle methodologies are either iterative or follow a sequential model (as the waterfall model does). As software development becomes more complex, these models cannot efficiently adapt to the continuous and numerous changes that occur. Agile methodology was developed to respond to changes quickly and smoothly. Although the iterative methodologies tend to remove the disadvantage of sequential models, they still are based on the traditional waterfall approach. Agile methodology is a collection of values, principles, and practices that incorporates iterative development, test, and feedback into a new style of development. For an overview of agile methodology, see the Agile Modeling site at: *http://www.agilemodeling.com/*.

The key differences between agile and traditional methodologies are as follows:

- **Development is incremental rather than sequential.** Software is developed in incremental, rapid cycles. This results in small, incremental releases, with each release building on previous functionality. Each release is thoroughly tested, which ensures that all issues are addressed in the next iteration.
- **People and interactions are emphasized, rather than processes and tools.** Customers, developers, and testers constantly interact with each other. This interaction ensures that the tester is aware of the requirements for the features being developed during a particular iteration and can easily identify any discrepancy between the system and the requirements.
- **Working software is the priority rather than detailed documentation.** Agile methodologies rely on face-to-face communication and collaboration, with people working in pairs. Because of the extensive communication with customers and among team members, the project does not need a comprehensive requirements document.

- **Customer collaboration is used, rather than contract negotiation.** All agile projects include customers as a part of the team. When developers have questions about a requirement, they immediately get clarification from customers.
- **Responding to change is emphasized, rather than extensive planning.** Extreme Programming does not preclude planning your project. However, it suggests changing the plan to accommodate any changes in assumptions for the plan, rather than stubbornly trying to follow the original plan.

Agile methodology has various derivate approaches, such as Extreme Programming, Dynamic Systems Development Method (DSDM), and SCRUM. Extreme Programming is one of the most widely used approaches.

## Extreme Programming

In Extreme Programming, rather than designing whole of the system at the start of the project, the preliminary design work is reduced to solving the simple tasks that have already been identified.

The developers communicate directly with customers and other developers to understand the initial requirements. They start with a very simple task and then get feedback by testing their software as soon as it is developed. The system is delivered to the customers as soon as possible, and the requirements are refined or added based on customer feedback. In this way, requirements evolve over a period of time, and developers are able to respond quickly to changes.

The real design effort occurs when the developers write the code to fulfill the specific engineering task. The engineering task is a part of a greater *user story* (which is similar to a use case). The user story concerns itself with how the overall system solves a particular problem. It represents a part of the functionality of the overall system. A group of user stories is capable of describing the system as a whole. The developers refactor the previous code iteration to establish the design needed to implement the functionality.

During the Extreme Programming development life cycle, developers usually work in pairs. One developer writes the code for a particular feature, and the second developer reviews the code to ensure that it uses simple solutions and adheres to best design principles and coding practices.

Discussion of the core practices of Extreme Programming is beyond the scope of this chapter. For more information, see the links referred to in "More Information" later in this section.

Test-driven development, which is one of the core practices in Extreme Programming, is discussed in greater detail later in this chapter.

### When to Use Extreme Programming

Extreme Programming is useful in the following situations:

- When the customer does not have a clear understanding of the details of the new system. The developers interact continuously with the customer, delivering small pieces of the application to the customer for feedback, and taking corrective action as necessary.
- When the technology used to develop the system is new compared to other technologies. Frequent test cycles in Extreme Programming mitigate the risk of incompatibility with other existing systems.
- When you can afford to create automated unit and functional tests. In some situations, you may need to change the system design so that each module can be tested in isolation using automated unit tests.
- When the team size is not very large (usually 2 to 12 people). Extreme Programming is successful in part because it requires close team interaction and working in pairs. A large team would have difficulty in communicating efficiently at a fast pace. However, large teams have used Extreme Programming successfully.

### More Information

For more information about the core practices in Extreme Programming, see the following resources:

- "What Is Extreme Programming?" at:
  *http://www.xprogramming.com/xpmag/whatisxp.htm*
- "Extreme Programming: A Gentle Introduction" at:
  *http://www.extremeprogramming.org/*

## Test-Driven Development

Test-driven development is one of the core practices of Extreme Programming. The practice extends the feedback approach, and requires that you develop test cases before you develop code. Developers develop functionality to pass the existing test cases. The test team then adds new test cases to test the existing functionality, and runs the entire test suite to ensure that the code fails (either because the existing functionality needs to be modified or because required functionality is not yet included). The developers then modify the functionality or create new functionality so that the code can withstand the failed test cases. This cycle continues until the test code passes all of the test cases that the team can create. The developers then refactor the functional code to remove any duplicate or dead code and make it more maintainable and extensible.

Test-driven development reverses the traditional development process. Instead of writing functional code first and then testing it, the team writes the test code before

the functional code. The team does this in very small steps — one test and a small amount of corresponding functional code at a time. The developers do not write code for new functionality until a test fails because some functionality is not present. Only when a test is in place do developers do the work required to ensure that the test cases in the test suite pass. In subsequent iterations, when the team has the updated code and another set of test cases, the code may break several existing tests as well as the new tests. The developers continue to develop or modify the functionality to pass all of the test cases.

Test-driven development allows you to start with an unclear set of requirements and relies on the feedback loop between the developers and the customers for input on the requirements. The customer or a customer representative is the part of the core team and immediately provides feedback about the functionality. This practice ensures that the requirements evolve over the course of the project cycle. Testing before writing functional code ensures that the functional code addresses all of the requirements, without including unnecessary functionality.

With test-driven development, you do not need to have a well-defined architectural design before beginning the development phase, as you do with traditional development life cycle methodologies. Test-driven development allows you to tackle smaller problems first and then evolve the system as the requirements become more clear later in the project cycle.

Other advantages of test-driven development are as follows:

- Test-driven development promotes loosely coupled and highly cohesive code, because the functionality is evolved in small steps. Each piece of the functionality needs to be self-sufficient in terms of the helper classes and the data that it acts on so that it can be successfully tested in isolation.
- The test suite acts as documentation for the functional specification of the final system.
- The system uses automated tests, which significantly reduce the time taken to retest the existing functionality for each new build of the system.
- When a test fails, you have a clear idea of the tasks that you must perform to resolve the problem. You also have a clear measure of success when the test no longer fails. This increases your confidence that the system actually meets the customer requirements.

Test-driven development helps ensure that your source code is thoroughly unit tested. However, you still need to consider traditional testing techniques, such as functional testing, user acceptance testing, and system integration testing. Much of this testing can also be done early in your project. In fact, in Extreme Programming, the acceptance tests for a user story are specified by the project stakeholder(s) either before or in parallel to the code being written, giving stakeholders the confidence that the system meets their requirements.

## Steps in Test-Driven Development

The test-driven development process consists of the steps shown in Figure 2.1.

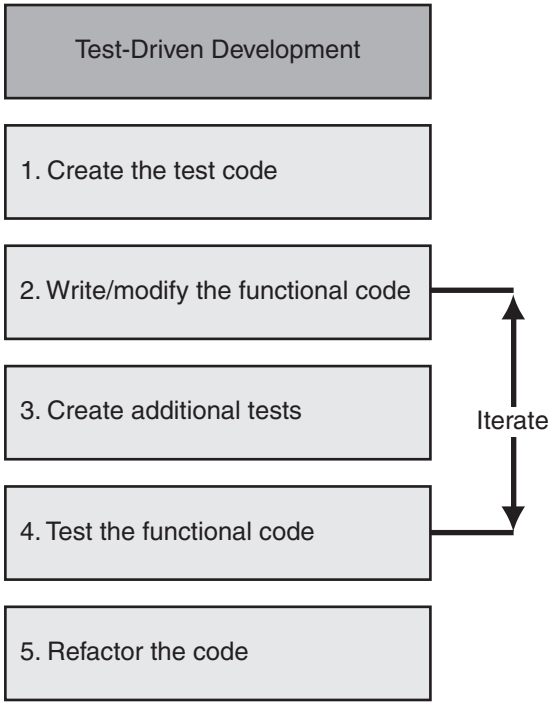


**Figure 2.1**
*Test-driven development process*

The steps can be summarized as follows:

1. **Create the test code.** Use an automated test framework to create the test code. The test code drives the development of functionality.
2. **Write/Modify the functional code.** Write the functional code for the application block so that it can pass all test cases from the test code. The first iteration involves developing new functionality, and subsequent iterations involve modifying the functionality based on the failed test cases.
3. **Create additional tests.** Develop additional tests for testing of the functional code.
4. **Test the functional code.** Test the functional code based on the test cases developed in Step 3 and Step 1. Repeat steps 2 through 4 until the code is able to pass all of the test cases.

5. **Refactor the code.** Modify the code so that there is no dead code or duplication. The code should adhere to best practices for maintainability, performance, and security.

The following subsections describe each step in detail.

## Step 1: Create the Test Code

You create the test code before any code is written for the functionality of the application block. You can either write a custom test suite or use a testing framework (such as NUnit) to automate testing of the API. The general guidelines for writing the test code are as follows:

- In Step 1, your goal is to write test code that tests basic functionality.
- Write a test case for each code requirement. The developers will write the functional code based on the tests so that the test cases have an execution result of "pass."
- Avoid writing code for multiple test cases at any one time. Write code for a single test case, and proceed through the remaining cycle of coding and refactoring.
- Write code that tests only small pieces of functionality. If the test code is complex, divide it into smaller tests so that each one tests only a small piece of functionality.

After you complete Step 1, you should have various test suites written to test the functionality. For example, if you need to test the functionality of the CMAB using NUnit, create test code first, as follows:

```
[Test]
public void GetXmlFileSection() {
string sourceSection = "XmlFile";
CustomConfigurationData custom = ConfigurationManager.Read( sourceSection ) as
CustomConfigurationData;

Assert.Equals("Red", custom.Color, "Red color was expected" );
    Assert.Equals(45, custom.Size, "45 size was expected" );
Assert.Equals("Some text", custom.SomeText, "Some text was expected" );
}
public class CustomConfigurationData{
public CustomConfigurationData() {}
    public string Color{
        get{ return _color; }
        set{ _color = value; }
    } string _color;

    public string SomeText    {
        get{ return _some_text; }
        set{ _some_text = value; }
    } string _some_text;
    public int Size{
        get{ return _size; }
        set{ _size = value; }
```

```
    } int _size;
    public override string ToString(){
return "Color = " + _color + "; FontSize = " + _size.ToString(
System.Globalization.CultureInfo.CurrentUICulture );
    }
}
```

You will not be able to compile the test code until the developer writes the functional code. The developer then writes the code so that the test compiles and has an execution result of "pass."

### More Information

For more information about NUnit, see *http://www.nunit.org*.

## Step 2: Write or Modify the Functional Code

In Step 2, your goal is to develop functionality that passes the test cases that were created in Step 1. If you are in the first iteration, in all probability the test code will not compile and you can assume that it failed. You must modify the functionality being tested so that the test code compiles and can be executed.

After the functionality passes all of the test cases created in Step 1, the developer stops developing the module.

The functionality in the following example passes the test case that was created in Step 1.

```
class ConfigurationManager {
public static object Read(string sourceSection){
CustomConfigurationData customConfiguration = new CustomConfigurationData();
customConfiguration.Color = "Red";
customConfiguration.SomeText = "Some text";
customConfiguration.Size = "45";
return obj;
}
```

The outcome of Step 2 is code that implements basic functionality. However, in the later iterations, when additional test cases are created to test the functionality, the code in the previous example will fail. The developers must make further changes in subsequent iterations so that all of the functionality can pass the additional test cases. The developers continue to improve the functionality until all of the existing test cases have an execution result of "pass."

## Step 3: Create Additional Tests

After Step 2, the code has the basic functionality to pass all of the test cases that were created in Step 1. The tester must now test this functionality for various types of input.

You should ensure that the amount of time spent developing additional test cases is proportionate to the criticality of the feature. For example, if various code paths in an API need to be tested by providing different input, you can write multiple test stubs, with each stub catering to one possible code path.

The general guidelines for creating these additional tests are as follows:

- Create additional tests that could possibly break the code. Continue to create these additional tests until you have no more test cases.
- Write test cases that focus on the goal of the code (what the user needs the code to do), rather than the implementation. For example, if the goal of a function is to return values from a database based on some identifier, test the function by passing both valid and invalid identifier values, rather than testing only those values that the current implementation supports (which can be only a valid set of values).
- If all of the test cases that were created in Step 1 have passed but still the functionality does not work as intended, you have probably missed an important test scenario and must develop additional test cases that reproduce the failure of the functionality. These test cases are in addition to the existing test cases that pass invalid input to the API to force an exception.
- Avoid writing separate test cases for each unique combination of input. The number of test cases you write should be based on the risk assessment of the feature and on ensuring that all critical scenarios and the majority of test input have been accounted for.

The following additional test case will fail the functionality that was written for the example in Step 2.

```
[Test]
public void GetSqlSection() {
string sourceSection = "SQLServer";
CustomConfigurationData custom = ConfigurationManager.Read( sourceSection ) as
SqlData;

Assert.Equals("John", custom.FirstName, "John was expected");
Assert.Equals("Trovalds", custom.LastName, "Trovalds was expected");
}
public class SqlData{
public SqlData () {}
    public string FirstName{
        get{ return _firstname; }
        set{ _firstname = value; }
    } string _firstname;
    public string LastName    {
        get{ return _lastName; }
        set{ _lastName = value; }
    } string _lastName;
}
```

## Step 4: Test the Functional Code

The next step is to test the functionality by using the additional test cases. To do so, execute all of the test cases from Step 1and Step 3, and note any test cases that fail the test. The developers must then make changes to the functional code so that it passes all of the existing test cases.

Repeat steps 2, 3, and 4 until the testers can no longer create additional test cases that break the functionality of the code. Each time a test fails, the developers fix the functionality based on the test cases and submit the improved code to the testers for a new round of testing.

## Step 5: Refactor the Code

After the completion of the previous steps, the code can be run and is capable of handling all of the test cases. In Step 5, the developers refactor the code to make it maintainable and extensible so that changes in one part of the module can be worked on in isolation without changing the actual interfaces exposed to the end user.

In addition to making code easier to maintain, refactoring removes duplicate code and can reduce code complexity. This helps to ensure that the basic principles of loosely coupled and highly cohesive design are followed without breaking the functionality in any way. For example, if you have the functionality shown in the first example, you can refactor it as shown in the second example. (In these examples, the helper class does not own the data it works on. Instead, the caller passes the data to the helper class. This is typical of the BOB class anti-pattern.)

```
[Original Code]
Class BOBClass
{
    void DoWork()
    {
        Helper hlpObject = new HelperClass();
        If (hlpObject.CanDoWork)
        {
            Double x = hlpObject.SomeValue + hlpObject.SomeOtherValue;
            hlpObject.NewValue = x;
        }
    }
}
Class Helper
{
    bool CanDoWork = true;
    double SomeValue;
    double SomeOtherValue;
    double NewValue;
}
```

```
[Refactored Code]


Class BOBClass{
    void DoWork(){
        Helper hlpObject = new HelperClass();
        hlpObject.DoWork();
    }
}
Class Helper{
    bool CanDoWork = true;
    double SomeValue;
    double SomeOtherValue;
    double NewValue;
    void DoWork(){
        If (CanDoWork)
            NewValue = SomeValue + SomeOtherValue;
}
}
```

Depending on the size of the application you are developing, you should consider refactoring the code at regular intervals during the development cycle. Doing so helps to reduce code redundancy and to optimize the design, and it prevents the refactoring exercise from being a monumental task at the end of the project.

After completing Step 5, you repeat the test-driven development process for each subsequent feature requirement, until the system being developed is complete.

Although the test-driven model requires you to test each line of code as it is developed, you still need to perform other types of testing, such as performance testing, security testing, and globalization testing.

## Agile Testing: Example

Agile methodology with Extreme Programming and test-driven development was used to develop the Smart Client Offline Application Block. The following are highlights of the approach taken on the project:

- The test team and the development team were not formally separated. The developers worked in pairs, with one person developing the test cases and the other writing the functionality for the module.
- There was much more interaction among team members than there is when following a traditional development model. In addition to using the informal chat-and-develop mode, the team held a 30 minute daily standup meeting, which gave team members a forum for asking questions and resolving problems, and weekly iterative review meetings to track the progress for each iterative cycle.

- Project development began without any formal design document. The specifications were in the form of user stories that were agreed upon by the team members. In the weekly iterative review meetings, team members planned how to complete these stories and how many iterations to assign for each story.
- Each story was broken down into several tasks. All of the stories and corresponding tasks were written down on small cards that served as the only source of design documentation for the application block.
- While developing each task or story, NUnit test suites were written to drive the development of features.
- No formal test plans were developed. The testing was primarily based on the tasks or stories for feature development. The development team got immediate feedback from the test team. Having the test team create the quick start samples gave the development team a perspective on the real-life usage of the application block.
- After the task or story passed all of the NUnit test cases and was complete, quick start samples were developed to showcase the functionality. The quick start samples demonstrated the usage of the application block and were useful for further testing the code in the traditional way (functional and integration tests). Any discrepancies found in this stage were reported immediately and were fixed on a case-by-case basis. The modified code was tested again with the automated test suites and then was handed over to be tested again with the quick start samples.

## Summary

This chapter compared various software development life cycle approaches and described how testing fits into each approach. The chapter also presented a process for test-driven development, which is one of the core practices of Extreme Programming, an agile development methodology.

By using test-driven development, you can ensure that each feature of an application block is rigorously tested early in the project life cycle. Early testing significantly reduces the risk of costly fixes and delays later in the life cycle and ensures a robust application block that meets all of the feature requirements.

# 3

# Testing Process for Application Blocks

## Objective

- Learn the process for testing application blocks.

## Overview

When testing application blocks, you follow the same basic steps regardless of the methodologies you use to develop and test the block. The purpose of these steps is to test the application block from different perspectives and to ensure that the application block conforms to all of the requirements and functional specifications.

This chapter describes the process you follow to test application blocks. If required, you can customize the process to suit your development strategy. Before testing the application blocks, you should conduct a risk analysis of the features and the importance of the functionality that the application blocks provide. The risk analysis determines the amount of testing effort you expend for each step.

## Testing Process

The process for testing application blocks is outlined below. The steps listed should be performed regardless of the testing methodology that is used for the blocks.

## Input

The following input is required to test an application block:

- Functional specifications
- Requirements
- Performance objectives
- Deployment scenarios

## Steps

Figure 3.1 shows the steps in the testing process for application blocks.

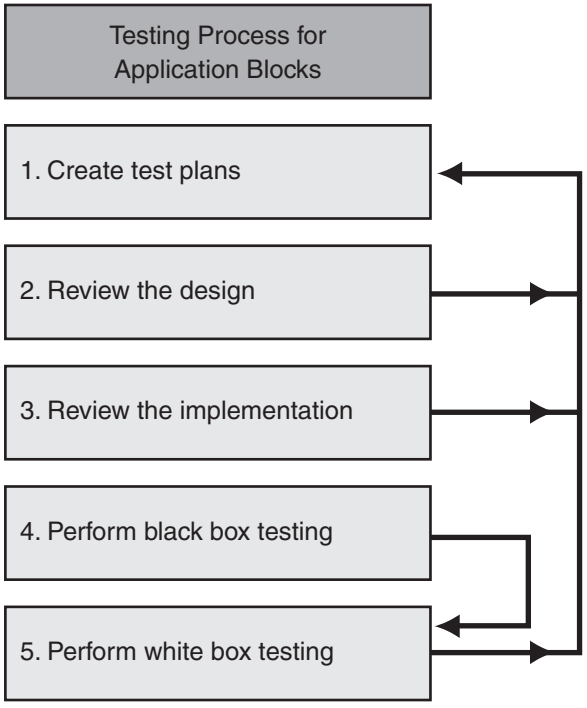


**Figure 3.1**
*The testing process for application blocks*

1. **Create test plans.** Create test plan documentation with a prioritized list of test cases and execution details.
2. **Review the design.** Review the design to ensure that it addresses all the deployment scenarios and the requirements. Also ensure adherence to guidelines for performance, security, globalization, maintainability, and so on.

3. **Review the implementation.** Review the code to ensure that various best practices are followed.
4. **Perform black box testing.** Test the code from an end-user perspective by executing all the end-user scenarios.
5. **Perform white box testing.** Analyze the code for failure scenarios and simulate them by passing invalid data using custom test harnesses.

## Step 1: Create Test Plans

The test plans document the test cases you will use to test the application block. The test cases cover all aspects of testing, including design review, code review, profiling, deployment testing, and load testing. The test plans help to ensure that you test all of the features and usage scenarios of the application block.

The test plan documentation consists of two documents:

- **Detailed test plan (DTP) document.** The detailed test plan document lists test cases in a prioritized order (High, Medium, and Low). The test case descriptions in this document briefly summarize the usage scenarios and features to be tested. For each test case, you assign a priority level based on the case's importance and its overall impact on the goal of meeting the desired objectives and requirements.
- **Detailed test case (DTC) document.** The detailed test case document maps to the detailed test plan document. This document describes the steps that the user must perform to execute each test case that is listed in the DTP document. It also lists the data that is required for the test and describes the expected results of the test.

For examples of DTP and DTC documents, see the "Appendix" section later in this chapter. You can use these examples as templates when you create the test plans for your application block.

You must update the DTP and DTC documents throughout the development life cycle. For example, you must update these documents if functional specifications or requirements change, or if you have additional input. You must also update them if you add test cases or if you modify the priority of the existing test cases later in the development cycle to account for additional usage scenarios or functional testing.

## Step 2: Review the Design

In this step, you execute the test cases from the DTP that are related to reviewing the design of the application block. The design review is very important from the testing point of view, because you can avoid most of the potential pitfalls (for example, those related to performance or security) in the early stages of a project by thoroughly reviewing the design of the application block. In the later stages, considerable rework may be required in the code to make any design changes.

The design review helps to ensure the following:

- The design meets all of the functional specifications and requirements for the application block.
- The design makes appropriate tradeoffs given the deployment scenarios for the application block. Examples of deployment targets that may have special requirements are mobile devices, the DMZ (also known as demilitarized zone or screened subnet) of a Web farm, and tablet PCs.
- The design ensures that the application block meets all performance objectives.
- The design addresses all the security threats possible in various deployment scenarios for the application block.
- The design adheres to best practices and principles related to coupling and cohesion, concurrency, communication, class design, exception management, resource management, caching, and so on, so that developers can easily extend and customize the application block.
- The design adheres to best practices for globalization and localization.

The design review also helps in identifying the scenarios that must be tested for one or more of the following:

- Possible security attacks
- Performance optimizations
- Profiling to ensure that there are no memory leaks

**Note:** You must update the DTP and DTC to document these possible scenarios.

### More Information

For more information about how to conduct a design review for the application block, see Chapter 4, "Design Review for Application Blocks."

## Step 3: Review the Implementation

In this step, you execute the test cases from the DTP that are related to reviewing the implementation (code) of the application block. This is an important step in unit testing the application block. By conducting an extensive code review, you can detect errors early in the life cycle. If you do not detect errors until the white box testing stage, correcting them may require significant and costly changes to the system, which may lead to a cascading effect of code changes between various modules.

The implementation of the application block is reviewed for the following:

- The code adheres to the design that was laid out in Step 1.

- Naming guidelines for code elements (class names, variables, method names, and assemblies, and so on) are followed.
- The code has comments at appropriate places to help customers understand the application block so they can easily customize it.
- The code follows best practices related to the following:
  - Performance and scalability
  - Writing secure code
  - Ensuring that relevant design patterns are being followed and looking for any possible anti-patterns in the code
  - Exception management
  - Resource management (memory allocation, connection pooling, object pooling, and so on)
  - Globalization and localization
- The block does not contain redundant or commented code that is never called.

The code review also helps in identifying the possible scenarios that must tested for one or more of the following:

- Boundary level conditions
- Special inputs
- Possible security attacks
- Performance optimizations
- Profiling to ensure that there are no memory leaks
- Thread safety and deadlock issues

**Note:** You must update the DTP and DTC to document these possible scenarios.

### More Information

For more information about implementation review of the application blocks, see Chapter 5, "Code Review for Application Blocks."

## Step 4: Perform Black Box Testing

In this step, you execute the test cases in the DTP that are related to black box testing of the application blocks. Black box testing is testing without knowledge of the internal workings of the system that is being tested. This type of testing simulates end-user experience.

Black box testing helps to ensure that the application block:

- Meets all of the objectives listed in the requirements document.
- Covers all of the functionalities that are specified in the functional specifications.
- Can gracefully handle all of the expected and exceptional usage scenarios. The error messages displayed are meaningful and help the user in diagnosing the problem.

You may need to develop one or more of the following to test the functionality of the application blocks:

- Test suites, such as NUnit, to test the API of the application block for various types of input.
- Dummy Windows forms or Web Forms applications that integrate the application blocks and are deployed in simulated target deployments.
- Automated scripts that test the APIs of the application blocks for various types of input.

The black box testing assumes that the tester has no knowledge of code and is intended to simulate the end-user experience. You can use sample applications to integrate and test the application block for black box testing. This approach tests all possible combinations of end-user actions. The test planning for black box testing can begin as soon as the requirements and the functional specifications are available. Black box testing includes the following:

- **Testing all of the external interfaces for all possible usage scenarios.** This means testing all of the external interfaces (such as public classes rather than the private or internal classes) that end users can integrate with their applications. Testing the external interfaces involves the following:
  - **Ensuring that the interfaces meet the functional specifications and address all of the requirements.** This type of testing ensures that the application block provides interfaces for meeting all of the requirements as stated in the functional specifications. This type of testing requires that you develop test suites. You must test for all of the ways in which clients of the application block can call the APIs. The usage scenarios include both the expected process flows and random input.
  - **Testing for various types of input.** This type of testing ensures that the interfaces return the output as expected and are robust enough to handle invalid data and exceptional conditions gracefully. The input data can be randomly generated either within a specified range expected by the application block, outside the specified range, or at the boundary of the range (known as boundary testing). Testing with data outside the specified range ensures that the application block is robust and can handle invalid data, and that the error messages generated are meaningful for the end user. Boundary testing ensures that the highest and lowest permitted values for input produce expected output.

- **Performance testing.** You execute performance-related test cases from the DTP in a simulated environment that is close to the real-world deployment. Performance testing verifies that the application block can perform under expected normal and peak load conditions, and that it can scale sufficiently to handle increased capacity. There are two main aspects of performance testing: load testing and stress testing. Each aspect has different end goals. You must plan and execute test cases for both aspects as follows:
  - **Load testing.** Use load testing to verify the application block behavior under normal and peak load conditions. Load testing enables you to verify that the application block can meet the performance objectives and does not overshoot the allocated budget for resource utilization such as memory, processor, and network I/O. It also enables you to measure response times and throughput rates for the application that is using the application block.

    Load testing also helps you to identify the overhead (if any, in terms of resource cost such as processor, memory, disk i/o or network i/o) of using the application block to achieve a desired functionality by testing applications with and without the application block to achieve the same result.
  - **Stress testing.** Use stress testing to evaluate the application block's behavior when it is pushed beyond peak load conditions. The goal of stress testing is to identify errors that occur only under high load conditions — for example, synchronization issues, race conditions, and memory leaks.

  The analysis from performance tests can serve as input for the implementation review and white box testing. If you detect a problem, you may need to review the code of the module you suspect to isolate the cause. (For example, if you have a problem of increased response wait time, a code review may reveal that a coarse-grained lock is causing the problem). During the white box testing stage, you can profile the suspected code path and optimize it.
- **Security testing.** Security testing from the black box perspective is done by simulating the target deployment environment and performing tests that simulate the actions an attacker can take in a real-world production environment.

You identify vulnerabilities in the application block (and hence expose threats) by attempting to simulate all possible types of input. The goal is to expose (at run time) any threats that may exist in the code because of poor design or poor coding practices. The scope of security testing for an application block is different from that for a real-world application and depends on the type of functionality that the application block provides. You can execute tests to:

- Check input validation techniques, if the application block has functionality that validates input data. You can provide various types of input with the objective of breaking the validation technique.
- Break the encryption and access sensitive data, if the application block handles sensitive data and uses encryption.

  - Check for buffer overflows.
  - Check for cross-site scripting errors.
  - Validate authorization and authentication mechanisms, if the application block provides this functionality.
- **Globalization testing.** The design and implementation have already been reviewed for adherence to best practices for globalization. For black box testing, you execute the globalization testing – related test cases from the DTP to ensure that the application block can handle international support and is ready to be used from various locales around the world.

  The goal of globalization testing is to detect problems in application design that could interfere with globalization. Globalization testing ensures that the code can handle all international support and that it supports any of the culture/locale settings without breaking functionality that would cause either data loss or display problems.

  To perform globalization testing, you must install multiple language groups and ensure that the culture/locale is not your local culture/locale. Executing test cases in both Japanese and German environments, for example, can cover most globalization issues.

### More Information

For more information, see the following resources:

- For more information about performance testing the application block, see Chapter 8, "Performance Testing for Application Blocks."
- For more information about globalization testing the application block, see Chapter 7, "Globalization Testing for Application Blocks."
- For a discussion of globalization, localizability, and localization testing, see "Globalization Testing" in the Microsoft® Visual Studio .NET documentation at: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsent7/html/vxconTestingForGlobalizationLocalization.asp.*
- For a checklist of globalization best practices, see "Best Practices for Globalization and Localization" in the Visual Studio .NET documentation at: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsent7/html/vxconBestGlobalizationPractices.asp.*

## Step 5: Perform White Box Testing

In this step, you execute the test cases in the DTP that are related to white box testing (also known as glass box, clear box, or open box testing) of the application blocks. White box testing derives the test cases and the related test data by analyzing the internal workings and program logic of the system.

White box testing assumes that the tester can look at the code for the application block and create test cases that test for any potential failure scenarios. This exercise is done to some extent in Step 3, where you can identify the potential scenarios for additional testing.

To determine what input data to use to test various APIs and what special code paths you must test, you analyze the source code for the application block. Therefore, you must update the test plans before you begin the white box testing.

The failure of a white box test may result in a change that requires you to repeat all of the black box testing and to redetermine the white box paths.

White box testing includes the following:

- **Profile the application block.** Profiling is the activity that allows you to monitor the behavior of a particular code path at run time when the code is actually being executed. You can profile your application block for one or more of the following:
  - **Code coverage.** Profiling for code coverage with the help of a tool ensures that your application does not contain redundant or commented code that is never called for any execution path.
  - **Memory allocation pattern.** Profiling the memory allocation helps you in studying direct and indirect memory allocations while executing a code path and detecting and analyzing memory leaks and other problems. For example, you can determine which generations (Gen 0, Gen 1, Gen2) contain the maximum number of objects, or whether any side effect allocations are taking place in a loop and are increasing the memory utilization significantly. (For example, using a string class for concatenation in a loop can cause many additional string object allocations.) Doing so can help you to optimize the memory utilization of the application block. You can use various profiling tools, such as CLR Profiler and Intel VTune, to analyze the memory allocation.
  - **Contention and deadlock issues.** You can analyze the application block for deadlocks by using tools such as WinDbg from the Microsoft Windows debugging toolkit.
  - **Time taken to execute a code path.** You can profile the time taken by the scenarios for which performance is critical. This profiling may require custom instrumentation of the code path, which may be distributed across separate computers. Various third-party tools are also available to help you measure the time taken to execute a particular scenario.

    If the application block needs to be integrated with a Web application, you may also want to monitor Request Execution Time (**ASP.NET\Request Execution Time** performance counter) for a test suite that uses the application block. The overhead of the suite should be minimal, and most of the processing time should be spent within the application block.

If, during profiling, you find that a particular condition must be tested under load (such as potential deadlock issues or inefficient memory cleanup) you can provide input on the type of metrics that should be used while load testing the application block:

- **Test internal subroutines.** You must test the various internal functions to ensure that they process the right data without any loss or inconsistency and return the expected output.
- **Test various loops and conditional statements.** You should test various types of loops (for example, simple, concatenated, and nested loops) and conditional statements (for example, relational expressions, simple conditions, compound conditions, and Boolean expressions) in the code components for accuracy.
- **Test various scenarios identified in Step 3.** You should test the scenarios you identified in the implementation review (Step 3). Analyze the code, and then submit the input to identify weaknesses in the code.
- **Test for security.** If your source code review reveals that a particular code access security configuration cannot restrict undesirable access in the targeted deployment environment, you should test the particular feature consuming the code path. Analyze the code, and then simulate the deployment environment to identify any scenarios in which the application block may expose sensitive information to attackers.

### More Information

For more information on white box and black box testing of the application block, see Chapter 6, "Black Box and White Box Testing for Application Blocks."

For a discussion of code coverage analysis, see "Perform Code Coverage Analysis with .NET to Ensure Thorough Testing" in MSDN Magazine at: *http://msdn.microsoft.com/msdnmag/issues/04/04/CodeCoverageAnalysis/default.aspx.*

## Summary

The chapter presents a process for testing an application block. The concepts that are outlined in the process apply regardless of the software development life cycle approach followed. The amount of effort and the number of iterations depend on the risk analysis of the overall significance of the application block at the macro level and each of the features at the micro level.

# Appendix

## Sample Test Case from a Detailed Test Plan Document

| | | |
|---|---|---|
| Scenario 1 | | Developing and executing NUnit test suite to test the various public classes available for the application block. |
| Priority | | High |
| Comments | | |
| 1.1 | High | Developing and executing NUnit test suite to test the reading and writing to the registry, SQL Server, and XML file storage media by using the Configuration Management Application Block. |

## Sample Test Case from a Detailed Test Case Document

| Test case | Priority | Condition to be tested | Execution details | Data required | Expected results | Actual result | Test OK (Y/N) |
|---|---|---|---|---|---|---|---|
| 1.1 | High | Developing and executing NUnit test suite to test reading and writing to the registry, SQL Server, and XML file storage media by using the Configuration Management Application Block. | 1. Create an assembly, ConfigurationManagement.UnitTests.<br><br>2. Create the following .cs files in the above assembly:<br>(Note that the static public methods of ConfigurationManager will be used for the read and write operations.)<br>XmlFileConfigurationManagement.cs<br>In this class, implement methods to read and write a custom object to an XML file for all combinations of the provider and cache settings for encryption and signing.<br>RegistryConfigurationManagement.cs<br>In this class, implement methods to read and write a custom object to the registry for all combinations of the provider and cache settings for encryption and signing.<br><br>SqlServerConfigurationManagement.cs<br>In this class, implement methods to read and write a custom object to a SQL Server database for all combinations of the provider and cache settings for encryption and signing.<br><br>Performance.cs<br>In this class, write methods to:<br>—Test the response time per read or write request for a large number of requests.<br>—Test the total number of requests (for read and write) served in a second.<br><br>After writing the methods, test the above two cases for all of the persistent media, XmlFileStorage, RegistryStorage, and SqlServerStorage.<br><br>Stress.cs<br>In this class, write methods to:<br>-Test multiple read and write from multiple threads,<br>-Read and write bulky objects<br><br>After writing the methods, test the above two cases for all of the media, XmlFileStorage, RegistryStorage and SqlServerStorage<br><br>3. Compile the above assembly.<br>4. Run NUnit on this assembly. | Configuration Management Application Block, NUnit | All NUnit test cases should run successfully for ConfigurationManagement.UnitTests.dll. | | |

# 4

# Design Review for Application Blocks

## Objectives

- Learn the process for design review of a .NET application block.

## Overview

The key to an efficient application block is a design based on sound principles. The goal of the design review is to ensure that the design for the .NET application block addresses all of the requirements and deployment scenarios. The reviewers should make sure that the design makes appropriate tradeoffs based on best practices and established principles for performance, scalability, security, maintainability, and so on. Design review is very important to the overall productivity of the project because it allows you to discover design errors early, rather than during development or testing, which can result in significant rework, as well as cost and schedule overruns.

This chapter describes the design review process for a .NET application block. To provide design review examples, this chapter refers to the Configuration Management Application Block (CMAB). The requirements for the CMAB are as follows:

- Must read and store configuration information transparently in a persistent storage medium. The storage media are Microsoft® SQL Server™, the registry, and XML files.
- Must provide a configurable option to store the information in encrypted form and in plain text by using XML notation.
- Can be used with desktop applications as well as Web applications that are deployed in a Web farm.

- Must cache configuration information in memory to reduce cross-process communication, such as reading from any persistent medium. This reduces the response time of the request for any configuration information. The expiration and scavenging mechanism for cached-in-memory data is similar to the cron algorithm in UNIX.
- Can store and return data from various cultures/locales without any loss of data integrity.

# Design Review Process

The process for the design review of application blocks is outlined below.

## Input

The following input is required for the design review:

- Application requirements
- Performance model
- Design-related documents (for example, architecture diagrams, class interaction diagrams, and so on)
- Deployment scenarios
- Security policy document

## Steps

The design review consists of the following steps:

1. Create test plans.
2. Verify that the design addresses all functional specifications and requirements.
3. Verify that the design addresses all deployment scenarios.
4. Verify that the design considers the performance and scalability tradeoffs and best practices.
5. Verify that the design considers the security tradeoffs and best practices.
6. Verify that the design employs best practices and principles for ease of maintenance and customization.
7. Verify that the design addresses globalization issues for the specified cultures.
8. Verify that the design has an effective exception management strategy.

The following sections cover each step in detail.

## Step 1: Create Test Plans

As a first step, create detailed test plans (DTP) and detailed test cases (DTC) for the design review. When you create these plans, use various perspectives, such as performance, security, exception management, and so on.

Table 4.1 shows some of the test cases that were used in the design review perspective for the CMAB. These test cases are based on the first iteration of test planning. As your design progresses, you may have test cases created to evaluate several prototypes.

**Table 4.1: Test Cases from DTP Document for CMAB Design Review**

| Scenario 1 | | Conduct design review of the CMAB |
|---|---|---|
| Priority | | High |
| Comments | | |
| 1.1 | High | Verify that the design addresses all of the requirements of the CMAB. |
| 1.2 | High | Verify that the design address all of the deployment scenarios for the CMAB. |
| 1.3 | High | Verify that the design follows the best practices and considers the tradeoffs for better performance and scalability. |
| 1.4 | High | Verify that the design follows security best practices. |
| 1.5 | High | Verify that the design addresses the globalization requirements. |
| 1.6 | Medium | Verify that the design considers the globalization best practices. |
| 1.7 | High | Verify that the design follows exception management best practices. |
| 1.8 | High | Verify that the design follows maintainability and extensibility best practices. |

### More Information

For more information about how to create test plans for an application block, see Chapter 3, "Testing Process for Application Blocks."

## Step 2: Verify That the Design Addresses All Functional Specifications and Requirements

The first important step in the design review process is to make sure that the design follows the functional specifications and meets all requirements. This is critical to the later stages of development because missing functionality can seriously restrict the usefulness of the application block.

It is helpful if the requirements and functional specification documents are translated into checklists. The reviewers can then work through these checklists to ensure

that each specified function and requirement is translated into a design feature for the application block. A sample checklist for CMAB is shown in Table 4.2.

**Table 4.2: Sample Checklist for CMAB**

| Functional requirement | Implemented? | Feature that implements requirement |
|---|---|---|
| The CMAB must provide a standard interface for reading and writing application configuration data. The interface must be easy to use and independent of the underlying data store. | Yes | Custom storage provider (CSP) that implements either the **IConfigurationStorageWriter** or the **IConfigurationStorageReader** interface. |
| The CMAB must allow programmers to work with configuration data using the most appropriate in-memory data structure. | Yes | Custom section handler (CSH) that implements two interfaces: **IConfigurationSectionHandler** and **IConfigurationSectionHandlerWriter**. |
| The CMAB must be easily configurable using standard .NET configuration files. | Yes | **Configuration Manager** class<br><br>Configuration settings in the App.config/Web.config file:<br><applicationConfigurationManagement<br>defaultSection="XXXXXX " ><br>……..<br></applicationConfigurationManagement> |
| The CMAB must support the optional caching of application configuration data to improve performance. The use of caching must be controlled by configuration settings, independent of the underlying data store, and transparent to the application that is using the CMAB. | Yes | **CacheFactory** class<br><br>Configuration settings in the App.config/Web.config file:<br><configCache<br>enabled="true"<br>refresh="1 * * * *" /> |
| The CMAB must support the optional encryption and decryption of application configuration data to improve data security. The use of encryption must be controlled by configuration settings, independent of the underlying data store, and transparent to the application that is using the CMAB. | Yes | Data protection provider (DPP) that implements the **IDataProtection** interface.<br><br>Configuration settings in the App.config/Web.config file:<br><protection Provider assembly="…"<br>type="…"<br>hashKey="…"<br>symmetricKey="…"<br>initializationVector="…"/> |

| Functional requirement | Implemented? | Feature that implements requirement |
|---|---|---|
| The CMAB must provide a set of static methods so that the user does not have to instantiate any objects before use. | Yes | **public static object Read**( **string** sectionName ) {….}<br><br>**public static void Write**(**string** sectionName, **object** configValue) {……} |
| Most applications use a single (default) configuration section; therefore the CMAB should provide a simpler interface for interacting with the default configuration section. | Yes | Configuration settings in the App.config/Web.config file:<br><br><applicationConfigurationManagement defaultSection="XXXXXX " ><br>…….<br></applicationConfigurationManagement><br><br>**public static void Write**( **Hashtable** value ) {…}<br>**public static HashTable Read**() {……} |

## Step 3: Verify That the Design Addresses All Deployment Scenarios

When reviewing the design for the application block, make note of the deployment scenarios in which the application block will be used. The design may assume certain features that are not available in all deployment situations. This could make the application block unusable for some of the deployment scenarios listed in the requirements.

For example, the CMAB can be deployed on a desktop as well as in a Web farm. You should ensure that the design does not use machine-specific encryption/decryption services, such as the Data Protection API, a service that is specific to the Microsoft Windows® 2000 operating system. The Data Protection API encrypts data by using machine-specific keys so that the data cannot be decrypted on other computers. For this reason, using this service could make the application block unsuitable for deployment in a Web farm. To correct this, the application block could use a machine-agnostic service, such as the **System.Security.Cryptography** namespace in the Microsoft .NET Framework Class Library.

As an alternative solution, the CMAB could provide multiple options for encrypting the data so that the user can choose the one that is most appropriate for his or her scenario. The encrypting option should be configurable in the configuration file for the application block.

Multiple deployment scenarios require that you make tradeoffs with respect to performance and security. If these tradeoffs are not determined in the design phase,

then the application block may be too slow or may be a threat to the security of the network on which it is deployed. (These tradeoffs are validated more thoroughly in the Steps 4 and 5, where you will perform the performance and security design reviews.)

For example, with CMAB, you should carefully review the following deployment considerations:

- CMAB caches the configuration data in memory. There may be scenarios where the available memory is reduced because of an increased amount of data cached in memory. This reduction in available memory may cause a performance bottleneck under high load situations on a Web farm or even on a desktop application that is required to have only a small working set. The design must address this possibility by providing an expiration mechanism that flushes the cache if the memory utilization crosses a particular threshold.
- CMAB is required to write to SQL Server, which may be deployed on a separate computer. This may require the opening of SQL Server – specific TCP/IP ports in the internal firewall. Some networks may limit communication to HTTP port 80 on the internal firewall. If this is one of your deployment scenarios, the CMAB must have another abstraction layer installed on the computer running SQL Server so that it can capture information over port 80, and then store it in the database.

The deployment scenarios should be treated as deployment-related requirements. The documents that can help you to capture these scenarios are the functional requirements document, deployment topologies diagrams, documented deployment scenarios, and so on.

You must carefully review each deployment scenario, and then analyze its potential impact on the design. This review process ensures that any impact is reflected in the design model for the application block.

You may need to develop a simple code prototype at this stage, and then test it in a simulated deployment scenario to ensure that you have not overlooked any implications of a particular scenario.

### More Information

- For more information about security-related deployment considerations, see Chapter 4, "Design Guidelines for Secure Web Applications," in *Improving Web Application Security: Threats and Countermeasures* on MSDN at: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnetsec/html/THCMCh04.asp*.

- For questions about architecture and design review with respect to security-related design tradeoffs for deployment considerations, see Chapter 5, "Architecture and Design Review for Security," in *Improving Web Application Security: Threats and Countermeasures* on MSDN at: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnetsec/html/THCMCh05.asp.*
- For more information about performance-related deployment considerations, see Chapter 3, "Design Guidelines for Application Performance," in *Improving Web Application Security: Threats and Countermeasures* on MSDN at: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/scalenetchapt03.asp.*
- For questions about architecture and design review with respect to performance-related design tradeoffs for deployment considerations, see Chapter 4, "Architecture and Design Review of a .NET Application for Performance and Scalability," in *Improving .NET Application Performance and Scalability* on MSDN at: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/scalenetchapt04.asp.*

## Step 4: Verify That the Design Considers Performance and Scalability Tradeoffs and Best Practices

Performance and scalability should be considered early in the design phase of any product. It is very important that you review the design from this perspective, or the final product may consume more resources than allocated and may fail to meet performance objectives. In addition, the application block may not be able to handle the targeted work load. Performance objectives and resource utilization limits should be available in the requirements document or the performance model.

The performance and scalability review can be divided into the following steps. You may need to develop prototype code and then test the prototype in a simulated environment to validate the design assumptions.

1. **Verify that the design can meet the performance objectives using the allocated resources (CPU, memory, network I/O, and disk I/O).**

   You may need to break a single usage scenario into smaller steps, and then allocate resources to these smaller steps. Validate the allocated budget by prototyping and testing it in a simulated environment.

   **Note:** This step is an important activity that is covered primarily in the performance modeling phase. If you do not plan to have an explicit performance modeling phase, you should still perform this exercise if you have any resource constraints.

For more information about performance objectives and performance modeling, see Chapter 2, "Performance Modeling," in *Improving .NET Application Performance and Scalability* on MSDN at: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/scalenetchapt02.asp.*

For more information about how to conduct performance testing, see Chapter 16, "Testing .NET Application Performance," in *Improving .NET Application Performance and Scalability* on MSDN at: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/scalenetchapt16.asp.*

2. **Verify that the design addresses performance tradeoffs for various deployment scenarios.** For example, CMAB caches configuration data in physical memory. Available memory on the computer can be reduced because of the amount of data cached in memory. This reduction in available memory may cause a performance bottleneck under high load scenarios on a Web farm. The design must address this issue by providing an expiration mechanism that flushes the cache if memory utilization crosses a particular threshold value.

   For more information about performance-related deployment considerations, see Chapter 3, "Design Guidelines for Application Performance," in *Improving .NET Application Performance and Scalability* on MSDN at: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/scalenetchapt03.asp.*

   For more information about the performance-related design tradeoffs and deployment considerations, see Chapter 4, "Architecture and Design Review of a .NET Application for Performance and Scalability," in *Improving .NET Application Performance and Scalability* on MSDN at: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/scalenetchapt04.asp.*

3. **Verify that the design follows principles and best practices for performance and scalability.** As mentioned previously, performance and scalability best practices focus on using technology-agnostic principles in your design. Some important design principles are as follows:

   - **Minimize transitions across boundaries.**

     Try to keep transitions across boundaries to a minimum. Boundaries include application domains, apartments, processes, and computers. Each time a block must cross a boundary, it incurs costs in terms of security, serialization of data, additional latency involved in crossing the boundary, and so on.

   - **Use early binding where possible.**

     Application blocks that use late binding may be flexible, but late binding can affect performance because of the overhead involved in dynamically identifying and loading assemblies at run time. Most scenarios that use late binding provide configurable settings in a file.

- **Pool shared or scarce resources.**

  Resources that are limited in number and are essential to optimized functioning of the application block must be treated as shared resources, and no single piece of code should hold on to them for a long time. Resources, such as database connections, threads, and so on, fall under this category. Review the application block design to locate places where thread creation can be replaced by allocating the parallel tasks to the CLR thread pool.

  For example, consider that the **GetAltitudeData** and **GetVelocityData** methods must be executed in parallel:

```
public void GetFlightData(){
    GetAltitudeData(); //Executed in the current thread
Thread seperateThread = new Thread(new ThreadStart(GetVelocityData));
    seperateThread.Start();
}
```

  Rather than allocating a thread from the process thread pool, the**GetAltitudeData** method creates a new thread. This parallel execution of tasks defeats the concept of sharing resources.

  The following code shows how to use shared threads in the thread pool. Here the new thread is allocated from the thread pool; therefore, thread management is optimized and is taken care of by the .NET Framework.

```
public void GetFlightData(){
GetAltitudeData(); //Executed in the current thread
ThreadPool.QueueUserWorkItem(new WaitCallBack(GetVelocityData));
}
```

- **Partition functionality into logical layers.**

  Keep the methods that perform similar functions or act on the same data in the same class. Classes that are similar in functionality should be in the same assembly. In this way, closely related classes and data are located near each other, which reduces boundary transitions. Grouping data logically increases both performance and maintainability.

- **Reduce contention by minimizing lock times.**

  Review the transactions and the granularity of the locks used in the application block. Lock times can introduce resource contention, and, as a result, the application block may not be able to meet performance objectives.

For more information about performance-related design guidelines, see Chapter 3, "Design Guidelines for Application Performance," in *Improving .NET Application Performance and Scalability* on MSDN at: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/scalenetchapt03.asp.*

For questions about architecture and design review for performance, see Chapter 4, "Architecture and Design Review of a .NET Application for Performance and Scalability," in *Improving .NET Application Performance and Scalability* on MSDN at: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/scalenetchapt04.asp.*

For a checklist on architecture and design review of .NET applications from a performance perspective, see "Checklist: Architecture and Design Review for Performance and Scalability," in *Improving .NET Application Performance and Scalability* on MSDN at: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/ScaleNetCheck02.asp.*

4. **Verify that the design follows technology-specific best practices.** The application block design should follow the best practices for .NET-related technologies. The design should instantiate .NET-specific design patterns for efficient resource cleanup, patterns for reducing the chattiness, and so on. For example, if the application block has a class that accesses an unmanaged COM component across multiple methods, the class must implement the *Dispose* pattern to allow the clients of the API to deterministically release unmanaged resources.

   If you develop a custom collection for value types and require frequent enumeration through the collection, you should implement the *Enumerator* pattern to reduce the overhead of virtual table lookup and boxing or unboxing.

   For more information about performance-related best practices, see the Part III, "Application Performance and Scalability," and the Checklists section of *Improving .NET Application Performance and Scalability* on MSDN at: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/ScaleNet.asp,*

5. **Verify that the design features do not compromise security for the sake of performance.** During the design review process, you should identify the situations where the cost of security is significant and is hurting performance. However, make sure that you do not compromise security for the sake of performance. You should make a list of all such situations and carefully analyze them during the security review of the design. Analyzing these situations helps you make appropriate tradeoffs for performance and security.

   For example, if during prototype testing for the CMAB you find that frequent encryption and decryption causes excessive processor cycles, you can mitigate

the performance impact by scaling up or out instead of eliminating data encryption. In other situations, you should do a threat analysis before lowering the security barrier.

### More Information

For more information about performance-related best practices for .NET technologies, see *Improving .NET Application Performance and Scalability*, on MSDN at: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/ScaleNet.asp*.

## Step 5: Verify That the Design Considers the Security Tradeoffs and Best Practices

Reviewing the application block design from the security perspective helps you to identify and understand security issues that may surface during later stages of the application block development life cycle. The cost and effort involved in fixing such issues increases as development progresses.

The design review should ensure the following:

- Plugging the application block into an application does not introduce any security flaws in production.
- The design evaluates the security tradeoffs and follows the best practices that lead to a secure design.

You should consider the following when you review the design:

- Make sure that the application block design identifies the trust boundaries. Verify that the design identifies privileged code and uses sandboxing for the privileged code. (Sandboxing puts the privileged code in a separate assembly.)
- Make sure that there is no dependency on a particular service or protocol that may not be available in a production environment.
- Make sure that the application block has access to required resources so that it can provide the intended functionality. Ensure that all such resources can be made available in production scenarios without compromising security.
- Make sure that the application block does not depend on special privileges or rights. The block should be designed to function with least privileges. If the application block is going to need special privileges, the design should evaluate and address the possible security tradeoffs.
- Determine how the application block handles sensitive data. Does the block use cryptography? Does it follow best practices while doing so? Wherever possible, make sure that the design uses the classes available in the .NET Framework for cryptography.
- Make sure that the design follows best practices for class design. For example, the design should restrict class and member visibility; the design should follow basic guidelines, such as sealing of non-base classes; and so on.

To ensure that the application block design is secure, consider the following best practices (as appropriate, depending on the functionality provided by the block):

- **Identify and review any areas that may be susceptible to SQL injections.**

  If the application block is going to perform database access, make sure that the design ensures the correct use of data type, size, and format of the parameterized SQL queries (if used). Although stored procedures are used as risk mitigation for SQL injection attacks, they should be reviewed for any other possible security threats.
- **Review the areas where the application block accesses the file system or any other resources.**

  Identify areas in the design where file access is required. Ensure that an authorization strategy is in place to access these resources.
- **Review the areas where sensitive data is used or passed.**

  Review those portions of the design that pass, process, or store sensitive data such as user names, passwords, personal information, and so on. Look for the encryption strategy used.
- **Review exception management.**

  Review the exception handling in the application block. Improper exception handling (unstructured) could lead to open connections or open files that are never closed, which could in turn lead to security attacks.

### More Information

For an exhaustive list of all design review considerations and checklists, see Chapter 5, "Architecture and Design Review for Security," in *Improving Web Application Security: Threats and Countermeasures* on MSDN at: *http://msdn.microsoft.com/library/en-us/dnnetsec/html/THCMCh05.asp*.

## Step 6: Verify That the Design Uses Best Practices for Ease of Maintenance and Customization

Another area that is of critical importance to the users of an application block is the ease with which it can be customized and maintained after integration into an application. When completing your design review, consider the manageability of the code. Ensure that the design addresses the following best practices:

- The design follows object-oriented principles so that the classes own the data on which they act.
- The communication among various objects uses message-based communication instead of following the remote procedure call (RPC) model. Message-based

communication requires that the communicating entities define and communicate by using a data contract. The use of a data contract ensures a high degree of encapsulation, thereby minimizing the impact that a change in one component can have on other components.

- The data format for input parameters and return values is consistent.
- The design follows a pattern-based approach for addressing various problems. A pattern-based approach ensures that, besides using a tried and tested solution to specific problems, developers can easily grasp the algorithm for a particular implementation.
- The logic that enforces policies or provides complimentary features such as security, encryption, communication restriction, and so on, is abstracted as much as possible from the core functionality.

## More Information

For more information, see the following resources:

- For more information about designing applications, see *Application Architecture for .NET: Designing Applications and Services* on MSDN at: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/distapp.asp.*
- For data-related problems, see *Data Patterns* on MSDN at: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpatterns/html/Dp.asp.*
- For a repository of patterns that help you solve various problems, see *Enterprise Solution Patterns Using Microsoft .NET* on MSDN at: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpatterns/html/Esp.asp.*
- For more information about the *Factory* design pattern, see "Exploring the Factory Design Pattern" on MSDN at: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/factopattern.asp.*
- For more information about the *Observer* design pattern, see "Exploring the Observer Design Pattern" on MSDN at: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/observerpattern.asp.*
- For more information about the *Singleton* design pattern, see "Exploring the Singleton Design Pattern" on MSDN at: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/singletondespatt.asp.*

## Step 7: Verify That the Design Addresses Globalization Issues for the Desired Cultures

Globalization is the process of ensuring that a piece of software supports localized versions of user interface elements, such as messages, and can operate on regional data.

Before starting the design review, you know all of the cultures that the application block is required to support. The common issues related to globalization generally entail features such as string operations, formatting date and time in specific cultures, using calendars in specific cultures, comparing and sorting data in specific cultures, and so on.

Most of these issues can be addressed during the design process so that minimal changes are required to localize the application block for a particular region. When reviewing the application block design, verify the following:

- All resources (error messages, informational messages, or any other nonexecutable data) that are used in the application block are stored in a resource file.
- Any objects that need to be persisted in resource files are designed to be serializable.
- If the application block has to be packaged with resources for different cultures, one resource file should be the default file and the other files should be contained in separate satellite assemblies. The default resource assembly is a part of main assembly.
- Any classes that perform culture-specific operations should explicitly use the culture-aware APIs included in the .NET Framework. The classes should expose overloaded methods that accept culture-specific information. Review that the **CultureInfo** type and the **IFormatProvider** type are used to specify the culture in the overloaded methods for which the culture-specific operation must be performed.
- The application block should primarily use Unicode encoding, although other types of encoding (such as double byte character sets [DBCS]) are also available. If you plan to use a database, such as SQL Server 2000, you should use Unicode-compatible data types for database design. For example, CMAB stores configuration information in a SQL Server database. This stored information can be from different character sets; therefore, you should review that the database design uses data types such as **nchar**, **nvarchar**, **ntext**, and so on.

### More Information

- For more information about globalization testing of an application block, see "Globalization Testing" on MSDN at: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsent7/html/vxconglobalizationtesting.asp.*
- For more information about satellite assemblies, see "Creating Satellite Assemblies" on MSDN at: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconcreatingsatelliteassemblies.asp.*
- For more information about best practices for globalization, see "Best Practices for Developing World-Ready Applications" on MSDN at: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconbestpracticesforglobalapplicationdesign.asp.*
- For more information about best practices for globalization and localization, see "Best Practices for Globalization and Localization" on MSDN at: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsent7/html/vxconBestGlobalizationPractices.asp.*
- For more information about issues related to globalization and localization, see "Globalization and Localization Issues" on MSDN at: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsent7/html/vxconGlobalizationLocalization.asp.*

## Step 8: Verify That the Design Has An Effective Exception Management Strategy

An effective exception management strategy for an application block ensures that the block is capable of detecting exceptions, logging and reporting errors and warnings, and generating events that can be monitored externally.

When you review the application block design, verify that the design adheres to the following basic principles of effective exception management:

- Instead of throwing generic exceptions, the application block should use specific classes for throwing specific types of exceptions. These exceptions should derive from the **ApplicationException** class. In this way, the clients of the application block can determine the appropriate action if an exception of a particular type occurs. For example, if the CMAB fails to read the configuration file during application initialization, it should throw an exception of type **ConfigurationException** or a class derived from **ConfigurationException**. The details of the error message can be written to the message property of the exception class.

Consider the following code snippet in an application block that accesses a SQL data store.

```
try{
    // Data access logic specific to the application block
}
catch (Exception e){
    //Perform some exception-related operation like logging or rollback
    //operations.
    throw e;
}
```

The preceding code snippet catches an exception and rethrows it to the calling code as an instance of the generic **Exception** class. This makes it difficult for the client to distinguish between types of exceptions and to have efficient application exception management logic in place. The preceding code snippet also introduces the redundant overhead of unnecessarily catching and rethrowing all of the exceptions without any value addition.

The following code snippet shows how CMAB handles a similar scenario.

```
try{
    // Data access logic specific to the application block
    rows = SqlHelper.ExecuteNonQuery( );

}
catch( Exception e)
{
    throw new ConfigurationException( Resource.ResourceManager[
"RES_ExceptionCantExecuteStoredProcedure",SetConfigSP, SectionName, paramValue,
e ] );
    // log the exception in the event sink
}
```

CMAB catches the exception that occurs in the data access logic with SQL Server, and throws the exception as a **ConfigurationException**, which is an exception of a specific type. Moreover, the exception is logged in an event sink before it is sent to the client of the application block.

You should review all sequence diagrams (if available) and determine the exceptional flows for each usage scenario. Also note any application block – specific exceptions that can occur during the execution of normal process flow. This list will help you to determine whether or not the design for exception management provides specific exception types where required. Using this approach ensures that the application block can generate meaningful exceptions.

- Throwing exceptions is expensive; therefore, exceptions should not be used to control application logic flow. Exceptions are best thrown under exceptional situations. For example, if the client of CMAB requests specific information for a key value and no information is returned, this is an expected situation. The

application block should not throw an exception. However, if the client requests information, and CMAB fails to connect to persistent storage (such as SQL Server) because of security issues (for example, an Access Denied error), this is a good case for throwing an exception.

Consider the following code snippet.

```
// Bad scenario to throw exception
static void ConfigExists( string KeyForConfig) {
    //... search for Key
    if ( dr.Read(KeyForConfig) ==0 ) // no record found
    {
        throw( new Exception("Config Not found"));
    }
}
```

These conditions should be handled by returning a simple Boolean value. If the Boolean value is true, the configuration information exists in the database.

```
static void ConfigExists( string KeyForConfig) {
    //... search for key
    if ( dr.Read(KeyForConfig) ==0 ) // no record found
    {
        return false;
    }
    . . .
}
```

- Another significant design decision is how functions should report exceptions to the caller. Functions should avoid using error codes as return values or out parameters for two main reasons. First, throwing exceptions ensures notification. Second, clients of an application block must customize their application exception management to suit the nonstandard technique of passing error codes.

  Consider the following code snippet.

```
class GetEmployeeDetails{
    public int GetEmployeeName(out string name){
        int errorCode = 0;
        try{
            // Data access logic specific to the application block
            name = SqlHelper.ExecuteNonQuery( );
        }
        catch( Exception e ) {
            // Perform some exception-related operation like logging or
            //rollback operations.
errorCode = 3;
        }
        finally{
            return errorCode;
        }
    }
}
```

```
public void DisplayName(){
    //Creating an instance of the class that contains the GetEmployeeName
    //method
    string employeeName;
    GetEmployeeDetails varDetails = new GetEmployeeDetails();
    int errorCode;
    try{
        errorCode = varDetails.GetEmployeeName(out emloyeeName);
        if(errorCode == 3){
            //throw error based on the return code
            //not intuitive
        }
    }
    catch(Exception e){
        //Exception Management Code for all the remaining errors
    }
}
```

In the preceding code snippet, an error code is returned instead of an exception. Therefore, the calling function from the client, **DisplayName**, must change its exception management to handle the error code from the application block.

Compare this to the following code where, instead of using error codes, the application block throws an exception, which simplifies the exception management code of the client.

```
public int GetEmployeeName(out string name){
    try{
        // Data access logic specific to the application block
        name = SqlHelper.ExecuteNonQuery( );
    }
    catch( Exception e ) {
        // Perform some exception related operation like logging or
        //rollback operations.
        throw new ConfigurationException(…..,e);
    }
}
public void DisplayName(){
    //Creating an instance of the class that contains the GetEmployeeName   \
    //method
    string employeeName;
    GetEmployeeDetails varDetails = new GetEmployeeDetails();
    try{
        varDetails.GetEmployeeName(out emloyeeName);
    }
    catch(ConfigurationException e){
        // more intuitive and standardized
        //Exception Management Code
    }
    catch(Exception e){
        //Exception Management Code
    }
}
```

- The design should use a flexible error-reporting mechanism. This mechanism should be configurable so that the client can determine the event sinks for logging of errors based on particular criteria, such as severity. For example, the default configuration of CMAB logs high severity errors in the event log and all the warnings and informational messages in Windows Event Trace.

  This configurable system should also provide the flexibility of turning the reporting mechanism on and off. This reporting system can be very beneficial when debugging an application block; however, because of the performance overhead, it may not be desirable in a production environment.
- The design for the application block should ensure that in addition to errors, other informational messages are logged and reported by raising appropriate events. These events are useful for monitoring the health of the application block. For example, the CMAB can report any request that holds locks for writing or updating information for more than 3 milliseconds.

### More Information

- For more information about exception management architecture, see the *Exception Management Architecture Guide* on MSDN at: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/exceptdotnet.asp.*
- For more information about design review of exception management from a security perspective, see Chapter 5, "Architecture and Design Review for Security," in *Improving Web Application Security: Threats and Countermeasures* on MSDN at: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnetsec/html/THCMCh05.asp.*
- For more information about design review of exception management from a performance perspective, see Chapter 4, "Architecture and Design Review of a .NET Application for Performance and Scalability," in *Improving .NET Application Performance and Scalability* on MSDN at: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/scalenetchapt04.asp.*

## Tools for Architecture and Design Review

The tools available for architecture and design review are as follows:

- **FxCop.** This tool can be used to check for compliance of code with various types of guidelines. However, FxCop is of limited help when you are reviewing the design documents for the application block.

  For more information about using FxCop and the rules it checks for compliance, see the FxCop Team Page at: *http://www.gotdotnet.com/team/fxcop/.*

- **Checklists.** There are checklists available for performing architecture and design review from the performance and security perspectives.

  For architecture and design review from a performance perspective, see the Checklist section of *Improving .NET Application Performance and Scalability* on MSDN at: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/ScaleNet.asp*.

  For architecture and design review from a security perspective, see the Checklist section of *Improving Web Application Security: Threats and Countermeasures* on MSDN at: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnetsec/html/threatcounter.asp*.

# Summary

The chapter presented a design review process for application blocks. The process attempts to ensure that all of the functional requirements have been addressed and that the design follows best practices related to performance, security, globalization, maintainability, and extensibility.

# 5

# Code Review for Application Blocks

## Objective

- Learn the process for reviewing the implementation of application blocks, including various parameters such as performance, security, exception management, and globalization and localization features.

## Overview

The implementation of application blocks should be validated against the various coding standards and best practices in the language that is used for developing the application block. The code should be reviewed from different perspectives, such as adherence to design and functional specifications, to ensure that it follows best practices related to parameters, including performance, security, globalization and localization, and exception management.

The examples in this chapter assume a design review of an application block named Configuration Management Application Block (CMAB). The requirements for the CMAB are the following:

- It provides the functionality to read and store configuration information transparently in a persistent storage medium. The storage mediums are SQL Server, the registry, and an XML file.
- It provides a configurable option to store the information in encrypted form and plain text using XML notation.
- It can be used with desktop applications and Web applications that are deployed in a Web farm.

- It caches configuration information in memory to reduce cross-process communication, such as reading from any persistent medium. This reduces the response time of the request for any configuration information. The expiration and scavenging mechanism for the data that is cached in memory is similar to the cron algorithm in UNIX.
- It can store and return data from various locales and cultures without any loss of data integrity.

# Application Block Code Review

This section describes the steps involved in performing a code review for an application block.

## Input

The following input is required for a code review:

- Requirements (use cases, functional specifications, deployment scenarios, and security-related requirements for the target deployments)
- Design-related documents (architecture diagrams and class interaction diagrams)

## Code Review Steps

The process for an application block code review is shown in Figure 5.1.

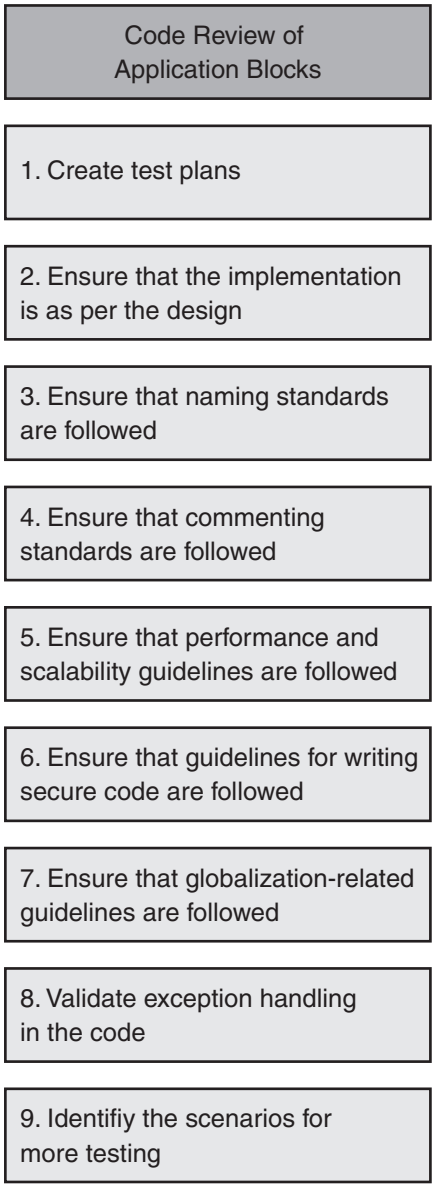


**Figure 5.1**
*The code review process for application blocks*

As shown in Figure 5.1, application block code review involves the following steps:

1. **Create test plans.** Create test plans that list all test cases and execution details from a code review perspective.

2. **Ensure that the implementation is in accordance with the design.** The implementation should adhere to the design decided on in the architecture and design phase.
3. **Ensure that naming standards are followed.** The naming standards for assemblies, namespaces, classes, methods, and variables should be in accordance with the guidelines specified for the Microsoft® .NET Framework.
4. **Ensure that commenting standards are followed.** The comments in the implementation should adhere to the standards for the language used for developing the application block.
5. **Ensure that performance and scalability guidelines are followed.** The code should follow the implementation best practices for .NET Framework. This optimizes performance and scalability.
6. **Ensure that guidelines for writing secure code are followed.** The code should follow the implementation best practices. This results in hack-resistant code.
7. **Ensure that globalization-related guidelines are followed.** The code should follow globalization-related best practices in such a way that the application block can be easily localized for different locales.
8. **Validate exception handling in the code.** The goal of exception handling should be to provide useful information to end users and administrators. This minimizes unnecessary exceptions at the same time.
9. **Identify the scenarios for more testing.** During the white box testing phase, identify the scenarios that need more testing.

The next sections describe each of these steps.

## Step 1: Create Test Plans

Create detailed test plans and detailed test cases for code review from various perspectives, including performance, security, and exception management.

Tables 5.1, 5.2, and 5.3 show the detailed test plan for the CMAB from the code review perspective.

**Table 5.1: Sample Test case from the detailed test plan document for the CMAB**

| | | |
|---|---|---|
| Scenario 1 | | To ensure the functional mapping toward the design specification document. |
| Priority | | High |
| Comments | | |
| 1.1 | Low | To ensure the functional mapping toward the design specification document. |

**Table 5.2: Sample Test case from the detailed test plan document for the CMAB**

| | | |
|---|---|---|
| Scenario 2 | | To use various tools against the application block. |
| Priority | | High |
| Comments | | |
| 2.1 | High | To validate the adherence to various programming practices by using FxCop to analyze various assemblies. |
| 2.4 | High | To verify that there are no politically incorrect instances in the comments by running an automated tool against the application block assemblies. |

**Table 5.3: Sample Test case from the detailed test plan document for the CMAB**

| | | |
|---|---|---|
| Scenario 3 | | To review the code of the application block using various guidelines prescribed by Microsoft for .NET development. |
| Priority | | High |
| Comments | | |
| 3.1 | High | To verify that the CMAB code follows the best practices and considers the trade-offs for better performance and scalability. |
| 3.2 | High | To verify that the CMAB code follows the best practices and trade-offs for security features. |
| 3.3 | High | To verify that the CMAB code considers the best practices for globalization. |
| 3.4 | High | To verify that the CMAB code follows the best practices for exception management. |

For more information about how to create test plans for the application block, see Chapter 3, "Testing Process for Application Blocks."

## Step 2: Ensure That the Implementation Is in Accordance with the Design

In typical development cycles, design precedes implementation. Developers write the source code in accordance with the design and the functional specification documents. The design and functional specification documents include the following:

- **High-level design documents.** These include documents that show the component-level view of the application block. The component-level view contains the specifications about the classes in the components, the public methods (along with their signatures) in the classes and the description of the all components, the

classes, and the public methods. The high level design document may also contain the description and flow of individual use cases.

- **Low-level design documents.** These include documents that detail information such as the class-level variables, some pseudo code to illustrate the algorithm for the functions within the class, entity relationship diagrams showing the relationship between various classes and components, and sequence diagrams showing the process flow for various use cases.
- **List of assumptions and constraints.** This list includes assumptions, design standards, system constraints, and implementation considerations.

During code review, you must ensure that the application block implementation is in accordance with the design and functional specifications. It is preferable to translate the design, requirements, and the functional specification documents into checklists. During code review, review these checklists to ensure that each of the classes and functions is in accordance with the specifications for the application block. The code should be modularized in a way that avoids code repetition for the same functionality.

For example, if you are doing the code review of the CMAB, you should watch out for the following:

- The code should maintain a design of inheritable interfaces to handle the different persistent mediums that are to be used to save the configuration settings.
- The code should consist of modules that specifically address every persistent medium that is listed in the functional specification document.
- The setting options in the configuration file for enabling and disabling caching in memory information should be implemented using parameterized methods. This ensures that a minimal amount of code can effectively address all the options and their implementations with appropriate options for scalability.

## Step 3: Ensure That Naming Standards Are Followed

A good naming style is one of the parameters that help the code to be consistent and identifiable. It allows quick walkthrough and easy understanding of the code, thereby helping redress the problems in the code structure. This helps achieve process-dependent maintainability instead of people-dependent maintainability. Any developer can easily understand the functionality from the functional specifications and read through the code to see the actual implementation. This helps when a developer other than the original author fixes bugs or adds additional functionality.

Some common points that need to be handled during a code review from the naming conventions perspective are the following:

1. **Capitalization styles.** Classes, events, properties, methods, and namespaces should use Pascal casing; whereas parameter names for functions should use Camel casing. Pascal casing requires the first letter in the identifier and the first letter of each subsequent concatenated word to be capitalized. For example, the class should be declared as shown in the following code.

   ```
   //It is recommended to name a class like this
   public ImplementClass
   ```

   The casing in the following code is not recommended for naming a class.

   ```
   //It is not recommended to name a class like this
   public implementclass
   ```

   Camel casing requires the first letter of an identifier to be lowercase. For example, a function that takes in parameters should have parameters declared as shown in the following code.

   ```
   void MyFunc(string firstName, string lastName)
   ```

   The casing in the following code is not recommended for the function arguments.

   ```
   //It is not recommended to name the function arguments like this
   void MyFunc(string FirstName, string LastName)
   ```

2. **Case sensitivity.** Check if all the code components are fully usable from both languages that are case-sensitive and languages that are not case-sensitive. Just because a particular language is case-insensitive, the naming conventions should not be a mix of different cases. Also, the identifier names should not differ by case alone. For example, you should avoid using the following declarations because a case-insensitive language cannot differentiate between them.

   ```
   string YourName
   string yourname
   ```

3. **Acronyms and abbreviations.** Well known and commonly used acronyms should be used in the code. However, the identifier names should not be made into abbreviations or parameter names as shown in the following code.

   ```
   //Use this declaration
   void ValidateUserIdAndPwd()
   //Instead of this
   void ValUIAP()
   ```

You can use Camel casing or Pascal casing for the identifiers and parameters that have acronyms that are longer than two characters. However, you should capitalize the acronyms that consist of only two characters as shown in the following code.

```
//Use this declaration
MyWebService.RA
//Instead of this
MyWebService.Ra
```

4. **Selection of words.** The words used for identifiers and parameter names should not be the same as standard words or key words in the .NET Framework class library.

For detailed information about naming guidelines and related checklists, see "Naming Guidelines" in the *.NET Framework General Reference* on MSDN at: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpgenref/html/cpconNamingGuidelines.asp.*

## Step 4: Ensure That Commenting Standards Are Followed

Commenting the code is one of the most important practices that significantly increases the maintainability and usability of the application block. The developer can read through the comments to understand and customize the code for the application block to suit the application needs.

Commenting the code is important, but it should not be redundant. A block of code that follows other coding standards does not need to have comments for every line of code and will be readily understandable even without extensive commenting. However, comments are important to give the necessary information about the purpose or methodology used by the logic. Some of the factors that need to be handled while doing the code review from the commenting standards perspective are the following:

- Make sure that all methods, classes, and namespaces have all the proper information such as comment headers, clearly specifying the purpose, parameters, exception handling, and part of the logic (if required) for that method.
- Make sure that there are comments for complex logic in all methods. Also make sure that these comments clearly explain the logic and its complexity. If required, you can use an example in some cases to help explain the logic.
- Make sure that there are appropriate comments to indicate why a particular variable is assigned a particular initial value. For example, if any integer variable is being initialized with a certain number, such as 0 or 1, the comment should clearly specify the relevance of the initialization.

- Make sure that you spell check comments and that proper grammar and punctuation is used.
- Make sure that the comments do not use politically incorrect words that can be considered as biased.
- Make sure you use language-specific styles for commenting the code.

Microsoft Visual C#® development tool allows you to comment inline within an identifier, such as a namespace, class, or function, using the C style syntax as shown in the following code.

```
//This is a single line comment
/*This is a block comment
  spanning multiple lines */
```

The language also allows you to add documentation comments in XML notation as shown in the following code.

```
/// <summary>
/// This class is used for reading the configuration...
/// </summary>
```

These types of comments usually precede a user-defined type, such as a class or interface. You can compile a documentation file for these comments using a documentation generator. The following are some of the main commenting guidelines for C#:

- Avoid block comments for descriptions; you should use documentation style comments for description of the identifiers.
- Single line comments are preferred when commenting a section of code.
- The length of comment should not significantly exceed the length of code it explains.

For Microsoft Visual Basic® .NET development system, the recommended approach is to use the (‘) symbol instead of using the REM keyword because it requires less space and memory.

For more information about guidelines for documentation comments in C#, see “B. Documentation comments” on MSDN at:
*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/csspec/html/vclrfCSharpSpec_B.asp.*

For more information about commenting guidelines for Visual Basic .NET, see “Comments in Code” on MSDN at:
*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vbcn7/html/vaconCommentsInCode.asp.*

## Step 5: Ensure That Performance and Scalability Guidelines Are Followed

The code review ensures that the implementation adheres to best design practices and implementation tips that help improve performance and scalability. Some of the important things you should look for in any source code are the following:

- **Code ensures efficient resource management.** From a performance perspective, the first and foremost step in the code review is to validate that the code handles its resources efficiently. The performance of the application block significantly depends on the way code engages and releases resources. If the code holds on to a scarce resource for longer than required, it may result in behaviors such as increased contention higher response times or increased resource utilization. The following are some of the guidelines for efficient resource management:
  1. The resources should be released as early as possible. This is especially true in cases of shared resources, such as database connections or file handles. As much as possible, the code should be using the Dispose (or Close) pattern on disposable resources. For example, if the application block is accessing an unmanaged resource, such as a network connection or a database connection across client calls, it provides clients a suitable mechanism for deterministic cleanup by implementing the Dispose pattern. The following code sample shows the Dispose pattern.

```
public sealed class MyClass: IDisposable
{
    // Variable to track if Dispose has been called
    private bool disposed = false;
    // Implement the IDisposable.Dispose() method
    public void Dispose(){
    // Check if Dispose has already been called
    if (!disposed)
    {
// Call the overridden Dispose method that contains common cleanup
// code
    // Pass true to indicate that it is called from Dispose
    Dispose(true);
// Prevent subsequent finalization of this object. This is not
// needed because managed and unmanaged resources have been
// explicitly released
    GC.SuppressFinalize(this);
    }
    }

    // Implement a finalizer by using destructor style syntax
    ~MyClass() {
// Call the overridden Dispose method that contains common cleanup
// code
    // Pass false to indicate the it is not called from Dispose
```

```
    Dispose(false);
    }

    // Implement the override Dispose method that will contain common
    // cleanup functionality
    protected virtual void Dispose(bool disposing){
    if(disposing){
    // Dispose time code
    . . .
    }
    // Finalize time code
    . . .
    }
    …
}
```

2. The code should use try-finally blocks in Visual Basic .NET or statements in C# to ensure resources are released, even in the case of an exception. The following code sample shows the usage of a **using** statement.

```
using( StreamReader myFile = new StreamReader("C:\\ReadMe.Txt")){
    string contents = myFile.ReadToEnd();
    //... use the contents of the file

} // dispose is called and the StreamReader's resources released
```

3. The classes accessing the unmanaged object should be separated out as wrapper classes for the object without referencing to other unmanaged objects. In this way, only the wrapper object is marked for finalization without other leaf objects being promoted into the finalization queue for an expensive finalization process.
4. Analyze the class and structure design and identify those that contain many references to other objects. These result in complex object graphs at run time, which can be expensive for performance. Identify opportunities to simplify these structures.
5. If you do not require a class level member variable after the call, check before making a long-running I/O call. If this is the case, set it to **null** before making the call. This enables those objects to be garbage collected while the I/O call is executing.
6. Identify the instances where your code caches objects to see if there is an opportunity to use weak references. Weak references are suitable for medium-sized to large-sized objects stored in a collection. They are not appropriate for very small objects. By using weak references, cached objects can be resurrected easily, if needed, or they can be released by garbage collection when there is memory pressure. Using weak references is just one way of implementing

caching policy. For more information about caching, see the "Caching" section of "Chapter 3 — Design Guidelines for Application Performance" in *Improving .NET Application Performance and Scalability* on MSDN at: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/scalenetchapt03.asp*.

7. Ensure that the code does not call GC.Collect explicitly because the garbage collector is self-tuning; therefore, programmatically forcing garbage collection is more likely to hinder performance than to help it.
8. Investigate where the buffers (required mostly for I/O calls) are allocated in your code. You can reduce heap fragmentation if they are allocated when the application block is initialized at the application startup. You should also consider reusing and pooling the buffers for efficiency.
9. Consider using a **for** loop instead of **foreach** to increase performance for iterating through .NET Framework collections that can be indexed with an integer.

- **Code performs string operations efficiently.** String operations in managed code look deceptively simple, but they can be a cause of memory overhead if they are not handled efficiently. The reason behind this is that the strings are immutable. If you concatenate (or perform some other operation that changes the string itself) a string to another string, the old object has to be discarded with the new value being stored in a new object. To minimize the affect on these side effect allocations, especially in loops, you should ensure that the following guidelines are followed:

1. **StringBuilder** is used when the number of concatenations is unknown, such as in loops or multiple iterations. Concatenation using the **+** operator may result in a lot of side effect allocations that are discarded after each new iteration. The usage of **StringBuilder** is shown in the following code.

```
StringBuilder sb = new StringBuilder();
Array arrOfStrings = GetStrings();
for(int i=0; i<10; i++){
    sb.Append(arrOfStrings.GetValue(i));
}
```

2. If you know the number of appends and concatenate strings in a single statement or operation, give preference to the **+** operator.
3. Watch out for code that calls the **ToLower** method. Converting strings to lowercase and then comparing them involves temporary string allocations. This can be very expensive, especially when comparing strings inside a loop. You should give preference to using the **String.Compare** method. If you need to perform case-insensitive string comparisons, you can use the overloaded **String.Compare** method shown in the following code.

```
// The last argument is to signify a case-sensitive or case-//insensitive
comparison
String.Compare (string strA, string strB, bool ignoreCase);
```

The CMAB performs a number of string operations because it handles a lot of XML manipulation. Even after a thorough review of the code, it is important to ensure that there are no side effect allocations taking place. You can mark the various use cases for white box testing in such a way that they can be profiled using the CLR profiler.

For more information about CLR profiler, see "How To: Use CLR Profiler" in *Improving .NET Application Performance and Scalability* on MSDN at: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/scalenethowto13.asp.*

- **Code uses efficient data structures.** The data structures used in the code minimize the overhead of enumeration and provide appropriate functionality for searching and sorting. For example, if you are enumerating through the value types, you should implement the enumerator pattern for a custom collection to avoid the overhead of boxing or unboxing.

For more information, see the following resources:

- For a question-based review of the managed code to ensure adherence to best practices, see "Chapter 13 — Code Review: .NET Application Performance" in *Improving .NET Application Performance and Scalability* on MSDN at: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/ScaleNetChapt13.asp.*
- For a checklist-based review of the managed code, see "Checklist: Managed Code Performance" in *Improving .NET Application Performance and Scalability* on MSDN at: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/ScaleNetCheck06.asp.*
- For more information about the enumerator pattern, see "Chapter 5 — Improving Managed Code Performance" in *Improving .NET Application Performance and Scalability* on MSDN at: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/scalenetchapt05.asp.*

## Step 6: Ensure That Guidelines for Writing Secure Code Are Followed

Security code reviews help you identify areas in your code that do not follow the best practices for writing secure code. Identifying this code is important because it directly weakens the security in one way or other. The security vulnerabilities that get introduced because of they do not adhere to guidelines for writing secure code are difficult to catch during black box testing and may surface later as a threat in a production scenario.

The following are some of the vulnerabilities you can look for during security code reviews:

1. Check for hard coded information. There should not be any hard coded information, such as UserName, Password, or ConnectionString, in any portion of the source code. This information should be kept in a configuration file.
2. Code reviews should ensure that the code uses code access security appropriately, as discussed in the following examples:
   - During the code review, make sure that the privileged code identified during the design and build phase requests appropriate permissions while accessing secure resources or while performing privileged operations.
   - Review that the permission requests are appropriate and minimum. For example, if your code needs to read a specific key in the registry, it should request read permission to that specific key in the registry.
   - Check whether the assembly is strong named; make sure that it does not use the AllowPartiallyTrustedCallersAttribute (APTCA). In cases where it is must use APTCA, make sure that the appropriate code access security demands are used.
3. Check for a cross-site scripting attack. Never place too much trust on the data entered by the user; always validate it. Use regular expressions to validate data in ASP.NET applications.
4. Check for buffer overflows. Validate length and data type of parameters and calls to the unmanaged code (including Win32 DLLs and COM objects) through COM interop or P/Invoke layers.
5. Disable detail error messages and tracing. Check whether the detailed error messages are disabled from popping up on the client computer and tracing is disabled for the ASP.NET applications.
6. View state protection. Check whether the view state is enabled only when required. Also check that view state protection is enabled as applicable.

Security code reviews can also help you identify test scenarios for white box testing. For example, if your code handles input data, you can make sure during security code review that the input data is validated. Security code review can also help to identify white box testing cases by reviewing the logic and validations that have been done.

For more information about security code reviews, see "Chapter 21 — Code Review" in *Improving Web Application Security: Threats and Countermeasures* on MSDN at: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnetsec/html/THCMCh21.asp.*

## Step 7: Ensure That Globalization-related Guidelines Are Followed

Globalization is the process of ensuring that a piece of software supports localized versions of the user interface, such as messages shown to the user (such as exceptions, warnings, and information) and that it can operate on regional data (such as different date notations and currency formatting).

Before starting the code review, you should note all cultures the application block will be supporting. The common issues related to globalization features revolve mostly around a set of features. This includes features such as any type of string operations, including comparing, sorting, or indexing, formatting date and time in specific cultures, using calendars in specific cultures, or comparing and sorting data in specific cultures.

The following are some of the best practices you should review for the application blocks:

- Preferably, the application should use Unicode internally. It is preferable for the application block to use Unicode encoding, though other techniques, such as DBCS (Double-Byte Character System), are also available. If you plan to use a database, you should use Unicode-compatible data types for database design. For example, the CMAB stores configuration information in a SQL Server database. This information can be from different character sets; therefore, you should make sure that the database design uses data types such as nchar, nvarchar, and ntext.
- Ensure that the culture-aware classes provided by the **System.Globalization** namespace are used to manipulate and format data. The various rules for languages and countries, such as number formats, currency symbols, and sort orders, are aggregated into a number of standard cultures. In the .NET Framework, culture is represented programmatically using the **System.Globalization.CultureInfo** class.

  The .NET Framework uses a combination of two cultures to handle different aspects of localization, namely Current culture and Current user interface culture. Current culture determines how various data types are formatted, such as numbers and dates. Current user interface culture determines which localized resource file the resource manager loads. The code may set the culture in the configuration file as shown in the following code.

```
<configuration>
    <system.web>
     <globalization culture="en-US" uiCulture="de-DE"/>
    </system.web>
</configuration>
```

The code may also set the culture programmatically as shown in the following code.

```
using System.Globalization;
using System.Threading;

Thread.CurrentThread.CurrentCulture =
    CultureInfo.CreateSpecificCulture ("de");

// Set the user interface culture to the browser's accept language
Thread.CurrentThread.CurrentUICulture =
    Thread.CurrentThread.CurrentCulture;
```

- For sorting, check if the application block uses the **SortKey** class and the **CompareInfo** class.

```
// Creates a SortKey using the en-US culture.
CompareInfo myComp_enUS = new CultureInfo("en-US",false).CompareInfo;
SortKey mySK1 = myComp_enUS.GetSortKey( "llama" );
```

- For string comparisons, the **CompareInfo** class should be used.

```
// Defines the strings to compare.
String myStr1 = "This is one string";
String myStr2 = "This is another string";

// Creates a CompareInfo that uses the InvariantCulture.
CompareInfo myComp = CultureInfo.InvariantCulture.CompareInfo;
ConsoleWriteLine(" The result of case insensitive comparison is: " +
myComp.Compare( myStr1, myStr2, CompareOptions.IgnoreCase));
```

- Formatting should be done using the appropriate classes, such as the **DateTimeFormatInfo** class for date and time formatting and the **NumberFormatInfo** class for numeric formatting. The following code shows an example of using **DateTimeFormatInfo**.

```
DateTimeFormatInfo dfmt = Thread.CurrentThread.CurrentCulture.DateTimeFormat;
dfmt.LongDatePattern = @"\[yyyy-MMMM-dd\]";
string longDateString = DateTime.Now.ToLongDateString();
```

- If you need to convert from a string to a **DateTime** data type, you should use the **DateTime.Parse** method as shown in the following code.

```
DateTime dt = DateTime.Parse(aFrenchDateString, new CultureInfo("de"));
```

- If your application block is to be integrated with a Web application that can be accessed from different time zones, you should use the universal time feature provided by the .NET Framework. Use the universal time feature for internal storage of the configuration values that can be converted into local time zone

values upon retrieval. Use the **DateTime.ToUniversalTime** and **DateTime.ToLocalTime** methods for performing these conversions.

```
DateTime localDateTime = System.DateTime.Now;
System.DateTime univDateTime = localDateTime.ToUniversalTime();
```

- Check if the application block can read and write data to and from a variety of encodings by using the encoding classes in the **System.Text** namespace. Ensure that the code does not assume ASCII data and that it assumes international characters will be supplied anywhere a user can enter text. For example, in the CMAB, international characters should be accepted in server names, directories, file names, and so on. You can use the static **GetEncoding** method of the **System.Text.Encoding** class to retrieve an **Encoding** object for a specific Code-Page, and then use its **GetBytes** method to convert Unicode strings to a Code-Page specific byte array.

```
Encoding iso = Encoding.GetEncoding("iso8859-1");
Encoding unicode = Encoding.UTF8;
byte[] unicodeBytes = unicode.GetBytes(src);
iso.GetString(unicodeBytes);
```

- Whenever possible, check if strings are handled as entire strings instead of as a series of individual characters. This is especially important when sorting or searching for substrings. This will prevent problems associated with parsing combined characters.
- The text should be displayed using the classes provided by the **System.Drawing** namespace.
- For consistency across operating systems, make sure that the code does not allow user settings to override **CultureInfo**. The code should use the **CultureInfo** constructor that accepts a **useUserOverride** parameter and set it to **false**. This is most useful when using application blocks in an application that is not Web based. For Web-based applications, the effective settings are that of the ASP.NET process.
- If a security decision is based on the result of a string comparison or case change operation, check if the code is performing a culture-insensitive operation by explicitly specifying the **CultureInfo.InvariantCulture** property. This practice ensures that the result is not affected by the value of **CultureInfo.CurrentCulture**.

For more information, see the following resources:

- For more information about satellite assemblies, see "Creating Satellite Assemblies" in the *.NET Framework Developer's Guide* on MSDN at: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconcreatingsatelliteassemblies.asp.*

- For more information about best practices for globalization, see "Best Practices for Developing World-Ready Applications" in the *.NET Framework Developer's Guide* on MSDN at: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconbestpracticesforglobalapplicationdesign.asp.*
- For more information about best practices for globalization and localization, see "Best Practices for Globalization and Localization" on MSDN at: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsent7/html/vxconBestGlobalizationPractices.asp.*
- For more information about issues related to globalization and localization, see "Globalization and Localization Issues" on MSDN at: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsent7/html/vxconGlobalizationLocalization.asp.*

## Step 8: Validate Exception Handling in the Code

Robust exception handling is a very important feature that significantly affects the user experience with the application. It helps administrators managing an application and it helps developers to analyze problems that surface in the actual production environment. Exception handling is an important design feature. During the design phase, you capture all the possible exceptions that need to be handled for a particular use case. The code review makes sure that the code appropriately handles the exceptions. The following are some guidelines for exception handling that should be reviewed in the code:

1. Exception handling should not be used to control the application logic. It should be used only in exceptional cases because throwing exceptions is expensive. For example, if the CMAB is checking for configuration information from the database for a particular key and does not find any value, this is an expected condition and is not a fit case for throwing exception as shown in the following code.

```
// Bad scenario to throw exception
static void ConfigExists( string KeyForConfig) {
    //... search for Key
    if ( dr.Read(KeyForConfig) ==0 ) // no record found
    {
    throw( new Exception("Config Not found"));
    }
}
```

   These conditions should be handled by returning a simple Boolean value. If the Boolean value is **true**, the configuration information exists in the database.

```
static void ConfigExists( string KeyForConfig) {
    //... search for key
    if ( dr.Read(KeyForConfig) ==0 ) // no record found
```

```
{
    return false;
    }
    . . .
}
```

Returning error information using an enumerated type instead of throwing an exception is another commonly used programming technique in performance-critical code paths and methods.

2. All exception conditions identified for a particular use case should be handled. You should make sure the exceptions are not caught only to be rethrown to the user because throwing exceptions is expensive. It should be done only if you are adding some value or performing some action on the exception object, such as logging into an event sink.

```
try {
    //try to load an XML file
LoadXmlFile(xmlApplicationDocument, _applicationDocumentPath);
    }
    catch (XmlException)
    {
//Log the exception before re throwing it
LogExceptionInSink(XmlException)
//if the loading of the document fails throw a configuration
//exception stating that configuration document is invalid
throw new
ConfigurationException(Resource.ResourceManager["RES_ExceptionInvalidConfigurationDocument"]);
    }
finally
{
    // Code that gets run always, whether or not
    // an exception was thrown. This is usually
    // clean up code that should be executed
    // regardless of whether an exception has
    // been thrown.
}
```

3. Code does not swallow exceptions or inefficient code that catches, wraps, and rethrows exceptions for no valid reason. If you do not need to act on a particular exception, you should allow the exception to bubble up from the application block code to the client application. The user integrating the application block can capture the exception and decide on how to act on the exception.
4. Make sure that the **finally** blocks are used to ensure that the resources are cleaned up. Make sure of this even in the case of exception as shown in the following code.

```
SqlConnection conn = new SqlConnection("...");
try {
    conn.Open();
```

```
    // Some operation that might cause an exception

    // Calling Close as early as possible
    conn.Close();
    // ... other potentially long operations

}
finally {
    if (conn.State==ConnectionState.Open)
conn.Close();  // ensure that the connection is closed
}
```

You can also use the using statement in C# to clean up resources that require a call to **Dispose ()** method for deterministic clean up of resources.

5. Preferably, the logging mechanism should be configurable in such a way that the client can decide the event sinks for logging of particular type of errors. For example, the default configuration of the CMAB logs high severity errors in the event log and logs all the warnings and informational messages in Windows Event Trace.

   The configuration should also provide an option of switching on and off the logging either partially or completely. This is because logging might be an undesirable overhead in the production environment.

6. Check whether the exceptions caught by any part of the program or subprogram are propagated through the system and necessary actions are taken, depending on the exception type. Functions should avoid returning error codes to the calling functions; instead, an exception propagation mechanism should be used.

7. Make sure that the code uses custom exceptions. The .NET platform gives provision for creating custom exceptions. Custom exceptions generally help in giving more detailed information on the exception that occurred or in highlighting any business logic discrepancies. For example, if the CMAB fails to read the configuration file during the application initialization process, it throws an exception of type **ConfigurationException** or a class derived from **ConfigurationException**.

8. Check whether the unhandled exceptions are properly managed. For example, in ASP.NET applications, the application's Web.config file can be used to show a custom error page for the clients instead of showing the exact stack trace of the error to the clients by the following settings.

   ```
   <customErrors defaultredirect="http://hostname/error.aspx" mode="on">
   </customErrors>
   ```

9. Exception management should also ensure that any informational messages are also logged. These events are useful in monitoring the health of the application block. For example, the CMAB can report any requests that hold locks for writing or updating information for more than 3 milliseconds.

For more information about exception management architecture, see *Exception Management Architecture Guide* on MSDN at: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/exceptdotnet.asp.*

### Step 9: Identify the Scenarios for More Testing

During the code review process, testers get to review the code extensively through various perspectives, such as performance, security, and globalization. During this review, testers make sure the code is in accordance with the guidelines; however, this is not a substitute for actually testing the scenario. The goal of the review is to minimize surprises in the later stages.

During the code review, testers should always try to identify the usage scenarios for additional testing. These are the scenarios that require extensive testing rather than a simple code review, such as in the following examples:

- You have done a code review of an internal function of a component that takes numeric input. You would want to make sure that during white box testing, the function is thoroughly tested for all types of inputs such as 0, -ve numbers, numbers out of the accepted range, and decimals.
- During the code review, you come across a function that makes remote calls. You may want to profile the time taken by the function when the block is under a concurrent load of users. If the remote call is an I/O call for retrieving information in buffers, you may want to further load test it with different payloads to avoid the possible problems of buffer overflow.
- A particular function does the sorting of string data before returning the values in an array. This function needs additional testing for various cultures that have been specified in the requirements.

The scenarios identified during code review are tested more in the black box testing and white box testing stages, such as performance testing and globalization testing.

## Tools

The FxCop tool can be used to check for compliance of code with various types of guidelines. The tool analyzes binary assemblies (not source code) to ensure that they conform to the .NET Framework Design Guidelines, available as rules in FxCop.

The tool comes with a predefined set of rules, but you can customize and extend them.

For more information, see the following resources:

- To download the FxCop tool, see "About Developing Reusable Libraries" on GotDotNet at: *http://www.gotdotnet.com/team/libraries/default.aspx.*

- For general information about FxCop, see the "FxCop Team Page" on GotDotNet at: *http://www.gotdotnet.com/team/fxcop/.*
- To get help and support for the FxCop, see the GotDotNet message boards for discussions about the FxCop tool at: *http://www.gotdotnet.com/community/messageboard/MessageBoard.aspx?ID=234.*

## Summary

This chapter presented a process for reviewing the implementation of an application block. A thorough code review ensures that you can significantly cut down on the time spent during white box testing. The erring code that has been caught during the code review of the modules can be fixed very easily but if it slips through and is caught during the white box testing, it may result in cascading changes across the modules.

# 6

# Black Box and White Box Testing for Application Blocks

## Objectives

- Learn about black box testing an application block.
- Learn about white box testing an application block.

## Overview

After you complete the design and code review of the application block, you need to test the application block to make sure it meets the functional requirements and successfully implements the functionality for the usage scenarios it was designed and implemented for.

The testing effort can be divided into two categories that complement each other:

- **Black box testing.** This approach tests all possible combinations of end-user actions. Black box testing assumes no knowledge of code and is intended to simulate the end-user experience. You can use sample applications to integrate and test the application block for black box testing. You can begin planning for black box testing immediately after the requirements and the functional specifications are available.
- **White box testing.** (This is also known as glass box, clear box, and open box testing.) In white box testing, you create test cases by looking at the code to detect any potential failure scenarios. You determine the suitable input data for testing various APIs and the special code paths that need to be tested by analyzing the source code for the application block. Therefore, the test plans need to be updated before starting white box testing and only after a stable build of the code is available.

A failure of a white box test may result in a change that requires all black box testing to be repeated and white box testing paths to be reviewed and possibly changed.

The goals of testing can be summarized as follows:

- Verify that the application block is able to meet all requirements in accordance with the functional specifications document.
- Make sure that the application block has consistent and expected output for all usage scenarios for both valid and invalid inputs. For example, make sure the error messages are meaningful and help the user in diagnosing the actual problem.

You may need to develop one or more of the following to test the functionality of the application blocks:

- Test harnesses, such as NUnit test cases, to test the API of the application block for various inputs
- Prototype Windows Forms and Web Forms applications that integrate the application blocks and are deployed in simulated target deployments
- Automated scripts that test the API of the application blocks for various inputs

This chapter examines the process of black box testing and white box testing. It includes code examples and sample test cases to demonstrate the approach for black box testing and white box testing application blocks. For the purpose of the examples illustrated in this chapter, it is assumed that functionality testing is being done for the Management Application Block (CMAB). The CMAB has already been through design and code review. The requirements for the CMAB are the following:

- It provides the functionality to read and store configuration information transparently in a persistent storage medium. The storage mediums are SQL Server, the registry, and an XML file.
- It provides a configurable option to store the information in encrypted form and plain text using XML notation.
- It can be used with desktop applications and Web applications that are deployed in a Web farm.
- It caches configuration information in memory to reduce cross-process communication, such as reading from any persistent medium. This reduces the response time of the request for any configuration information. The expiration and scavenging mechanism for the data that is cached in memory is similar to the cron algorithm in UNIX.
- It can store and return data from various locales and cultures without any loss of data integrity.

# Black Box Testing

Black box testing assumes the code to be a black box that responds to input stimuli. The testing focuses on the output to various types of stimuli in the targeted deployment environments. It focuses on validation tests, boundary conditions, destructive testing, reproducibility tests, performance tests, globalization, and security-related testing.

Risk analysis should be done to estimate the amount and the level of testing that needs to be done. Risk analysis gives the necessary criteria about when to stop the testing process. Risk analysis prioritizes the test cases. It takes into account the impact of the errors and the probability of occurrence of the errors. By concentrating on the test cases that can lead to high impact and high probability errors, the testing effort can be reduced and the application block can be ensured to be good enough to be used by various applications.

Preferably, black box testing should be conducted in a test environment close to the target environment. There can be one or more deployment scenarios for the application block that is being tested. The requirements and the behavior of the application block can vary with the deployment scenario; therefore, testing the application block in a simulated environment that closely resembles the deployment environment ensures that it is tested to satisfy all requirements of the targeted real-life conditions. There will be no surprises in the production environment. The test cases being executed ensure robustness of the application block for the targeted deployment scenarios.

For example, the CMAB can be deployed on the desktop with Windows Forms applications or in a Web farm when integrated with Web applications. The CMAB requirements, such as performance objectives, vary from the desktop environment to the Web environment. The test cases and the test environment have to vary according to the target environments. Other application blocks may have more restricted and specialized target environments. An example of an application block that requires a specialized test environment is an application block that is deployed on mobile devices and is used for synchronizing data with a central server.

As mentioned earlier, you will need to develop custom test harnesses for functionality testing purpose.

## Input

The following input is required for black box testing:

- Requirements
- Functional specifications
- High-level design documents
- Application block source code

The black box testing process for an application block is shown in Figure 6.1.

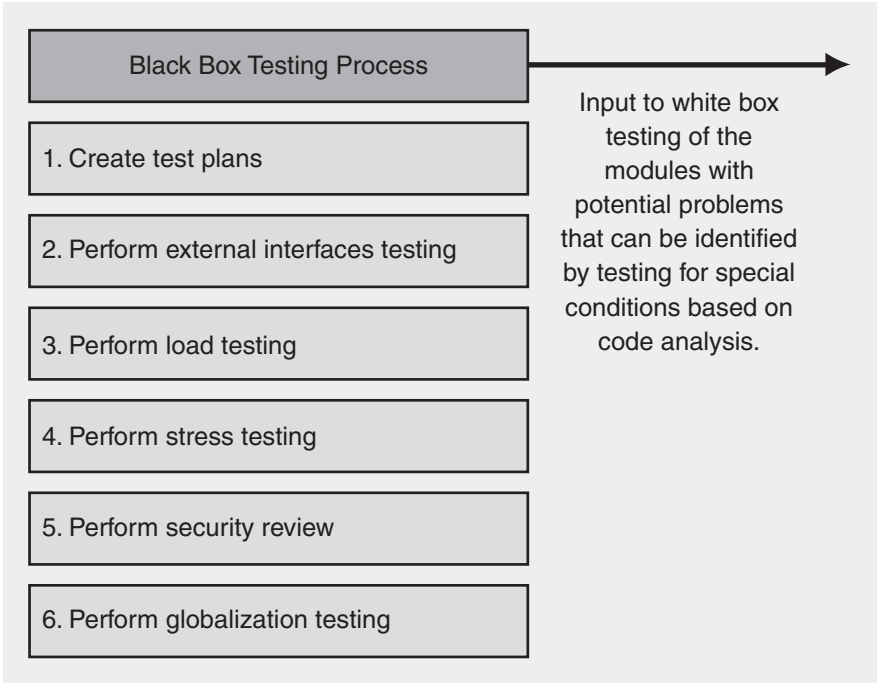


**Figure 6.1**
*Black box testing process*

## Black Box Testing Steps

Black box testing involves testing external interfaces to ensure that the code meets functional and nonfunctional requirements. The various steps involved in black box testing are the following:

1. **Create test plans.** Create prioritized test plans for black box testing.
2. **Test the external interfaces.** Test the external interfaces for various type of inputs using automated test suites, such as NUnit suites and custom prototype applications.
3. **Perform load testing.** Load test the application block to analyze the behavior at various load levels. This ensures that it meets all performance objectives that are stated as requirements.
4. **Perform stress testing.** Stress test the application block to analyze various bottlenecks and to identify any issues visible only under extreme load conditions, such as race conditions and contentions.

5. **Perform security testing.** Test for possible threats in deployment scenarios. Deploy the application block in a simulated target environment and try to hack the application by exploiting any possible weakness of the application block.
6. **Perform globalization testing.** Execute test cases to ensure that the application block can be integrated with applications targeted toward locales other than the default locale used for development.

The next sections describe each of these steps.

## Step 1: Create Test Plans

The first step in the process of black box testing is to create prioritized test plans. You can prepare the test cases for black box testing even before you implement the application block. The test cases are based on the requirements and the functional specification documents.

The requirements and functional specification documents help you extract various usage scenarios and the expected output in each scenario.

The detailed test plan document includes test cases for the following:

- Testing the external interfaces with various types of input
- Load testing and stress testing
- Security testing
- Globalization testing

For more information about creating test cases, see Chapter 3, "Testing Process for Application Blocks."

## Step 2: Test the External Interfaces

You need to test the external interfaces of the application block using the following strategies:

- **Ensure that the application block exposes interfaces that address all functional specifications and requirements.** To perform this validation testing, do the following:
  1. Prepare a checklist of all requirements and features that are expected from the application block.
  2. Create test harnesses, such as NUnit, and small "hello world"' applications to use all exposed APIs of the test application block.
  3. Run the test harnesses.

  Using NUnit, you can validate that the intended feature is working if the input is given on the expected lines.

The sample applications can indicate whether the application block can be integrated and deployed in the target environment. The sample applications are used to test for the possible user actions for the usage scenarios; these include both the expected process flows and the random inputs. For example, a Web application deployed in a Web farm that integrates the CMAB can be used to test reading and writing information from a persistent database, such as the registry, SQL, or an XML file. You need to test the functionality by using various configuration options in the configuration file.

- **Testing for various types of inputs.** After ensuring that the application block exposes the interfaces that address all of the functional specifications, you need to test the robustness of these interfaces. You need to test for the following input types:
  - Randomly generated input within a specified range
  - Boundary cases for the specified range of input
  - The number zero testing if the input is numeric
  - The null input
  - Invalid input or input that is out of the expected range

This testing ensures that the application block provides expected output for data within the specified range and gracefully handles all invalid data. Meaningful error messages should be displayed for invalid input. Boundary testing ensures that the highest and lowest permitted inputs produce expected output.

You can use NUnit for this type of input testing. Separate sets of NUnit tests can be generated for each range of input types. Executing these NUnit tests on each new build of the application block ensures that the API is able to successfully process the given input.

For example, consider the following API that is a part of an application block and takes in an integer argument.

```
public class SampleClass{
public string SampleAPI(int testArg){
// API specific logic where a check is made whether testArg has a
//value greater than 0 and is less than 65536. If the preceding
//condition is not satisfied then an exception of user defined type //
DataException is thrown.
}
}
```

The test harness code using NUnit framework for testing the various inputs to the above API will look like the following.

```
//NUnit tests provide various types of inputs for testing of the SampleClass
using NUnit.Framework;
namespace TestHarnessSample{
    [TestFixture]
    public class SampleTest{
    [Test]
[ExpectedException(typeof(DataException))]
    public void ZeroInputTest(){
    SampleClass testSample = new testSample ();
    //test for input as 0
    int result = testSample.SampleAPI (0);
    }
    [Test]
[ExpectedException(typeof(DataException))]
    public void NegativeInputTest(){
    SampleClass testSample = new testSample ();
    //test for input which is less than the expected range
    int result = testSample.SampleAPI (-1);
    }
[Test]
[ExpectedException(typeof(DataException))]
    public void NullInputTest(){
    SampleClass testSample = new testSample ();
    //test for null input
    int result = testSample.SampleAPI (null);
    }
[Test]
[ExpectedException(typeof(DataException))]
    public void LargeNumberTest()    {
    SampleClass testSample = new testSample ();
    //test for input which is greater than the expected range
    int result = testSample.SampleAPI (65537);
    }
[Test]
    public void ValidInputTest()    {
    SampleClass testSample = new testSample ();
    //test for input which is within the expected range
    int result = testSample.SampleAPI (65);
    Assert.AreEqual(3000, result);
    }
    }
}
```

Using the preceding test harness code, the various combinations of inputs can be tested for the **SampleClass**. The expected behavior for all test methods, except the **ValidInputTest** method, is that a user-defined exception of type **DataException** should be thrown. If the **DataException** is not thrown, the API is missing the intended input validation and the test is a failure. For the **ValidInputTest** method, a result of 3,000 is expected and if the result is not obtained, the particular test is a failure.

For a more specific example, consider the CMAB example. The test harness code that feeds various combinations of inputs to the **ConfigurationManager** class is shown in the following code.

```
using NUnit.Framework;
namespace CMABTestSample{
    [TestFixture]
    public class CMABTest{
        private string sectionName;
        [Test]
         [ExpectedException(typeof(ArgumentNullException))]
        public void Test1(){
            //test by passing an empty string as section name for
            //reading configuration data
            sectionName = "";
 Hashtable sampleTable =  (Hashtable)ConfigurationManager.Read(sectionName);
        }
        [Test]
         [ExpectedException(typeof(ConfigurationException))]
        public void Test2(){
            sectionName = " ";
            //test by passing a string with a space character as
            //section name for reading configuration data
 Hashtable sampleTable =
(Hashtable)ConfigurationManager.Read(sectionName);
        }
        [Test]
         [ExpectedException(typeof(ArgumentNullException))]
        public void Test3(){
            //test by passing a string with a space character as
            //section name for reading configuration data
             Hashtable sampleTable = (Hashtable)ConfigurationManager.Read(null);
        }
        [Test]
        public void Test4(){
            sectionName = "TestConfigSection";
            //test by passing a valid section name for reading
            //configuration data
 Hashtable sampleTable =     (Hashtable)ConfigurationManager.Read(sectionName);
             Assert.AreEqual(sampleTable["Item1"].Value, "XXXXX");
        }
    }
}
```

The preceding test harness code gives various combinations of inputs to the CMAB **ConfigurationManager** class for reading of data. The first three test methods force exceptions and tests that exceptions are thrown as expected. The last method, Test4(), passes a valid input parameter and expects a valid response of configuration data. The exceptions that should be generated in the respective test methods are mentioned in the **ExpectedException** attribute on top of every test method. If the indicated exception is not thrown by the **ConfigurationManager**, that particular test

is considered to be a failure. The CMAB is expected to throw **ArgumentNullException** if an empty string or null value is passed to the **ConfigurationManager**. If any other invalid input is passed, the CMAB is expected to throw **ConfigurationException**. The test method Test4() passes a valid input to the **ConfigurationManager.Read** method. The output is compared with the expected result by using the **Assert.AreEqual()** method of the NUnit framework. If the expected result does not match the output returned by the Configuration Manager, the test is a failure.

## Step 3: Perform Load Testing

Use load testing to analyze the application block behavior under normal and peak load conditions. Load testing allows you to verify that the application block can meet the desired performance objectives and does not overshoot the allocated budget for resource utilization such as memory, processor, and network I/O. The requirements document usually lists the resource utilization budget for the application block and the workload it should be able to support.

For example, the CMAB had the following performance objectives on a Web server (please note that these objectives are totally fictitious and are only for the purpose of illustration):

- The CPU overhead should not be more than 7 – 10 percent.
- The application block should be able to support a minimum of 200 concurrent users for reading data from SQL Server.
- The application block should be able to support a minimum of 150 concurrent users for writing data to SQL Server.
- The response time for a client (the client is firing requests from a 100 Mbps VLAN in the test lab) is not more than 2 seconds for the given concurrent load.

You can measure metrics related to response times, throughput rates, and so on, for the load test. In addition, you can measure other metrics that help you identify any potential bottlenecks.

To load test an application block, you need to develop a sample application that is an accurate prototype of applications that will be used in the target environment. In the case of the CMAB, and because one of the deployment scenarios is the Web environment, a simple Web application can be developed that uses the application block for reading and writing configuration information. Preferably, this application block should be tested in clustered and nonclustered environments because deploying in a Web farm is one of the deployment scenarios.

For a detailed process on load testing application blocks, see Chapter 8, “Performance Testing for Application Blocks.”

Table 6.1 shows a sample test case for load testing the CMAB.

**Table 6.1: Sample Test Case Document for Load Testing the CMAB When Reading Data from a SQL Store**

| | |
|---|---|
| Scenario 1.1 | Reading configuration data from a SQL store with data caching and data protection options enabled. |
| Priority | High |
| Execution details | Create a sample Web application that integrates the CMAB.Add counters to test the performance of the CMAB.<br>Configure ACT to set the following attributes for the test:<br>● **Number of users**: 200 concurrent users<br>● **Test duration**: 20 minutes<br>● **Think time**: 0Run ACT tool for the specific test duration. |
| Tools required | Sample application integrating the CMAB. |
| Expected results | **Throughput>** 150 requests per second<br>**Processor\%Processor Time<** 75 %<br>**Request Execution Time<=** 2 seconds (on 100 megabits per second [Mbps] LAN) |

## Step 4: Perform Stress Testing

Use stress testing to evaluate the application block's behavior when it is pushed beyond the normal or peak load conditions. The expectation from the system beyond load conditions is to either return expected output or return meaningful error messages to the user without corrupting the integrity of any data. The goal of stress testing is to discover bugs that surface only under high load conditions, such as synchronization issues, race conditions, and memory leaks.

The data that is collected in stress testing is based on the input from load testing and the code review. The code review identifies the potential areas in code that may lead to the preceding issues. The metrics collected in load testing also provides input for identifying the scenarios that need to be stress tested. For example, if during load testing, you observe that the application starts to show increased response times for increased load conditions when writing to SQL Server, you should check for any potential issues because of concurrency.

For a detailed process on stress testing application blocks, see Chapter 8, "Performance Testing for Application Blocks."

Table 6.2 shows a sample test case for stress testing the CMAB.

**Table 6.2: Sample Test Case Document for Stress Testing the CMAB When Reading Data from a SQL Store**

| | |
|---|---|
| Scenario 1.2 | Reading configuration data from a SQL store with data caching and data protection options enabled. |
| Priority | High |
| Execution details | Use the sample application created for load testing.<br>Add counters to identify potential bottlenecks.<br>Configure ACT to set the following attributes for the test:<br>% **Number of users**: 500 concurrent users<br>% **Test duration**: 60 minutes<br>% **Think time**: 0Run ACT tool for the specific test duration. |
| Tools required | Sample application integrating the CMAB. |
| Expected results | The ASP.NET worker process should not be recycled.<br>Response time should not exceed 7 seconds (on a 100 megabits per second [Mbps] LAN)<br>Server busy errors should not be more than 20 percent of the total response time because of contention-related issues. |

## Step 5: Perform Security Testing

Black box security testing the application block identifies security vulnerabilities within the application block by treating it as an independent unit. The testing is done at run time. The purpose is to forcefully break the interfaces of the application block, intercept sensitive data within the block, and so on. Sample test harnesses can be used to create a deployment scenario for the application block.

Depending on the functionality the application block provides, test cases can be identified. Examples of test cases and tests can be the following:

- If the application block accepts data from a user, make sure it validates the input data by creating test cases to pass different types of data, including unsafe data, through the application block's interfaces and confirming that the application block is able to stop it and handle it by providing appropriate error messages.
- If the application block accesses any secure resources, such as the registry or file system, identify test cases that can test for threats resulting from elevated privileges.
- If the application block handles secure data and uses cryptography, scenarios can be developed for simulating various types of attacks to access the data. This tests and ensures that the appropriate algorithms and methods are used to secure data.

### Step 6: Perform Globalization Testing

The goal of globalization testing is to detect potential problems in the application block that could inhibit its successful integration with an application that uses culture resources different than the default culture resources used for development. Globalization testing involves passing culture-specific input to a sample application integrating the application block. It makes sure that the code can handle all international support and supports any culture or locale settings without breaking functionality that would cause data loss.

To perform globalization testing, you must install multiple language groups and set the culture or locale to different cultures or locales, such as Japanese or German, from the local culture or locale. Executing test cases in both Japanese and German environments, and a combination of both, can cover most globalization issues.

For a detailed process on globalization testing application blocks, see Chapter 7, "Globalization Testing for Application Blocks."

## White Box Testing

White box testing assumes that the tester can take a look at the code for the application block and create test cases that look for any potential failure scenarios. During white box testing, you analyze the code of the application block and prepare test cases for testing the functionality to ensure that the class is behaving in accordance with the specifications and testing for robustness.

### Input

The following input is required for white box testing:

- Requirements
- Functional specifications
- High-level design documents
- Detailed design documents
- Application block source code

### White Box Testing Steps

The white box testing process for an application block is shown in Figure 6.2.

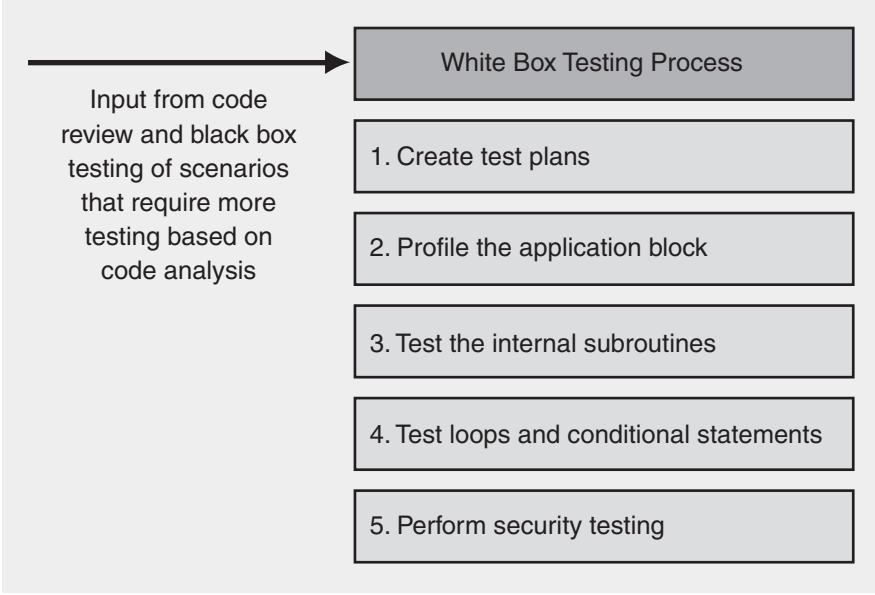


**Figure 6.2**
*White box testing process*

White box testing involves the following steps:

1. **Create test plans.** Identify all white box test scenarios and prioritize them.
2. **Profile the application block.** This step involves studying the code at run time to understand the resource utilization, time spent by various methods and operations, areas in code that are not accessed, and so on.
3. **Test the internal subroutines.** This step ensures that the subroutines or the nonpublic interfaces can handle all types of data appropriately.
4. **Test loops and conditional statements.** This step focuses on testing the loops and conditional statements for accuracy and efficiency for different data inputs.
5. **Perform security testing.** White box security testing helps you understand possible security loopholes by looking at the way the code handles security.

The next sections describe each of these steps.

## Step 1: Create Test Plans

The test plans for white box testing can be created only after a reasonably stable build of the application block is available. The creation of test plans involves extensive code review and input from design review and black box testing. The test plans for white box testing include the following:

- Profiling, including code coverage, resource utilization, and resource leaks
- Testing internal subroutines for integrity and consistency in data processing
- Loop testing; test simple, concatenated, nested, and unstructured loops
- Conditional statements, such as simple expressions, compound expressions, and expressions that evaluate to Boolean.

For more information about creating test cases, see Chapter 3, "Testing Process for Application Blocks."

## Step 2: Profile the Application Block

Profiling allows you to monitor the behavior of a particular code path at run time when the code is being executed. Profiling includes the following tests:

- **Code coverage.** Code coverage testing ensures that every line of code is executed at least once during testing. You must develop test cases in a way that ensures the entire execution tree is tested at least once. To ensure that each statement is executed once, test cases should be based on the control structure in the code and the sequence diagrams from the design documents. The control structures in the code consist of various conditions as follows:
  - Various conditional statements that branch into different code paths. For example, a Boolean variable that evaluates to "false" or "true" can execute different code paths. There can be other compound conditions with multiple conditions, Boolean operators, and bit-wise comparisons.
  - Various types of loops, such as simple loops, concatenated loops, and nested loops.

  There are various tools available for code coverage testing, but you still need to execute the test cases. The tools identify the code that has been executed during the testing. In this way, you can identify the redundant code that never gets executed. This code may be left over from a previous version of the functionality or may signify a partially implemented functionality or dead code that never gets called.

  Tables 6.3 and 6.4 list sample test cases for testing the code coverage of **ConfigurationManager** class of the CMAB.

**Table 6.3: The CMAB Test Case Document for Testing the Code Coverage for InitAllProvider Method and All Invoked Methods**

| | |
|---|---|
| Scenario 1.3 | Test the code coverage for the method **InitAllProviders()** in **ConfigurationManager** class. |
| Priority | High |
| Execution details | Create a sample application for reading configuration data from a data store through the CMAB.Run the application under the following conditions:<br>● With a default section present<br>● Without a default section<br>Trace the code coverage using an automated tool.<br>Report any code not being called in **InitAllProviders()**. |
| Tools required | Custom test harness integrating the application block for reading configuration data. |
| Expected results | The entire code for Init AllProviders() method and all the invoked methods Should be covered under the preceding conditions. |

**Table 6.4: The CMAB Test Case Document for Testing the Code Coverage for Read Method and All Invoked Methods**

| | |
|---|---|
| Scenario 1.4 | Test the code coverage for the method **Read** (sectionName) in the **ConfigurationManager** class. |
| Priority | High |
| Execution details | Create a sample application for reading configuration data from SQL database through the CMAB.<br>Run the application under the following conditions:<br>● Give a null section name or a section name of zero length to the **Read** method.<br>● Read a section whose name is not mentioned in the App.config or Web.config files.<br>● Read a configuration section that has cache enabled.<br>● Read a configuration section that has cache disabled.<br>● Read a configuration section successfully with the cache disabled, and then disconnect the database and read the section again.<br>● Read a configuration section with the section having no configuration data in the database.<br>● Read the configuration section that does not have provider information mentioned in the App.config or Web.config files.Trace the code coverage.Report any code left not being covered in the **Read** (sectionName) method. |
| Tools required | Custom test harness integrating the application block for reading of configuration data. |
| Expected results | The entire code for the **Read** (sectionName) method and the invoked methods should be covered under the preceding conditions. |

- **Memory allocation pattern.** You can profile the memory allocation pattern of the application block by using code profiling tools. You need to check for the following in the allocation pattern:
  - **The percentage of allocations in Gen 0, Gen 1, and Gen 2.** If the percentage of objects in Gen 2 is high, the resource cleanup in the application block is not efficient and there are memory leaks. This probably means the objects are held up longer than required (this may be expected in some scenarios). Profiling the application blocks gives you an idea of the type of objects that are being promoted to Gen 2 of the heap. You can then focus on analyzing the culprit code snippet and rectify the problem.

    An efficient allocation pattern should have most of the allocations in Gen 0 and Gen 1 over a period of time.

    There might be certain objects, such as a pinned pool of reusable buffers used for I/O work, that are promoted to Gen 2 when the application starts. The faster this pool of buffers gets promoted to Gen 2, the better.
  - **The fragmentation of the heap.** The heap fragmentation happens most often in scenarios where the objects are pinned and cannot be moved. The memory cannot be efficiently compacted around these objects. The longer these objects are pinned, the greater the chances of heap fragmentation. As mentioned earlier, there might be a pool of buffers that needs to be used for I/O calls. If these objects are initialized when the application starts, they quickly move the Gen 2, where the overhead of heap allocation is largely removed.
  - "**Side effect" allocations.** Large number of side effect allocations take place because of some calls in a loop or recursive functions, such as the calls to string-related functions **String.ToLower()** or concatenation using the **+** operator happening in a loop. This causes the original string to be discarded and a new string to be allocated for each such operation. These operations in a loop may cause significant increase in memory consumption.

  You can also analyze memory leaks by using debugging tools, such as WinDbg from the Windows Resource Kit. Using these tools, you can analyze the heap allocations for the process.

  For more information about how to use WinDbg for memory leaks, see "Debugging Memory Problems" in "Production Debugging for .NET Framework Applications" on MSDN at:
  *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/DBGch02.asp.*

  For more information about best practices related to garbage collection, see "Chapter 5 — Improving Managed Code Performance" of *Improving .NET Application Performance and Scalability* on MSDN at:
  *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/scalenetchapt05.asp.*

- **Cost of serialization.** There may be certain scenarios when the application block needs to serialize and transmit data across processes or computers. Serializing data involves memory overhead that can be quite significant, depending on the amount of data and the type of serializer or formatter used for serialization. You need to instrument your code to take the snapshots of memory utilized by the garbage collector before and after serialization.

  For more information about how to calculate the cost of serialization, see "Chapter 15 — Measuring .NET Application Performance" of *Improving .NET Application Performance and Scalability* on MSDN at: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/scalenetchapt15.asp.*

- **Contention and deadlock issues.** Contention and deadlock issues mostly surface under high load conditions. The input from load testing (during black box testing) give you information about the potential execution paths where contention and deadlocks issues are suspected. For example, in the case of the CMAB, you may suspect a deadlock if you see the requests timing out when trying to update a particular information in the persistent medium.

  You need to analyze these issues with invasive profiling techniques, such as using WindDbg tool, in the production environment on a live process or by analyzing the stack dumps of the process.

  For more information about production debugging of contention and deadlock issues, see "Debugging Contention Problems" in "Production Debugging for .NET Framework Applications" on MSDN at: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/DBGch03.asp.*

- **Time taken for executing a code path.** For scenarios where performance is critical, you can profile the time they take. Timing a code path may require custom instrumentation of the appropriate code. There are also various tools available that help you measure the time it takes for a particular scenario to execute by automatically creating instrumented assemblies of the application block. The profiling for time taken may be for complete execution of a usage scenario, an internal function, or even a particular loop within a function.

  For more information about how to time managed code and a working sample, see "How To: Time Managed Code Using QueryPerformanceCounter and QueryPerformanceFrequency" in *Improving .NET Application Performance and Scalability* on MSDN at: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/ScaleNetHowTo09.asp.*

- **Profiling for excessive resource utilization.** The input from a performance test may show excessive resource utilization, such as CPU, memory, disk I/O, or network I/O, for a particular usage scenario. But you may need to profile the

code to track the piece of code that is blocking resources disproportionately. This might be an expected behavior for a particular scenario in some circumstances. For example, an empty **while** loop may pump up the processor utilization significantly and is something you should track and rectify; whereas, a computational logic that involves complex calculations may genuinely warrant high processor utilization.

For more information about how to use CLR Profiler, see "How To: Use CLR Profiler" in *Improving .NET Application Performance and Scalability* on MSDN at: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/scalenethowto13.asp.*

## Step 3: Test the Internal Subroutines

Thoroughly test all internal subroutines for every type of input. The subroutines that are internally called by the public API to process the input may be working as expected for the expected input types. However, after a thorough code review, you may notice that there are some expressions that may fail for certain types of input. This warrants the testing of internal methods and subroutines by developing NUnit tests for internal functions after a thorough code review. Following are some examples of potential pitfalls:

- The code analysis reveals that the function may fail for a certain input value. For example, a function expecting numeric input may fail for an input value of 0.
- In the case of the CMAB, the function reads information from the cache. The function returns the information appropriately if the cache is not empty. However, if during the process of reading, the cache is flushed or refreshed, the function may fail.
- The function may be reading values in a buffer before returning them to the client. Certain input values might result in a buffer overflow and loss of data.
- The subroutine does not handle an exception where the remote call to a database is not successful. For example, in the CMAB, if the function is trying the update the SQL Server information but the SQL Server database is not available, it does not log the application in the appropriate event sink.

## Step 4: Test Loops and Conditional Statements

The application block may contain various types of loops, such as simple, nested, concatenated, and unstructured loops. Although unstructured loops require redesigning, the other types of loops require extensive testing for various inputs. Loops are critical to the application block performance because they magnify seemingly trivial problems by iterating through the loop multiple times.

Some of the common errors could cause the loop to execute infinite times. This could result in excessive CPU or memory utilization resulting in the application failing.

Therefore, all loops in the application block should be tested for the following conditions:

- Provide input that results in executing the loop zero times. This can be achieved where the input to the lower bound value of the loop is less than the upper bound value.
- Provide input that results in executing the loop one time. This can be achieved where the lower bound value and upper bound value are the same.
- Provide input that results in executing the loop a specified number of times within a specific range.
- Provide input that the loop might iterate *n*, *n*-1, and *n*+1 times. The out-of-bound loops (*n*-1 and *n*+1) are very difficult to detect with a simple code review; therefore, there is a need to execute special test cases that can simulate such cases.

When testing nested loops, you can start by testing the innermost loop, with all other loops set to iterate a minimum number of times. After the innermost loop is tested, you can set it to iterate a minimum number of times, and then test the outermost loop as if it was a simple loop.

Also, all of the conditional statements should be completely tested. The process of conditional testing ensures that the controlling expressions have been exercised during testing by presenting the evaluating expression with a set of input values. The input values ensure that all possible outcomes of the expressions are tested for expected output. The conditional statements can be a relational expression, a simple condition, a compound condition, or a Boolean expression.

## Step 5: Perform Security Testing

White box security testing focuses on identifying test scenarios and testing based on knowledge of implementation details. During code reviews, you can identify areas in code that validate data, handle data, access resources, or perform privileged operations. Test cases can be developed to test all such areas. Following are some examples:

- Validation techniques can be tested by passing negative value, null value, and so on, to make sure the proper error message displays.
- If the application block handles sensitive data and uses cryptography, then based on knowledge from code reviews, test cases can be developed to validate the encryption technique or cryptography methods.

For more information about application areas that can be validated, see the various checklists that are part of *Improving Web Application Security: Threats and Countermeasures* on MSDN at:
*http://msdn.microsoft.com/library/en-us/dnnetsec/html/CI_Index_Of.asp.*

# Tools

This section describes and points to some of the tools, technologies, and methods that can be used during the profiling process.

## Profiling

Profiling tools can profile the application blocks for analyzing resource utilization and the time it takes to complete a particular operation. They can diagnose potential bottlenecks for a particular operation. One such tool available for creating a memory utilization profile is CLR Profiler. It enables users to understand the interaction between the application and the managed heap. It includes information about how and where the memory allocation takes place and how efficient garbage collection is for the application block. CLR Profiler helps identify and isolate problematic code and track down memory leaks.

For more information about how to use CLR Profiler, see "How To: Use CLR Profiler" in Improving .NET Application Performance and Scalability on MSDN at: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/scalenethowto13.asp.*

There are various tools that you can use to profile .NET applications including Intel VTune and Xtremesoft AppMetrics. These tools help you identify and tune your application bottlenecks.

## Instrumentation

Instrumentation involves adding code to the application block. The code that is added will generate events that can be logged in various event sinks. These events can be used to capture application specific metrics, profiling, and tracing of the code. The various technologies that can be used to instrument application blocks are as follows:

- **Enterprise Instrumentation Framework (EIF).** EIF is a flexible and configurable instrumentation framework that encapsulates the functionality of Event Tracing for Windows (ETW), WMI (Windows Management Instrumentation), and event log service. It allows you to publish information such as errors, warnings, audits, and even business-specific events with the help of an extensible event schema. EIF also provides tracing of business processes and application blocks' execution paths.

  For more information, see "Enterprise Instrumentation Framework (EIF)" in the Microsoft® Visual Studio Developer Center on MSDN at: *http://msdn.microsoft.com/vstudio/teamsystem/eif/.*

  For more information about how to use EIF, see "How To: Use EIF" in *Improving .NET Application Performance and Scalability* on MSDN at: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/scalenethowto14.asp.*

- **Event Tracing for Windows (ETW).** ETW is suitable for logging high frequency events such as warnings and audits. For more information, see "Event Tracing" in "Platform SDK: Performance Monitoring" on MSDN at: *http://msdn.microsoft.com/library/en-us/perfmon/base/event_tracing.asp.*
- **Windows Management Instrumentation (WMI).** WMI is a management information and control technology built into the Microsoft Windows® operating system. WMI collects and analyzes performance-related data for application blocks through event-based monitoring. But logging to a WMI sink is an expensive operation; therefore, it should be used only to log infrequent and critical events and information.

  For more information, see "Windows Management Instrumentation" in "Windows Platform SDK" on MSDN at: *http://msdn.microsoft.com/library/en-us/wmisdk/wmi/wmi_start_page.asp.*
- **Custom performance counters.** Custom performance counters can be used to capture application-specific information. For example, in the CMAB, it can be the time taken to read information from a data store.

  For more information, see the following How To articles in *Improving .NET Application Performance and Scalability* on MSDN at: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/scalenet.asp*
  - "How To: Monitor the ASP.NET Thread Pool Using Custom Counters"
  - "How To: Time Managed Code Using QueryPerformanceCounter and QueryPerformanceFrequency"
  - "How To: Use Custom Performance Counters from ASP.NET"

# Summary

This chapter presented the fundamentals of functionality testing and explained the two categories of testing, namely black box testing and white box testing. The processes for black box testing and white box testing have been laid out step by step in this chapter. These processes, along with the detailed processes presented in other chapters, help ensure that you deliver robust application blocks and/or customize the application blocks for your applications.

# 7

# Globalization Testing for Application Blocks

## Objective

- Learn the process of globalization testing.

## Overview

Globalization is the process of internationalizing the application code so that it supports multiple locales and cultures. The process of internationalization ensures that the application is able to process content, referred to in this chapter as universal data, for any locale with little or no change to the code base. Locale is a combination of both language and cultural environment, including information such as the format of dates, times, currencies, character classification, and sorting rules for strings. Therefore, a goal of internationalization is to separate the locale-dependent content from locale-independent content. Another goal is to use code that is able to process the locale-dependent content. This ensures that the application can be easily localized for a particular culture or locale.

Globalization testing ensures that the code is able to handle the local content for the targeted locales without any data loss or inconsistency. It also ensures that the application block can be integrated with world-ready applications. A world-ready application mostly targets a limited number of locales; however, the goal for an application block may be to support integration with an application that targets any locale or culture. The scope of testing can be limited to testing with a few cultures from different regions to ensure that the application block is able to support content from diverse cultures.

The Configuration Management Application Block (CMAB) is used to illustrate concepts in this chapter.. The requirements for the CMAB are as follows:

- It provides the functionality to read and store configuration information transparently in a persistent storage medium. The storage media are Microsoft® SQL Server™, the registry, and an XML file.
- It provides a configurable option to store the information in encrypted form and plain text by using XML notation.
- It can be used with desktop applications and Web applications that are deployed in a Web farm.
- It caches configuration information in memory to reduce cross-process communication such as reading from any persistent medium. This caching reduces the response time of the request for any configuration information. The expiration and scavenging mechanism for the data that is cached in memory is similar to the cron algorithm in UNIX.
- It can store and return data from various locales and cultures without any loss of data integrity.

# Testing Process for Globalization

The following process formalizes the set of activities that are required to ensure a world-ready application block. The process can be easily customized to suit specific needs for an application block by creating test plans that are specific to particular scenarios.

## Input

The following input is required for the globalization testing process:

- Functional specifications of the application block
- Requirements for the application block
- Deployment scenarios for the application

## Steps

The process for globalization testing is shown in Figure 7.1.

The globalization testing process consists of the following steps:

1. **Create test plans.** Create test plans based on the priority of each scenario and the test platforms it needs to be tested on.
2. **Create the test environment.** Set up the test environment for multiple locales that are targeted for the application block.
3. **Execute the test cases.** Execute the test cases in the test environment.

4. **Analyze the results.** Analyze the results to ensure that there is no data loss or inconsistency in the output.

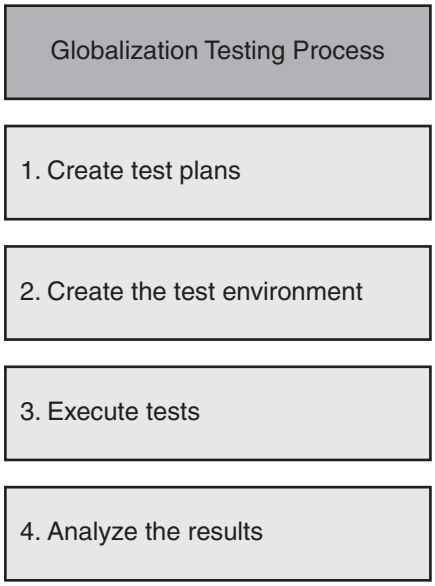


**Figure 7.1**
*The testing process for application blocks*

The next sections describe each of these steps.

## Step 1: Create Test Plans

You must create test plans for the application block scenarios that you must test for globalization. In general, you can develop the test cases (detailed test plan document) and the execution details for each test case (detailed test case document) based on the functional specifications and the requirements document. The functional specification document describes the various interfaces that the application block will expose. The requirements document specifies whether the application block is targeted toward a specific set of locales or whether it is just required to be globalization-compliant.

When you create the test plans, you should:

- Decide the priority of each scenario.
- Select the test platform.

The next sections describe each of these considerations.

### Decide the Priority of Each Scenario

To make globalization testing more effective, it is a good idea to assign a priority to each scenario that must be tested. You should check for the following to identify the high priority scenarios:

- Check if the application block needs to support text data in ANSI format.
- Check if the application block extensively processes strings by performing tasks such as sorting, comparison, concatenation, and transformation (conversion to lowercase or uppercase).
- Check if there are certain APIs that can accept locale-specific information such as address, currency, dates, and numerals.
- Use files for data storage or data exchange (for example, Microsoft Windows metafiles, security configuration tools, and Web-based tools).

In the case of the CMAB, the primary usage functionality is to store and retrieve configuration information from various data stores. Some of the globalization-related testing should test the following high priority scenarios in the CMAB to make sure that the data is stored and retrieved without any loss of integrity and consistency:

- The CMAB is able to store locale-specific information such as currency, dates, and strings in the supported data stores without any loss of data integrity.
- CMAB is able to return the locale-specific information in its original form if the locale of the user requesting the information is the same as the locale of the stored data. There may be a requirement that if users from diverse locales are accessing the information, the information should be converted to the user's locale before it is returned. This requires development of custom handlers that can deserialize the data to meet such a requirement for a particular scenario. However, the globalization testing needs to ensure that such customization is possible with the application block.

  Consider the scenario where the CMAB is integrated with an online chat application that can be accessed from different time zones. The Web server and the application clients fall under different time zones in such a way that the local time at the server and the local time at the client are different. During testing, you should ensure that the custom handlers convert the local time (local data) from the user into universal time (universal data) before storing it as configuration values. In this way, each user sees what time other users logged in according to his or her local time zone.

The following code snippet illustrates this scenario.

```
//Converts the local time to universal time before storing it for UserA
System.DateTime univDateTime = localDateTime.ToUniversalTime();
//Stores the universal date to the default configuration section
ConfigurationManager.Items("universalTime") = univDateTime.ToString("G",
DateTimeFormatInfo.InvariantInfo);
//The universal time is retrieved and converted to a value according to the //
time for the user making the request
```

The stored configuration values can be accessed using the CMAB and displayed in the local time, as and when required. If a user from a different time zone requests the same configuration information, the value can be converted to his or her time zone.

- Assuming that the CMAB provides out-of-the-box support for a fixed number of locales by providing satellite assemblies for resources, you should test all scenarios that result in an exception. Testing these scenarios ensures that the exception messages from the CMAB are on expected lines without any truncation or deformation (loss of characters or the introduction of junk characters) of the actual string.

## Select a Test Platform

Identify the operating system that testing is to be performed on. If the requirements explicitly state that the application block needs to support a specific set of cultures, you have the following options for choosing a test platform:

- Use the local build of the operating system. You can use the local build of the operating system, such as the U.S. build, and install different language groups. The application that is used to test the application block can then change the current UI culture and test that the exception messages and other data returned by the application block are in accordance with the current UI culture.
- Use the Multilanguage User Interface (MUI) operating system. The user can change the language that the UI of the operating system will be displayed in and test the application integrating the application block. The application block should also be able to return the error messages and other data based on the current culture settings. This approach is easier than installing multiple localized versions of the operating system.
- Use the localized build of the target operating system. This approach does not have a significant advantage over the preceding options.

If you do not have an explicit requirement for the locales that must be supported by the application block, you can test by installing a minimum of two language groups from diverse regions, such as Japanese and German. This ensures that the application block is able to support locales from diverse cultures.

### Sample Test Scenario

Consider an auction site that uses the CMAB to store and retrieve the prices that are quoted by various users. Suppose UserA (from Germany) quotes a price of  1,500 (euro) for his tennis kit through the auction site. UserB (from England) logs on to the auction site and views UserA's offer.

The price displayed to UserB should be £1,000 (British pounds). This is the equivalent of  1,500.

The following code illustrates the session for UserA.

```
//Configures the server language as U.S.-English for storage purposes
ConfigurationManager.Items("serverLanguage") = "en-US";
//Configures the language preference of the client as German
ConfigurationManager.Items("localLanguage") = "de-DE";

//Sets the culture/locale of the current thread to German
Thread.CurrentThread.CurrentCulture = new CultureInfo (ConfigurationManager.Items
("localLanguage"));

//Stores price quoted by UserA in euro
Double dblLocalPrice = Double.Parse(txtPrice.text, NumberStyles.Currency);

//Convert Currency Value
//User-defined function to convert currency value from euro (XEU)
//to U.S. dollars (USD)
Double dblServerPrice = convCurrency (dblLocalPrice);
Thread.CurrentThread.CurrentCulture = new CultureInfo (ConfigurationManager.Items
("serverLanguage"));
//Sets the culture/locale of the current thread to server's local language
//Storing the price in server through CMAB
ConfigurationManager.Items ("price") = dblServerPrice.ToString ("c");
```

The following code illustrates the session for UserB.

```
//Configures the language preference of the client as U.K. English
ConfigurationManager.Items ("localLanguage") = "en-GB";

//Retrieves the price in server through the CMAB
Double dblServerPrice = Double.Parse (ConfigurationManager.Items ("price"),
NumberStyles.Currency);
```

```
//User-defined function to convert currency value from U.S. dollars (USD)
//to British pounds (GBP)
Double dblLocalPrice = convCurrency (dblServerPrice);
//Sets the culture/locale of the current thread to U.K. English
Thread.CurrentThread.CurrentCulture = new
CultureInfo(ConfigurationManager.Items("localLanguage"));

//Displays the converted currency in U.K. English format
lblPrice.Text = dblLocalPrice.ToString("c");
```

## Step 2: Create the Test Environment

As mentioned earlier, to perform globalization testing, you must install multiple language groups on the test computers. After you install the language groups,, make sure that the culture or locale is not your local culture or locale.

You should make sure that the locale of the server that the test harness for the application block is hosted on is not the same locale as that of the test computers.

For the culture or locale example (the second test scenario in Step 1), configure the following:

- **Server.** Install U.S.-English language and time zone support.
- **UserA's system.** Install German language and time zone support.
- **UserB's system.** Install U.K. English and time zone support.

## Step 3: Execute Tests

After the environment is set for globalization testing, you should focus on potential globalization problems when you run your functional and integration test cases. Consider the following guidelines for the test cases to be executed:

- Put greater emphasis on test cases that deal with passing parameters to the application block. For the sample scenario considered earlier, you can consider test cases that determine the correct selection of the culture or locale in accordance with the user's location and choice of environment. It focuses on the following code for the preceding example.

  ```
  //Sets the culture/locale of the current thread to German
  Thread.CurrentThread.CurrentCulture = new
  CultureInfo(ConfigurationManager.Items("localLanguage"));
  //Stores price quoted by UserA, in euros
  Double dblLocalPrice = Double.Parse (txtPrice.text);
  ```

- Focus on test cases that deal with the input and output of strings, directly or indirectly.
- During testing, you should use test data that contains mixed characters from various languages and different time zones.

You can create automated test cases by using a test framework such as NUnit. The test stubs can focus on passing various types of input to the application block API. In this way, you automate the execution of test cases and ensure that each new build of the application block during the development cycle is world-ready.

## Step 4: Analyze the Results

The tests may reveal that the functionality of the application block is not working as intended for different locales. In the worst-case scenario, the functionality may fail completely, but in most of the scenarios, you may have issues similar to the following:

- Random appearance of special characters, such as question marks, ANSI characters, vertical bars, boxes, and tildes
- Incorrect formatting of data, such as date and currency, in the return values from the application block
- Error message text that does not appear in accordance with the current locale setting

Each of these issues has a different root cause. For example, the appearance of boxes or vertical bars indicates that the selected font cannot display some of the characters; the appearance of question marks indicates problems with Unicode-to-ANSI conversion.

For a complete list of globalization-related issues, see "Globalization and Localization Issues" on MSDN at:
*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsent7/html/vxconGlobalizationLocalization.asp.*

Usually, a simple code review of the module reveals mistakes such as hard-coded strings, misuse of an overloaded API that takes the culture or locale related inputs, or an incorrectly set culture-related property for the thread that the call is being executed in. There may be other scenarios where the code converts strings from lowercase to uppercase before performing a case-insensitive comparison. This may produce unexpected results for certain languages, such as Chinese and Japanese, that do not have the concept of uppercase and lowercase characters.

## Summary

This chapter explained the process for globalization testing of application blocks. It is important to note that you can significantly reduce rework due to globalization testing by performing a thorough code review to make sure that the code complies with all globalization-related best practices. In most scenarios, the testing effort can be focused around a specific set of issues that affect applications in general. For a complete list of globalization-related issues, see "Globalization and Localization Issues" on MSDN at:
*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsent7/html/vxconGlobalizationLocalization.asp.*

## Additional Resources

For more information, see the following resources:

- For more information about globalization testing, see "Globalization Testing" on MSDN at:
  *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsent7/html/vxconglobalizationtesting.asp.*
- For a step-by-step approach toward globalization, see "Globalization Step-by-Step" on the Microsoft Global Development and Computing Portal at:
  *http://www.microsoft.com/globaldev/getwr/steps/wrguide.mspx.*
- For more information about best practices for globalization, see "Best Practices for Developing World-Ready Applications" in the *.NET Framework Developer's Guide* on MSDN at:
  *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconbestpracticesforglobalapplicationdesign.asp.*
- For more information about best practices for globalization and localization, see "Best Practices for Globalization and Localization" on MSDN at:
  *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsent7/html/vxconBestGlobalizationPractices.asp.*
- For more information about issues related to globalization and localization, see "Globalization and Localization Issues" on MSDN at:
  *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsent7/html/vxconGlobalizationLocalization.asp.*

# 8

# Performance Testing for Application Blocks

## Objectives

- Learn performance testing fundamentals.
- Learn about load testing an application block.
- Learn about stress testing an application block.
- Learn about the tools that are used for performance testing.

## Overview

Performance testing an application block involves subjecting it to various load levels. The goals of performance testing can be summarized as follows:

- **To verify that the application block (or a prototype) meets the performance objectives within the budgeted constraints of resource utilization.** The performance objectives can include several different parameters such as the time it takes to complete a particular usage scenario (known as response time) or the number of concurrent or simultaneous requests that can be supported for a particular operation at a given response time. The resource constraints can be set with respect to server resources such as processor utilization, memory, disk I/O, and network I/O.
- **To analyze the behavior of the application block at various load levels.** The behavior is measured in metrics related to performance objectives and other metrics that help to identify the bottlenecks in the application block.
- **To identify the bottlenecks in the application block.** The bottlenecks can be caused by several issues such as memory leaks, slow response times, or contention under load.

Performance testing for an application block can be broadly categorized into two types:

- **Load testing.** Load testing helps you to monitor and analyze behavior of an application block under normal and peak load conditions. Load testing enables you to verify that the application block meets the desired performance objectives.
- **Stress testing.** Stress testing helps to analyze behavior of an application that integrates the application block when it is pushed beyond the peak load conditions. The goal of stress testing is to identify problems that occur only under high load conditions.

You can conduct performance testing during various phases of the development life cycle:

- **Design phase.** During this phase of the life cycle, you can conduct performance testing on a prototype to evaluate whether a particular design would meet the targeted performance objectives.
- **Implementation/construction phase.** During this phase of the life cycle, you can conduct performance testing to validate that the implementation of the modules meets the performance objectives.
- **Integration testing phase.** During this phase of the life cycle, you can conduct performance testing to ensure that the application that is integrating the application block can meet its own performance objectives.

You can either directly load the API for the application block by using a load generator or develop a prototype application that integrates the application block. Either approach is valid if the overhead of a prototype application is minimal. If you decide on the prototype application approach, you should make sure that the prototype application does not perform any expensive rendering operations or other actions that are not relevant to testing the application blocks.

The Configuration Management Application Block (CMAB) is used to illustrate concepts in this chapter. The requirements for the CMAB are the following:

- It provides the functionality to read and store configuration information transparently in a persistent storage medium. The storage media are the Microsoft® SQL Server™ database system, the registry, and XML files.
- It provides a configurable option to store the information in encrypted form in plain text using XML notation.
- It can be used with desktop applications and with Web applications that are deployed in a Web farm.
- It caches configuration information in memory to reduce cross-process communication such as reading from any persistent medium. This caching reduces the response time of the request for configuration information. The expiration and scavenging mechanism for the data that is cached in memory is similar to the CRON algorithm in UNIX.

The CMAB can store and return data from various locales or cultures without losing any data integrity.

In the case of the CMAB, the performance objectives are as follows (please note that these objectives are fictitious and are for illustration purposes only):

- The CPU overhead should not be more than 7 to 10 percent.
- The application block should be able to support a minimum of 200 concurrent users for the reading of data from SQL Server.
- The application block should be able to support a minimum of 150 concurrent users for the writing of data to SQL Server.
- The response time for a client is not more than 2 seconds for the given concurrent load. (The client is firing the request from a 100 megabits per second [Mbps] VLAN in the test lab.)

For more information about performance testing fundamentals, see "Chapter 16 — Testing .NET Application Performance" of *Improving .NET Application Performance and Scalability* on MSDN at:
*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/scalenetchapt16.asp.*

## Performance Objectives

Performance objectives are captured in the requirements phase and early design phase of the application life cycle. All performance objectives, resource budget data, key usage scenarios, and so on, are captured as a part of the performance modeling process. The performance modeling artifact serves as important input to the performance testing process. In fact, performance testing is a part of the performance modeling process; you may update the model depending on the application life cycle phase in which you are executing the performance tests.

The performance objectives may include some or all of the following:

- **Workload.** If the application block is to be integrated with a server-based application, it will be subject to a certain load of concurrent and simultaneous users. The requirements may explicitly specify the number of concurrent users that should be supported by the application block for a particular operation. For example, the requirements for an application block may be 200 concurrent users for one usage scenario and 300 concurrent users for another usage scenario.
- **Response time.** If the application block is to be integrated with a server-based application, the response time objective is the time it takes to respond to a request for the peak targeted workload on the server. The response time can be measured in terms of Time to First Byte (TTFB) and Time to Last Byte (TTLB). The response time depends on the load that is on the server and the network bandwidth over which the client makes a request to the server. The response time is specified for

different usage scenarios of the application block. For example, a write feature may have a response time of less than 4 seconds; whereas a read scenario may have a response time of less than 2 seconds for the peak load scenario.

- **Throughput.** Throughput is the number of requests that can be served by the application per unit time. A simple application that integrates the application block is supposed to process requests for the targeted workload within the response time goal. This goal can be translated as the number of requests that should be processed per unit time. For an ASP.NET Web application, you can measure this value by monitoring the **ASP.NET\Request/sec** performance counter. You can measure the throughput in other units that help you to effectively monitor the performance of the application block; for example, you can measure read operations per second and write operations per second.
- **Resource utilization budget.** The resource utilization cost is measured in terms of server resources, such as CPU, memory, disk I/O, and network I/O. The resource utilization budget is the amount of resources consumed by the application block at peak load levels. For example, the processor overhead of the application block should not be more than 10 percent.

For more information about performance modeling, see "Chapter 2 — Performance Modeling" of *Improving .NET Application Performance and Scalability* on MSDN at: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html /scalenetchapt02.asp.*

# Load Testing

Load testing analyzes the behavior of the application block with workload varying from normal to peak load conditions. This allows you to verify that the application block is meeting the desired performance objectives.

## Input

The following input is required toload test an application block:

- Performance model( workload characteristics, performance objectives, and resource budget allocations)
- Test plans

## Load Testing Steps

Load testing involves six steps:

1. **Identify key scenarios.** Identify performance-critical scenarios for the application block.
2. **Identify workload.** Distribute the total load among the various usage scenarios identified in Step 1.

3. **Identify metrics.** Identify the metrics to be collected when executing load tests.
4. **Create test cases.** Create the test cases for load testing of the scenarios identified in Step 1.
5. **Simulate load.** Use the load-generating tools to simulate the load for each test case, and use the performance monitoring tools (and in some cases, the profilers) to capture the metrics.
6. **Analyze the results.** Analyze the data from the performance objectives as the benchmark. The analysis also identifies potential bottlenecks.

The next sections describe each of these steps.

## Step 1: Identify Key Scenarios

Generally, you should start by identifying scenarios that can have a significant performance impact or that have explicit performance goals. In the case of application blocks, you should prepare a prioritized list of usage scenarios, and all of these scenarios should be tested.

In the case of the CMAB, the two major functionalities are reading and writing configuration data. The CMAB functionalities can be extended more based on various scenarios, such as whether caching is enabled or disabled, usage of different data stores, or different encryption providers, and so on. Therefore, the load-testing scenarios for the CMAB are the combinations of all the configuration options. The following are some of the scenarios for the CMAB:

- Read a declared configuration section from a file store with caching disabled and data encryption enabled.
- Write configuration data to a file store with encryption enabled.
- Read configuration data from a SQL store with caching and data encryption enabled.
- Write configuration data to a SQL store with data encryption enabled.
- Initialize the Configuration Manager for the first time when the Configuration Manager is performing user operations.

For the CMAB, performance degradation probability is high in a case where data must be written to a file store, because concurrent write operations are not supported on a file and the response time is expected to be greater in this case.

## Step 2: Identify Workload

In this step, you identify the workload for each scenario or distribute the total workload among the scenarios. Workload allocation involves specifying the number of concurrent users that are involved in a particular scenario, the rate of requests, and the pattern of requests. You may have a workload defined for each usage scenario in terms of concurrent users (that is, all users firing requests at a given

instant without any sleep time between requests). For example, the CMAB has a targeted workload of 200 concurrent users for a read operation on the SQL store with caching disabled and encryption enabled.

In most real-world scenarios, the application block may be performing parallel execution of multiple operations from different scenarios. You may therefore want to analyze how the application block performs with a particular workload profile that is a mix of various scenarios for a given load of simultaneous users (that is, all users have active connections, and all of them may not be firing requests at same time), with two consecutive requests separated by specific *think time* (that is, the time spent by the user between the two consecutive requests).

Workload characteristics are determined by using workload modeling techniques. For more information about workload modeling, see "Chapter 16 — Testing .NET Application Performance" of *Improving .NET Application Performance and Scalability* on MSDN at:
*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/scalenetchapt16.asp.*

The following steps help you to identify the workload profile:

- **Identify the maximum number of simultaneous users accessing each of the usage scenarios in isolation.** This number is based on the performance objectives identified during performance modeling. For example, in the case of the CMAB, the expected load of users is 1,200 simultaneous users.
- **Identify the expected mix of usage scenarios.** In most real-world server – based applications, a mix of application block usage scenarios might be accessed by various users. You should identify each mix by naming each one as a unique profile. Identify the number of simultaneous users for each scenario and the pattern in which the users have to be distributed across test scenarios. Distribute the workload based on the requirements that the application blocks have been designed for. For example, the CMAB is optimized for situations where read operations outnumber write operations; it is not meant for online transaction processing (OLTP) – style applications. Therefore, the workload has to be distributed so only a small part of the workload is allocated for write operations. Group users together into user profiles based on the key scenarios they participate in.

  For example, in the case of the CMAB, for any given database store, there will be a read profile and a write profile. Each profile has its respective use case as the dominant one. A sample composition of a read profile for a SQL store is shown in Table 8.1. The table assumes that out of a total workload of 1,000 simultaneous users, 600 users are using the SQL store.

**Table 8.1: CMAB Read Profile for SQL Server Database Store**

| Read profile for SQL Server | Percentage of the workload for SQL store | Simultaneous users |
|---|---|---|
| Reading from a SQL store | 90 | 540 |
| Writing to a SQL store | 10 | 60 |
| Total | 100 | 600 |

In this way, you will have the read profiles for the registry and XML file data stores. Assuming that each of these gets a share of 200 users out of the total workload, the workload profile for the CMAB is as shown in Table 8.2.

**Table 8.2: Sample Workload Profile for the CMAB**

| User profile | Percentage of the workload for SQL store | Simultaneous users |
|---|---|---|
| Read profile — SQL Server | 60 | 600 |
| Read profile — registry | 20 | 200 |
| Read profile — XML file | 20 | 200 |
| Total | 100 | 1000 |

In addition to helping you simulate real-world scenarios, testing for combinations like those shown in the tables helps you to identify any possible negative impact on performance because of any contention. In the CMAB example, when a configuration is being read and written into a data store at the same time, it is locked for writing, and the read operation has to wait for the locks to be released.

- **Identify the average think time.** Think time is the time spent by the user between two consecutive requests. This value may be as low as zero, a fixed value, or a random value in a range of numbers. In the case of the CMAB, the think time is a random think time of zero to 3 seconds.
- **Identify the duration of test for each of the profiles identified above.** You need to run load tests for the user profiles and each isolated scenario identified earlier. The duration of test depends on the end goal of running the performance tests and can vary from 30 minutes to more than 100 hours. If you are interested in learning whether the usage scenario meets the performance objectives, a quick test of 20 minutes will suffice. However, if you are interested in analyzing the behavior of a write scenario over a sustained load for long hours when the database log file size tends to grow, you might want to run a long test lasting 4 to 5 days.

## Step 3: Identify Metrics

Identify metrics that are relevant to your performance objectives and those that help you to identify bottlenecks. As the number of test iterations increases, more metrics can be added based on the analysis in previous iterations to identify potential bottlenecks.

Regardless of the application block, there are metrics that should be captured during load testing. These metrics are listed in Table 8.3.

**Table 8.3: Metrics to Be Measured for All Test Cases**

| Object | Counter | Instance |
|---|---|---|
| Processor | %Processor Time | _Total |
| Process | Private Bytes | <Process> |
| Memory | Available MBytes | Not applicable |
| ASP.NET | Request Execution Time | Not applicable |
| ASP.NET | Requests Rejected | Not applicable |
| ASP.NET applications | Requests/Sec | Your virtual directory |

The metrics in Table 8.3 give a coarse-grained view of memory and CPU utilization and performance for any application block during the early iteration cycles of the load test. In subsequent iterations, you can add more metrics for a fine-grained picture of resource utilization and performance. For example, suppose that during the first iteration of load tests, the ASP.NET worker process shows a marked increase in the **Process\Private Bytes** counter, indicating a possible memory leak. In the subsequent iterations, additional memory counters related to generations can be captured to study the memory allocation pattern for the application. These counters are listed in Table 8.4.

**Table 8.4: Performance Counters for Analyzing the Managed Memory**

| Object | Counter | Instance |
|---|---|---|
| .NET CLR Memory | # Bytes in all Heaps | Not applicable |
| .NET CLR Memory | # Gen 0 Collections | Not applicable |
| .NET CLR Memory | # Gen 1 Collections | Not applicable |
| .NET CLR Memory | # Gen 2 Collections | Not applicable |

In addition to the metrics in Table 8.2, you should capture metrics specific to each application block's performance objectives and test scenario goals. This may require additional instrumentation of the code by adding custom performance counters. For example, in the case of the CMAB, you can monitor the counters listed in Table 8.5

while running tests for performing read or write operations in isolation on a data store.

**Table 8.5: Performance Counters for a Read or Write Operation in Isolation**

| Objective | Counter | Instance |
|---|---|---|
| Throughput | ASP.NET\Requests/Sec | Your virtual directory |
| Execution time | ASP.NET\Request Execution Time | Not applicable |

However, if you need to perform a mix of usage scenarios, simultaneously capturing out-of-the-box counters gives you only average values processed by the server. If you are interested in the number of read operations per second and write operations per second, you may need to add custom performance counters in the application block source code.

For a detailed list of performance counters and the scenarios, see "Chapter 15 — Measuring .NET Application Performance" of *Improving .NET Application Performance and Scalability* on MSDN at: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/scalenetchapt15.asp.*

## Step 4: Create Test Cases

The test cases are created based on the scenarios and the profile mix identified in the previous steps. In general, the inputs for creating the test cases are performance objectives, workload characteristics, and the identified metrics. Each test case should mention the expected results in such a way that each test case can be marked as a pass or fail after execution.

### Test Case for the CMAB Sample Application Block

In the case of the CMAB, the test case would be the following:

- **Scenario:** Reading configuration data from a SQL store with data caching and data protection options enabled
- **Number of users:** 200 concurrent users
- **Test duration:** 40 minutes
- **Think time:** Random think time of 0 to 3 seconds
- **Expected results:** The expected results are the following:
  - **Throughput:** 200 requests per second (**ASP.NET\Requests/sec** performance counter)
  - **Processor\%Processor Time:** 75 percent
  - **Memory\Available Mbytes:** 25 percent of total RAM
  - **Request execution time:** 2 seconds (on 100 Mbps LAN)

## Step 5: Simulate Load

The load is generated using load generator tools, such as Microsoft Application Center Test (ACT), that simulate the number of users as specified in the workload characteristics. For each test cycle, incrementally increase the load. You should continue to increase the load and record the behavior until the threshold crosses the limit for the resources identified in the performance objectives. The number of users can also be increased slightly beyond the peak operating levels. The metrics are captured using performance monitoring tools, such as ACT or System Monitor.

For more information about using ACT for performance testing, see "How To: Use ACT to Test Performance and Scalability" of *Improving .NET Application Performance and Scalability* on MSDN at:
*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/ScaleNetHowTo10.asp.*

## Step 6: Analyze the Results

The metrics captured at various load levels should be analyzed to determine whether the performance of the application block being tested shows a trend toward or away from the performance objectives. The measured metrics should also be analyzed to diagnose potential bottlenecks. Based on the analysis, you can capture additional metrics in the subsequent test cycles.

For a general template for creating the load testing report, see the "Reporting" section in "Chapter 16 — Testing .NET Application Performance" of *Improving .NET Application Performance and Scalability* on MSDN at:
*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/scalenetchapt16.asp.*

Primarily, load testing is done from a black box testing perspective. However, load testing results (such as information about potential deadlocks, contention, and memory leaks) can be used as input for the white box testing phase. You can analyze the code more by profiling using specialized tools such as WinDbg from the Windows Resource Kit.

The load testing process produces the following output:

- Updated test plans
- Potential bottlenecks that need to be analyzed in the white-box testing phase
- Peak operating capacity
- Behavior of the application block at various load levels

# Stress Testing

Stress testing an application block means subjecting it to load beyond the peak operating capacity and at the same time denying resources that are required to process the load. An example of stress testing is hosting the application that uses the application block on a server that already has a process utilization of more than 75 percent because of existing applications, and subjecting the application block to a concurrent load above the peak operating capacity.

The goal of stress testing is to evaluate how the application block responds under such extreme conditions. Stress testing helps to identify problems that occur only under high load conditions. Stress testing application blocks identifies problems such as memory leaks, resource contentions, and synchronization issues.

Stress testing uses the analysis from load testing. The test scenarios and the maximum operating capacities are obtained from load testing.

The stress testing approach can be broadly classified into two types: sustained testing and maximal testing. The difference is usually the time the stress test is scheduled to run for, because a sustained stress test usually has a longer execution time than a maximal stress test. In fact, stress testing can accomplish its goals by intensity or quantity. A maximal stress test tends to concentrate on intensity; in other words, it sets up much more intense situations than would otherwise be encountered but it attempts to do so in a relatively short period of time. For example, a maximal stress test may have 500 users concurrently initiating a very data-intensive search query. The intensity is much greater than a typical scenario. Conversely, a sustained stress load tends to concentrate on quantity because the goal is to run much more in terms of the number of users or functionality, or both, than would usually be encountered. So, for example, a sustained stress test would be to have 2000 users run an application designed for 1000 users.

## Input

The following input is required for stress testing an application block

- Performance model (workload characteristics, performance objectives, key usage scenarios, resource budget allocations)
- Potential problematic scenarios from the performance model and load testing
- Peak load capacity from load testing

## Stress Testing Steps

Stress testing includes the following steps:

1. **Identify key scenarios.** Identify test scenarios that are suspected to have potential bottlenecks or performance problems, using the results of the load-testing process.

2. **Identify workload.** Identify the workload to be applied to the scenarios identified earlier using the workload characteristics from the performance model, the results of the load testing, and the workload profile used in load testing.
3. **Identify metrics.** Identify the metrics to be collected when stress testing application blocks. The metrics are now identified to focus on potential performance problems that may be encountered during the testing process.
4. **Create test cases.** Create the test cases for the key scenarios identified in Step 1.
5. **Simulate load.** Use load-generating tools to simulate the load to stress test the application block as specified in the test case, and use the performance monitoring and measuring tools and the profilers to capture the metrics.
6. **Analyze the results.** Analyze the results from the perspective of diagnosing the potential bottlenecks and problems that occur only under continuous extreme load condition and report them in a proper format.

The next sections describe each of these steps.

## Step 1: Identify Key Scenarios

Identify scenarios from the test cases used for load testing that may have a performance problem under high load conditions.

To stress test the application block, identify the test scenarios that are critical from the performance perspective. Such scenarios are usually resource-intensive or frequently used. These scenarios may include functionalities such as the following:

- Synchronizing access to particular code that can lead to resource contention and possible deadlocks
- Frequent object allocation in various scenarios, such as developing a custom caching solution, and creating unmanaged objects

For example, in the case of the CMAB, the test scenarios that include caching data and writing to a data store such as file are the potential scenarios that need to be stress tested for memory leaks and synchronization issues, respectively.

## Step 2: Identify Workload

Identify the workload for each of the performance-critical scenarios. Choose a workload that stresses the application block sufficiently beyond the peak operating capacity.

You can capture the peak operating capacity for a particular profile from the load testing process and incrementally increase the load and observe the behavior at various load conditions. For example, in the case of the CMAB, if the peak operating capacity for a writing to a file scenario is 150 concurrent users, you can start the stress testing by incrementing the load with a delta of 50 or 100 users and analyze the application block's behavior.

## Step 3: Identify Metrics

Identify the metrics that help you to analyze the bottlenecks and the metrics related to your performance objectives. When load testing, you may add a wide range of metrics (during the first or subsequent iterations) to detect any possible performance problems, but when stress testing, the metrics monitored are focused on a single problem. For example, to capture the contentions in the application block code when stress testing the "writing to a file" scenario for the CMAB, you need to monitor the counters listed in Table 8.6.

**Table 8.6: Metrics to Measure When Stress Testing the "Writing to a File" Scenario**

| **Base set of metrics:**<br>**Object** | **Counter** | **Instance** |
|---|---|---|
| Processor | % Processor Time | _Total |
| Process | Private Bytes | aspnet.wp |
| Memory | Available MBytes | Not applicable |
| ASP.NET | Requests Rejected | Not applicable |
| ASP.NET | Request Execution Time | Not applicable |
| ASP.NET applications | Requests/Sec | Your virtual directory |
| **Contention-related metrics**<br>**Object** | **Counter** | **Instance** |
| .NET CLR LocksAndThreads | Contention Rate/sec | aspnet_wp |
| .NET CLR LocksAndThreads | Current Queue Length | aspnet.wp |

## Step 4: Create Test Cases

The next step is to create test cases. When load testing, you have a list of prioritized scenarios, but when stress testing you identify a particular scenario that needs to be stress tested. There may be more than one scenario or there may be a combination of scenarios that you can stress test during a particular test run to reproduce a potential problem.

Document your test cases for a list of scenarios identified in Step 1.

### Test Case for the CMAB Sample Application Block

In the case of the CMAB, a sample test case would be the following:

- **Scenario:** Write to a file
- **Number of users:** 350 concurrent users

- **Test duration:** 10 hours
- **Think time:** 0
- **Expected results:**
  - The ASP.NET worker process should not be recycled.
  - Throughput should not fall below 30 requests per second (**ASP.NET\Requests/sec** performance counter).
  - Response time should not exceed 10 seconds (on 100 Mbps LAN).
  - Server busy errors should not be more than 25 percent of the total response because of contention-related issues.

## Step 5: Simulate Load

The load is simulated using load generator tools such as ACT. The metrics are captured using performance monitoring tools such as System Monitor. You should make sure that the client systems that are used to generate loads do not cross the resource utilization thresholds.

## Step 6: Analyze the Results

The captured metrics should be analyzed for diagnosing the bottleneck. If the metrics are below the accepted level of performance, you may need to do one of the following:

- Debug the code during white-box testing to identify any possible contention issues.
- Examine the stack dumps of the worker process to diagnose the exact cause of deadlocks.
- Perform a design review of the module to identify whether you need to consider a design change to satisfy the performance goal.

For example, to reduce contention when writing to a file, you can update the changes in shadow files, which are exact replicas of the original files, and later merge the changes to the original file.

For more information about stress testing, see "Stress Testing Process" in "Chapter 16 — Testing .NET Application Performance" of *Improving .NET Application Performance and Scalability* on MSDN at: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/scalenetchapt16.asp*.

# Tools

Various tools are used during the performance testing process. These tools can be broadly classified into two categories:

- Load simulators and generators
- Performance monitoring tools

The next sections describe each of these categories.

## Load Simulators and Generators

These tools simulate or generate the specified load in terms of users, active connections, and so on. In addition to generating load, these tools can also help you to gather related metrics, such as response time and requests per second. They can also generate reports that help you to analyze the captured metrics.

One tool that falls into this category is Microsoft Application Center Test (ACT). ACT is designed to stress test Web applications or Web services. ACT is a processor-intensive tool that can quickly stress the client computer. You should distribute the load among various clients running ACT if the client shows high processor utilization. ACT is included with Enterprise editions of Microsoft Visual Studio® .NET 2003 development system.

For more information about how to use ACT, see "How To: Use ACT to Test Performance and Scalability" in *Improving .NET Application Performance and Scalability* on MSDN at:
*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/ScaleNetHowTo10.asp.*

For more general information about ACT, see "Microsoft Application Center Test 1.0, Visual Studio .NET Edition" on MSDN at:
*http://msdn.microsoft.com/library/en-us/act/htm/actml_main.asp.*

## Performance Monitoring Tools

Performance monitoring tools capture the metrics during load and stress testing. The following tools can be used to monitor resource utilization and other performance metrics during testing:

- **System Monitor.** System Monitor is a standard component of the Microsoft Windows® operating system. It can be used to monitor performance objects and counters. It can also be used to monitor instances of various hardware and software components. System Monitor is useful when you need to log the metrics for a particular test duration ranging from a few minutes to a few days.

- **Microsoft Operations Manager (MOM).** MOM provides event-driven operations monitoring and performance tracking capability. MOM is suitable for collecting large amounts of data over a long period of time. MOM agents on individual servers collect data and send it to a centralized server, where the data is stored in a MOM database. For more information, see "Microsoft Operations Manager" on MSDN at:
  *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/momsdk/htm/momstart_1nad.asp.*
- **Network Monitor (NetMon).** NetMon is used to monitor network traffic. It provides statistics such as packet size, network utilization, routing, timing, and other statistics that can be used to analyze system performance when testing the performance of application blocks that have to access resources over the network. For more information, see "Network Monitor" on MSDN at:
  *http://msdn.microsoft.com/library/en-us/netmon/netmon/network_monitor.asp.*

## Summary

This chapter explained the processes for the two types of performance testing, load testing and stress testing, from the perspective of testing of the application blocks. The processes laid down in the chapter are generic and can be customized for your own needs; for example, you may choose to have your profiles for load testing organized differently than suggested in the chapter.

# 9

# Integration Testing for Application Blocks

## Objectives

- Unit test a customized application block.
- Integration test a customized application block.

## Overview

Microsoft application blocks are designed, developed, and tested extensively so that they can be integrated with your applications in an off-the-shelf manner, without any further modification or testing. If you need to modify or extend the existing functionality to suit your application requirements, you may need to unit test the application block before integrating it with your application. There are several benefits to customizing an application block, including the following:

- You can add functionality to the application block so that it is able to fulfill application-specific needs. For example, if the application block supports Microsoft® SQL Server™ 2000, you may need to write custom code to support Oracle databases.
- You can modify the design or code to suit the deployment-specific requirements. For example, you could change the permission and link demand settings in the application block code to suit the application requirements.
- You can extend the interfaces exposed by the application block to write custom implementations, rather than using the implementation provided by application block.

After the application block is customized, you need to unit test the changes to make sure that the customizations do not break existing functionality. You also need to make sure that the changes adhere to the requirements of the application that the application block is to be integrated with.

Whether the application block is customized or not, the integration of the application block with the application needs to be tested to ensure that the application block handles all input from the application and meets all performance and security objectives.

For the purpose of illustration, this chapter refers to a fictitious application and an existing application block. The application requires customization of the application block before integration.

## Sample Application Block

The application block referred to throughout this chapter is the Configuration Management Application Block (CMAB). This chapter assumes that the CMAB has already been through test iterations and is ready to be used without modification. The CMAB has the following features:

- It provides the functionality to read and store configuration information transparently in persistent storage media. The storage media are SQL Server, the registry, and an XML file.
- It provides a configurable option to store the information in encrypted form and plain text using XML notation.
- It can be used with desktop applications and Web applications that are deployed in a Web farm.
- It caches configuration information in memory to reduce cross-process communication such as reading from any persistent medium. This caching reduces the response time of the request for any configuration information. The expiration and scavenging mechanism for the data that is cached in memory is similar to the CRON algorithm in UNIX.
- It can store and return data from various locales and cultures without any loss of data integrity.

## Sample Business Application

For the purpose of illustration, this chapter uses a Web application that needs to use the CMAB to access configuration-related information. The Web application must customize the CMAB to access configuration data from an Oracle database because the CMAB does not support this type of database off-the-shelf. This section describes the details of the Web application and its use of the CMAB. The business-to-customer Web application is an online bookstore that acts as a liaison between the

publishers and the customers. The business process involved and the players in the business process are shown in Figure 9.1.

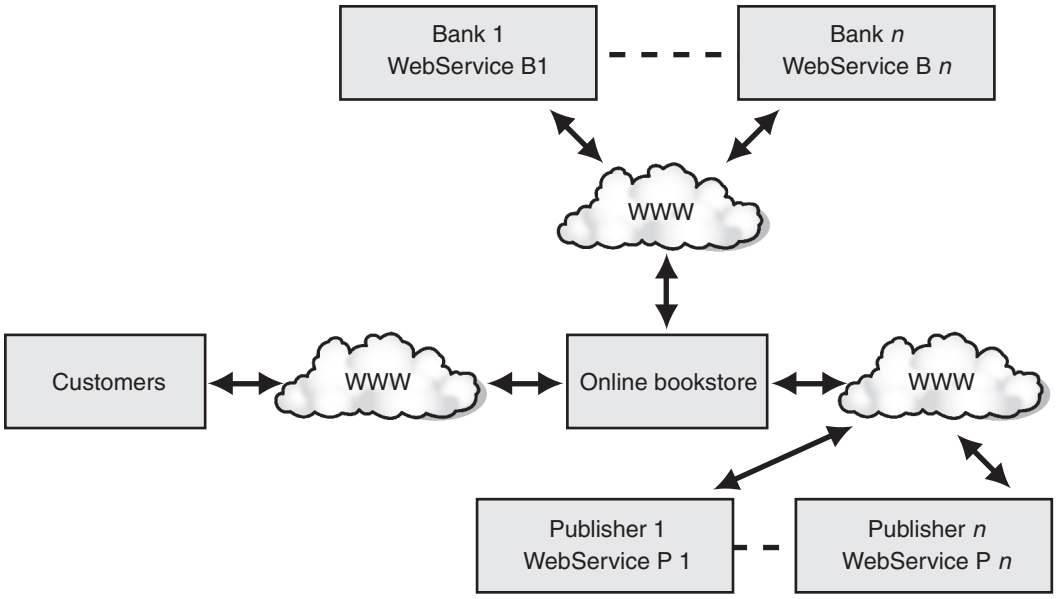


**Figure 9.1**
*Online bookstore sample application*

The sample application lists the book titles, which are categorized according to the publisher or the type of book (such as fiction or nonfiction). The user can use either the Browse or the Search feature to search for a particular title. Based on the information the user sees, the user can buy a soft copy (e-book) or a hard copy (print edition) of the book. The user must be a registered user to buy a book. To complete the purchase, the online bookstore contacts the publishers and the banks through their respective Web services. The online bookstore currently targets U.S.-based customers only, but it needs globalization support so that it can extend its target audience in the future.

The use cases for the online bookstore sample application integrated with the CMAB are as follows:

- **Registering with the Web site.** Users must enter their profile information when registering. The profile information is stored in a database.
- **Personalization.** Users can personalize the Web page templates and the content to be displayed. Users can personalize a variety of information, as follows:
  - Users can choose menu color, style, and positions.
  - Users can choose the number of titles listed per page.

  - Users can choose information to be listed with each title, such as author, an abstract of the book, and ISBN.
  - Users can choose how books are categorized, such as by publisher or subject.
  - Users can choose to list titles from particular publishers only.
  - Users can choose the method of payment.
- **Browse.** When a user logs on and browses through the listed titles, the titles and other information are displayed based on the user's personalized information. Advertisements for new titles are displayed based on the historical information for each user.
- **Search.** The advertisements for new titles are displayed based on the title that the user searches for.
- **Order the soft copy.** When the user buys a soft copy of a book, after the purchase is made, an event notification is sent to the Web services for the publisher and the bank.
- **Order the hard copy.** When the user orders a hard copy of the book from the publisher, after making the payment, an event notification is sent to the Web services for the publisher and the bank.

The online bookstore sample application has a considerable amount of information to be configured. This information is managed transparently using the CMAB. The configuration information can be categorized into user-based configuration information and administrator-controlled configuration information.

The user-based configuration information consists of the following:

- **User interface personalization information.** The user interface personalization information consists of the template for the Web pages and the content to be displayed. This information is stored in the "Personalization" use case and is retrieved in all other use cases through the CMAB.
- **Usage pattern-based personalized information.** The historical usage pattern information is stored during execution of the "Browse" and "Order..." use cases, and is retrieved during the browsing or searching scenarios through the CMAB. This information is used to display news and advertisements for new titles.

The administrator-controlled configuration information consists of the following:

- **URLs of the Web services.** The URLs of the Web services for the publisher and the bank are stored in a persistent medium and are retrieved using the CMAB in the use cases described earlier.
- **Database connection strings.** A database connection string is used to store and retrieve user profile information from the database; therefore, this information is always required. The data is stored in encrypted form.

- **Sinks where information, warnings, and exceptions have to be logged.** The settings for the event sinks are retrieved using the CMAB. Event sinks are used for logging events of different categories (such as information, warnings, exceptions, and so on).
- **Different authorization providers for each bank or payment mode.** The details and settings about the authorization providers are retrieved from the configuration data store — using the CMAB — when a payment is made by the user either by providing the bank account details or credit card details in the "Order..." use cases.
- **Business event settings that generate notification events.** These are the details and the settings of the business events that generate notifications that are sent to the Web services when buying or ordering actions are complete.

The preceding configuration information is managed (stored, retrieved, and cached) using the CMAB. The data store to be used for storing configuration data for the sample application is an Oracle database. Because the CMAB is not packaged with the support for reading and writing data to and from an Oracle data store, you must add this functionality to the CMAB by implementing the exposed interfaces. You can add the functionality for reading and writing data to and from an Oracle data store by adding a customized custom storage provider assembly to the CMAB. You then need to unit test the customized CMAB to ensure that the changes function properly.

## Unit Testing a Customized Application Block

Application blocks may need to be customized based on the requirements of the application. After the application block is customized, it must be unit tested to make sure that the customizations do not break the functionality of existing features. Unit testing also ensures that the customizations are in accordance with the requirements of the application that the block is to be integrated with.

For example, in the case of the online bookstore sample application, the CMAB is required to read and write configuration data to and from an Oracle data store. Therefore, the CMAB has to be customized to manage configuration data stored in an Oracle database. The unit testing is based on the process described in Chapter 3, "Testing Process for Application Blocks."

The following are required for planning the unit tests for the customized application block:

- Requirements and functional specifications for the CMAB and the customizations to be made to the CMAB.
- Performance targets for the sample application use cases using the CMAB.
- Deployment architecture for the sample application.

The unit testing process for the CMAB includes the following steps:

1. Create test plans.
2. Review the design.
3. Review the implementation.
4. Perform black box testing.
5. Perform white box testing.
6. Regression test the existing functionality.

The next sections describe each of these steps.

## Step 1: Create Test Plans

The test plan and the test cases include all scenarios that involve the custom storage provider for Oracle. The test plan and test cases should include cases related to design review, code review, white box testing, load testing, stress testing, globalization testing, and security testing. Table 9.1 shows a sample test plan, and Table 9.2 shows a sample test case. The steps in Table 9.2 summarize the test cases for various aspects for which test plans need to be created.

**Table 9.1: Sample Test Plan from the Detailed Test Plan Document for the CMAB**

| | | |
|---|---|---|
| Scenario 1 | | Use the NUnit test suite to test reading and writing data to an Oracle data store through the custom storage provider for the Oracle data store. |
| Priority | | High |
| Comments | | |
| 1.1 | High | Use the NUnit test suite to test reading and writing data to an Oracle data store through the custom storage provider for the Oracle data store. |

**Table 9.2: Sample Test Case from the Detailed Test Case Document for the CMAB**

| | |
|---|---|
| Serial number | 1.1 |
| Priority | High |
| Scenario to be tested | Use the NUnit test suite to test reading and writing data to an Oracle database using the custom storage provider. |
| Execution details | 1. Create an assembly named **ConfigurationManagement.UnitTests**.<br>2. Create the **OraceConfigurationManagement.cs** file with the **OracleConfigurationManagement** class in the assembly that you created in the preceding step. (Note that the static public methods of the **ConfigurationManager** will be used for the read and write operations.)<br>3. In the **OracleConfigurationManagement** class, implement methods that perform the following functions:<br>● Implement a method that reads a configuration section from the Oracle database for all different combinations of settings for encryption and signing.<br>● Implement a method that writes a configuration section to the Oracle database for all the different combinations of settings for encryption and signing.<br>4. Compile the assembly that you created.<br>5. Run NUnit on this assembly. |
| Data required | Not applicable. |
| Expected results | Not applicable. |
| Actual results | Not applicable. |
| Test passed | Not applicable. |

## Step 2: Review the Design

Review the design of the custom storage provider for Oracle to ensure that it addresses all functional requirements. There are various other important review criteria that you should verify during the review. For more information about these criteria, see Chapter 4, "Design Review for Application Blocks." Table 9.3 on the next page summarizes the test cases for the design review of the custom storage provider for an Oracle database.

**Table 9.3: Sample Design Review – related Test Cases from the Detailed Test Plan Document for the CMAB**

| | | |
|---|---|---|
| Scenario 2 | | Conduct the design review of the custom storage provider proposed for the application block. |
| Priority | | High |
| Comments | | |
| 2.1 | High | Verify that the design addresses all functional specifications and the requirements for the Oracle custom storage provider. |
| 2.2 | High | Verify that the design suites the deployment scenario for the online bookstore sample application. |
| 2.3 | High | Verify that the design makes appropriate tradeoffs based on the best practices for performance and scalability. |
| 2.4 | High | Conduct threat analysis to verify that the design does not create any security vulnerabilities in the application. |
| 2.5 | Medium | Verify that the design follows best practices related to ease of maintenance. |
| 2.6 | Medium | Verify that the design adheres to the globalization-related best practices to ensure smooth localization of the application block. |
| 2.7 | High | Verify that the design has an exception management strategy that ensures proper propagation of exceptions without repeatedly throwing exceptions. |

## Step 3: Review the Implementation

Review the code for the functionality implemented for the Oracle custom storage provider. While reviewing the code, check whether naming standards have been followed, comments are consistent with the other application block modules, loops and conditional statements are optimized for better performance, security requirements are met when connecting to an Oracle database, and exception handling is consistent with the other storage provider modules. The review criteria are based on the processes described in Chapter 5, "Code Review for Application Blocks." Table 9.4 summarizes the test cases for code review of the implementation of the custom storage provider for an Oracle database.

**Table 9.4: Sample Code Review – related Test Cases from the Detailed Test Plan Document for the CMAB**

| | | |
|---|---|---|
| Scenario 3 | | Conduct a code review of the custom storage provider for an Oracle database. |
| Priority | | High |
| Comments | | |
| 3.1 | High | Ensure that the implementation addresses all functional specifications and the requirements for the Oracle custom storage provider. |
| 3.2 | Medium | Verify that naming standards are followed. |
| 3.3 | Medium | Verify that commenting guidelines are followed. |
| 3.4 | High | Verify that the performance and scalability guidelines are followed. |
| 3.5 | High | Verify that the guidelines for writing secure code are followed. |
| 3.6 | High | Verify that globalization-related guidelines are followed. |
| 3.7 | High | Run the FxCop code review tool to ensure that all guidelines supported by the tool are followed in the code. |
| 3.8 | High | Verify that the extended functionality follows the exception management strategy in the CMAB. |

## Step 4: Perform Black Box Testing

The black box testing ensures that the API exposed by the custom storage provider for an Oracle database consistently provides the expected output for various types of application-specific input. You need to develop a custom test suite, such as the NUnit suite, or dummy applications to test the module for various types of input. The input could include data such as the user's personalization information, the URLs that need to be retrieved for various Web services, or the business event settings.

The custom storage provider should also be load tested to verify that it is able to handle normal and peak loads as defined in the performance objectives of the sample application. You need to conduct a series of performance tests to ensure that there are no memory leaks and no contentions when the application block is load tested for long durations.

There are other steps that need to be executed during the black box testing phase of the application block. For more information about the additional steps, see Chapter 6, "Black Box and White Box Testing for Application Blocks." Table 9.5 on the next page summarizes the test cases for black box testing of the custom storage provider for an Oracle database.

**Table 9.5: Sample Black Box Test – related Test Cases from the Detailed Test Plan Document for the CMAB**

| | | |
|---|---|---|
| Scenario 4 | | Conduct black box testing of the custom storage provider for an Oracle database. |
| Priority High | | |
| Comments | | Not applicable. |
| 4.1 | High | Create NUnit tests to test the external interfaces of the custom storage provider for an Oracle database. |
| 4.2 | High | Test the external interfaces for various types of input that are within the specified range of expected input for an application that integrates the application block. |
| 4.3 | High | Test the external interfaces for various types of input that are at the boundary of the specified range of expected input for an application that integrates the application block. |
| 4.4 | High | Test the external interfaces for various types of input that are outside the specified range of expected input for an application that integrates the application block. This input may also be categorized as invalid input. |
| 4.5 | High | Conduct load tests of the custom storage provider to ensure that it meets the performance targets for the application block. |
| 4.6 | High | Conduct load tests for an extended duration to ensure that there are no memory leaks. |
| 4.7 | High | Conduct stress tests by reading and writing large amounts of data on a per-user basis to analyze bandwidth utilization and application block behavior under high loads. |
| 4.8 | High | Conduct load tests for targeted peak operating capacity to ensure that there is no contention. |
| 4.9 | High | Conduct stress tests to analyze the behavior of the custom storage provider beyond the targeted peak operating capacity. |

## Step 5: Perform White Box Testing

White box testing involves analyzing the code and identifying the internal functions, such as a particular code path that needs to be tested for various inputs. White box testing also involves profiling the code from various perspectives. Table 9.6 summarizes the test cases for white box testing of the first iteration of a custom storage provider for an Oracle database. The table may be updated based on the inputs from code review and black box testing phases, during which there may be evidence of code that needs to be further analyzed and tested. For detailed information about white box testing, see Chapter 6, "Black Box and White Box Testing for Application Blocks."

**Table 9.6: Sample White Box Test – related Test Cases from the Detailed Test Plan Document for the CMAB**

| | | |
|---|---|---|
| Scenario 5 | | Conduct white box testing of the custom storage provider for an Oracle database. |
| Priority | | High |
| Comments | | |
| 5.1 | High | Run a code coverage tool to ensure that there is no redundant code in the custom storage provider, and that all possible paths in the code can be traversed under various usage scenarios. |
| 5.2 | High | Analyze the memory allocation pattern of the custom storage provider to ensure that there are no side effect allocations, and that the number of collections for Gen 0, 1, 2 is near the ideal ratio of 1:0.1:0.01. |
| 5.3 | High | Test internal functions for various types of inputs to ensure that there is no loss of data submitted by the user, and that the results are as expected. |
| 5.4 | High | Perform security testing by simulating hostile deployment environments after analyzing the code access security. |

## Step 6: Regression Test the Existing Functionality

After you ensure that the newly added functionality is working as expected and has passed the test criteria, you should regress the existing functionality to ensure that the modifications have not broken the existing code. You can do this by running the test suites created for the application block. These NUnit tests were also used during the development of the application block.

The test case from the detailed test plan document that corresponds to regression testing is shown in Table 9.7.

**Table 9.7: Sample Regression Test – related Test Cases from the Detailed Test Plan Document for the CMAB**

| | | |
|---|---|---|
| Scenario 6 | | Run the existing NUnit tests for the application block to ensure that the existing functionality is working as expected. |
| Priority | | Medium |
| Comments | | |
| 6.1 | Medium | Run the existing NUnit tests for the application block to ensure that the existing functionality is working as expected. |

After the custom storage provider for Oracle passes the unit testing phase, it is integrated with the sample application. At that point, integration testing is done to ensure that the CMAB addresses all of the requirements of the application.

### More Information

For more information about unit testing a customized application block, see the following resources:

- For more information about how to conduct design review for the application block, see Chapter 4, "Design Review for Application Blocks."
- For more information about how to conduct code review for the application block, see Chapter 5, "Code Review for Application Blocks."
- For more information about how to perform functionality testing for the application block, see Chapter 6, "Black Box and White Box Testing for Application Blocks."
- For more information about how to perform performance testing for the application block, see Chapter 8, "Performance Testing for Application Blocks."
- For more information about how to perform globalization testing for the application block, see Chapter 7, "Globalization Testing for Application Blocks."

## Integration Testing an Application Block

Integration testing is the logical extension of unit testing. You may be using the application block as packaged, or you may have customized it before integrating it in your application. In either case, you need to conduct integration testing to ensure that the application block can meet the application-specific requirements. Integration testing ensures the following:

- The interfaces between the integrated units of the application and the application block are able to interact in accordance with the specifications.
- The modules of the application with which the application block has been integrated meet all of the functional requirements, performance objectives, globalization objectives, security objectives, and so on.

Integration testing is important because a piece of code that functions correctly when it is tested as a separate unit can demonstrate problems when it is integrated into the actual application. For example, an incompatible data type could be provided by the application to the application block, or data storage could be locked by some part of the application. In the worst scenario, you may discover issues such as the application block being unable to meet the performance objectives of the application or a loss of data at particular times, such as when the block passes decimals with accuracy to 10 digits of the decimal.

However, in most situations you will discover issues such as improper invocation of the interfaces exposed by the application block, missing error handling, failure to trap all of the exceptions thrown by the application block, incorrect data in configuration files, or mismatch of data types leading to loss of data. The majority of the errors that are discovered during the integration phase of the application block occur at the interface between the application block and the application.

Note that the integration testing of the application block is part of the overall testing process for the application itself. The application may also be integrating the other modules, and the following process is not a substitute for any other kind of testing that needs to be done for the application. The following process focuses on testing the integration of the application block with the application.

For the purposes of illustration, the CMAB is assumed to be integrated with the sample application, which requires configuration data to be stored in an Oracle database. Therefore, a custom storage provider has to be added to the CMAB for reading and writing configuration data from and to an Oracle database. The sample application allows users to personalize the Web site and order a book online.

## Input for Integration Testing

The following are required for integration testing of an application block:

- Functional specifications of the online bookstore sample application and application block
- Requirements for the sample application and application block
- Performance objectives for the sample application
- Deployment scenarios for the sample application

## Steps for Integration Testing

The integration testing process for an application block is shown in Figure 9.2.

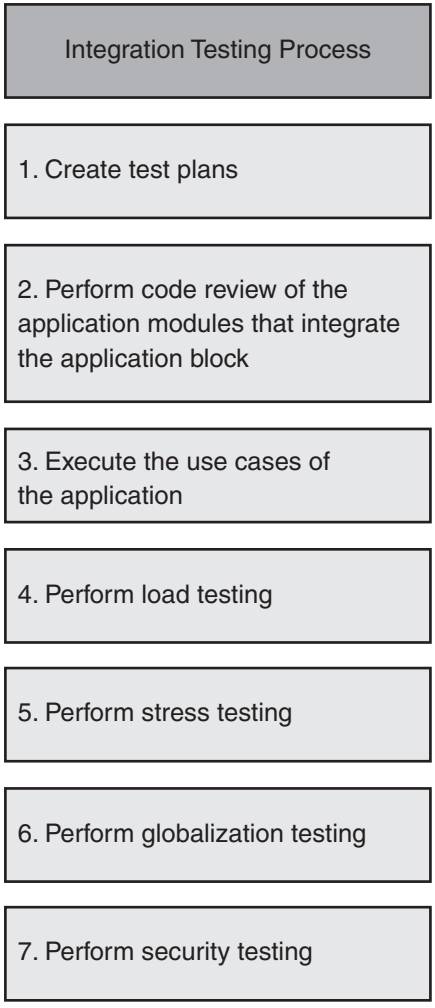


**Figure 9.2**
*The integration testing process*

The integration testing process includes the following steps:

1. **Create test plans.** Create the test plans for integration testing of the application block with the application.
2. **Perform a code review of the application modules that integrate the application block.** The code review helps to ensure that the application code correctly invokes the application block's API.

3. **Execute the use cases of the application.** These are the use cases that consume the application block during their execution.
4. **Perform load testing.** Execute load testing – related test cases for the use cases that integrate the application block.
5. **Perform stress testing.** Execute stress testing – related test cases for the use cases that integrate the application block.
6. **Perform globalization testing.** Execute globalization testing – related test cases for the use cases that integrate the application block.
7. **Perform security testing.** Execute security testing – related test cases for the use cases that integrate the application block.

## Step 1: Create Test Plans

You can create the detailed test plan and the detailed test case documents for testing the integration of the application block with the application even before the actual integration of the application block is completed. These documents are based on the functional specifications of the application modules that will use the application block. The detailed test plan needs to consider all of the possible use cases, the exceptional flows of the use cases, and testing of use cases with various input types (both valid and invalid input). The detailed test plan also includes test cases for ensuring that the integrated modules satisfy the requirements related to performance, security, globalization, and so on.

For the online bookstore sample application, the detailed test plan and detailed test case should consider all scenarios that involve the CMAB; modules that the sample application uses to store and retrieve the configuration information, such as the personalized user information; possible input to the CMAB that includes input from different locales; and possible return values from the CMAB to the sample application, such as the menu information or URLs of the Web services.

## Step 2: Perform Code Review of the Application Modules That Integrate the Application Block

You should perform the code review of the modules that integrate the application block to ensure the following:

- The configuration settings in the configuration files, if any, are correct and in accordance with the requirements.
- The interfaces of the application block are invoked as intended with the expected data types.
- All of the specific exceptions that are thrown by the application block are properly captured and acted upon in the application. This may include logging them to specific sinks or showing appropriately formatted messages to the users. The

application block may use the inner exception object for additional information about the exception; the code review should ensure that, where applicable, the information from the inner exception object is also captured.

Note that the scope of the code review is limited to usage of the application block. A thorough code review of the application modules is part of the testing process for the application itself.

Table 9.8 summarizes the test cases for the code review of the online bookstore sample application modules that use the CMAB.

**Table 9.8: Sample Integration Test-related Test Cases from the Detailed Test Plan Document for the Online Bookstore Sample Application**

| | | |
|---|---|---|
| Scenario 1 | | Perform a code review of the application modules that integrate the application block. The details of the criteria used for the review will be documented in the detailed test case document. |
| Priority | | Medium |
| Comments | | |
| 1.1 | Medium | Perform code review of the "Registering with the Web Site" use case. |
| 1.2 | Medium | Perform code review of the "Personalization" use case. |
| 1.3 | Medium | Perform code review of the "Browse" use case. |
| 1.4 | Medium | Perform code review of the "Search" use case. |
| 1.5 | Medium | Perform code review of the "Order the Soft Copy" use case. |
| 1.6 | Medium | Perform code review of the "Order the Hard Copy" use case. |

## Step 3: Execute the Use Cases of the Application

After implementing the code required for application block integration, you need to test the functionality of the use cases related to the application block. The functional testing involves the following:

1. Test the normal flow of the use cases, which includes all possible combinations that each use case can be executed in. This involves performing various user actions by providing the expected input within the specified range and may include the boundary input.
2. Test the exception flow by providing invalid input or conditions that generate exceptions, such as shutting down the Oracle database server during the use case execution or providing incorrect configuration details in the configuration file.

These are the essential test cases related to the testing of application-specific functionality. You will always have these test cases, irrespective of whether or not you are integrating a module with an application. These test cases ensure that the application is functioning as expected.

For the sample application, you must test the use cases related to the CMAB. The test cases are summarized in Table 9.9. The detailed test cases for each test case from Table 9.9 are summarized in Table 9.19 in the "Appendix" section of this chapter.

You may need to make temporary changes to the application module code to simulate invalid input or exceptional conditions. In most scenarios, you will not need to develop test suites because all input will be provided through the application by user actions or background processes.

**Table 9.9: Sample Functional Test – related Test Cases from the Detailed Test Plan Document for the Online Bookstore Sample Application**

| | | |
|---|---|---|
| Scenario 2 | | Test the functionality of the use cases of the online bookstore sample application, which integrates the CMAB. |
| Priority | | High |
| Comments | | |
| 2.1 | High | Test the "Registering with the Web Site" use case. |
| 2.2 | High | Test the "Personalization" use case for personalization of Web pages and the display content. |
| 2.3 | High | Test the "Browse" use case to browse through titles based on the personalized information. |
| 2.4 | High | Test the "Search" use case to search for particular titles. |
| 2.5 | High | Test the "Order the Hard Copy" use case for ordering a hard copy of a book from the publisher. |
| 2.6 | High | Test the "Order the Soft Copy" use case for ordering a soft copy of a book from the publisher. |

Table 9.10 on the next page summarizes the expected results for execution of each test case from Table 9.19 (listed in the "Appendix" section later in this chapter). You should note that Table 9.19 lists only one of the possible flow paths for the use case. If the use case involves any alternate flow paths, you should test all of the flow paths with different combinations of input data.

**Table 9.10: Sample Expected Results for the Online Bookstore Sample Application**

| Serial number | Scenario to be tested | Expected results | Actual results | Test passed |
|---|---|---|---|---|
| 2.1 | Registering with the Web Site | The user should be able to register without errors and the profile should be stored in the database, defined by the connection string that has been stored as administrative configuration information in the Oracle data store. | | |
| 2.2 | Personalization | – The user should be able to personalize the Web page style and content to be displayed, and the personalization information should be stored in the Oracle data store.<br>– When the user logs on the next time, the profile information should be retrieved, and customized content should be displayed on the Web page. The information is retrieved from the database based on the connection string that is part of the administrative configuration data.<br>– The warning, information, and exception events should be logged according to the settings in the configuration information that is retrieved through the CMAB from the Oracle data store. | | |
| 2.5 | Order the Hard Copy | – The user should be able to order a hard copy from the publisher.<br>– Profile information should be retrieved from the database based on the connection string retrieved as part of the administrative configuration data through the CMAB from the Oracle data store.<br>– The appropriate publisher Web services should be contacted based on the URLs retrieved from the configuration data stored through the CMAB in the Oracle data store.<br>– The warning, information, and audit and exception events should be logged according to the settings in the configuration information that is retrieved through the CMAB from the Oracle data store.<br>– The payment authorization should be done according to the settings in the configuration information that is retrieved through the CMAB from the Oracle data store.<br>– Event notification should be published according to the settings in the configuration information that is retrieved through the CMAB from the Oracle data store. | | |

## Step 3: Perform Load Testing

Load testing of the integrated application is required to analyze its behavior at various load levels. Load testing helps to verify that the application meets the performance objectives required for the application after integrating the application block. Because of the integration, there may be bottlenecks in the application — such as resource contention with other modules —that could reduce application performance. Load testing helps to identify such bottlenecks that usually surface only when the application is subjected to a load of concurrent users.

The test cases are the same test cases that you would execute for the application, whether or not you are integrating an application block.

The steps to perform load testing follow the load testing process described in Chapter 16, "Testing .NET Application Performance" of *Improving .NET Application Performance and Scalability* on MSDN at:
*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/scalenetchapt16.asp.*

The steps to perform load testing are the following:

1. Identify the test scenarios.
2. Identify the workload.
3. Identify the metrics.
4. Create the test cases.
5. Simulate the load.
6. Analyze the results.

The next sections describe each of these steps as they apply to load testing.

### Identify the Test Scenarios

The inputs for this step can be obtained from the following sources:

- Functional specifications or usage scenarios of the online bookstore sample application and application block
- Requirements for the sample application and application block
- Performance objectives for the sample application
- Performance model

The selected usage scenarios of the application are ones that integrate and use application block.

Consider the online bookstore sample application. The CMAB has been integrated to manage the configuration information for this application. The sample usage scenarios are listed in Table 9.19 in the "Appendix" section later in this chapter.

## Identify the Workload

The second step in the process of load testing is the identification of workload. The input for the workload identification is taken from the performance model, marketing data, requirements, and the test scenarios identified in the preceding step. The key components of a workload model for the application are as follows:

- **The maximum expected number of simultaneous users logged on to the application.** These users have active connections to the Web site; however, all of them may not be sending requests at any given time. The excepted load for the online bookstore sample application is 1000 simultaneous users.
- **Various user profiles.** In a real-world environment, a user might log on, personalize a Web page, search or browse for items, and then purchase the books. The important point is that each user does multiple operations. Therefore, it is important to have various user profiles and to find out the appropriate distribution of the maximum load among these user profiles. Each user profile may have one or more dominant use cases that the user of that profile most frequently requests, as compared to others.

  In the case of the online bookstore sample application, the typical operations are as follows:

  - Log on
  - Register
  - Personalize
  - Browse
  - Search
  - Buy soft copy
  - Buy hard copy

  Table 9.11 shows the operations that constitute a Browse user profile. Ideally you would study the existing profiles of similar existing applications or conduct market research to develop suitable user profiles for your application.

**Table 9.11: Sample Browse User Profile for Online Bookstore Sample Application**

| Operation | Number of times performed in a single session |
|---|---|
| Log on use case | 1 |
| Browse use case | 6 |
| Search use case | 2 |

Profiles for the sample application are the following:

- Register profile
- Personalize profile
- Browse profile

- Search profile
- Buy Soft Copy profile
- Buy Hard Copy profile

At any given time, a Web application has users from various profiles logging on. Therefore, the workload for an application consists of a user profile mix, with each profile constituting a percentage of the total users logged on. Table 9.12 shows the user profile mix for the sample application.

**Table 9.12: Sample User Profile Mix for Online Bookstore Sample Application**

| User profile | Percentage | Simultaneous users |
|---|---|---|
| Register | 5 | 50 |
| Personalize | 10 | 100 |
| Browse | 35 | 350 |
| Search | 25 | 250 |
| Buy Soft Copy | 15 | 150 |
| Buy Hard Copy | 10 | 100 |
| Total | 100 | 1000 |

- **Think time between requests**. Think time is the time that a user spends between two consecutive requests. You should estimate the think time based on the complexity of the Web page. The think time needs to be higher if the user needs to fill in personal details, as is the case with "Registering with the Web site" use case. Based on the complexity, calculate the average think time for each of the requested pages. The think time can be anywhere from 5 seconds to more than a minute.

## Identify the Metrics

The third step in the process of load testing is the identification of metrics. Metrics are used to determine quantitatively whether the performance goals specified in the requirements are being met and are also used to identify any bottlenecks affecting the performance and scalability of the application. Metrics related to performance goals may be available from System Monitor, or you may need to instrument the code to capture custom metrics. For example, the online bookstore sample application has performance goals for the following, which require custom instrumentation of the code:

- Average number of read operations from the Oracle data store per second
- Average number of write operations to the Oracle data store per second
- Average time for a title search to return results
- Average time for a user to log on
- Average time to submit the personalized information

Metrics in the initial iterations generally give a coarse-grained picture of the overall behavior of application. During subsequent iterations, you can add more metrics to focus on a specific area for a fine-grained picture. For example, if you see an increase in private bytes during testing, you can add counters related to various managed heaps.

In the case of the online bookstore sample application, where hundreds of concurrent users store and read personalized information, browse and search product catalogs, and so on, you need to monitor the disk-related counters because there is a possibility of the disk becoming a bottleneck on the server hosting the Oracle data store.

You should monitor a set of counters (shown in Table 9.13) related to resource utilization on all of the servers. These counters help you to analyze the overall resource utilization of the servers and to determine whether any of the resources — such as processor, memory, disk, or network I/O — are a bottleneck.

**Table 9.13: Base Set of Metrics for All Servers in the Deployment**

| Object | Counter | Instance |
|---|---|---|
| Processor | % Processor Time | _Total |
| Processor | % Interrupt Time | _ Total |
| Processor | % Privileged Time | _ Total |
| System | Processor Queue Length | Not applicable |
| System | Context Switches/sec | Not applicable |
| Memory | Available Mbytes | Not applicable |
| Memory | Pages/sec | Not applicable |
| Memory | Cache Faults/sec | Not applicable |
| Physical Disk | Avg. Disk Queue Length | (Disk instance) |
| Physical Disk | Avg. Disk Read Queue Length | (Disk instance) |
| Physical Disk | Avg. Disk sec/Read | (Disk instance) |
| Physical Disk | Avg Disk sec/Write | (Disk instance) |
| Server | Pool Nonpaged Failures | Not applicable |
| Process | Page Faults/sec | Total |
| Process | Working Set | (Monitored process) |
| Process | Private Bytes | (Monitored process) |
| Process | Handle Count | (Monitored process) |

Table 9.14 lists the counters that need to be captured during the first iteration of the load testing. You can add more counters based on the analysis in subsequent iterations.

**Table 9.14: Metrics to Be Monitored on the Web Tier**

| Object | Counter | Instance |
|---|---|---|
| ASP.NET | Requests Current | Not applicable |
| ASP.NET | Requests Queued | Not applicable |
| ASP.NET | Requests Reject | Not applicable |
| ASP.NET | Request Execution Time | Not applicable |
| ASP.NET | Request Wait Time | Not applicable |
| ASP.NET Applications | Requests/Sec | Your virtual directory |
| ASP.NET Applications | Requests Executing | Your virtual directory |
| ASP.NET Applications | Requests in Application Queue | Your virtual directory |
| ASP.NET Applications | Requests Timed Out | Your virtual directory |
| ASP.NET Applications | Cache Total Hit Ratio | Your virtual directory |
| ASP.NET Applications | Cache API Hit Ratio | Your virtual directory |
| ASP.NET Applications | Output Cache Hit Ratio | Your virtual directory |
| ASP.NET Applications | Errors Total/sec | Your virtual directory |
| ASP.NET Applications | Pipeline Instance Count | Your virtual directory |
| .NET CLR Memory | % Time in GC | Aspnet_wp |
| .NET CLR Memory | # Bytes in All Heaps | Aspnet_wp |
| .NET CLR Memory | # of Pinned Objects | Aspnet_wp |
| .NET CLR Memory | Large Object Heap Size | Aspnet_wp |
| .NET CLR Exceptions | # of Exceps Thrown/sec | Aspnet_wp |
| .NET CLR LocksAndThreads | Contention Rate/sec | Aspnet_wp |
| .NET CLR LocksAndThreads | Current Queue Length | Aspnet_wp |
| .NET CLR Data | SqlClient: Current # connection pools | Not applicable |
| .NET CLR Data | SqlClient: Current # pooled connections | Not applicable |
| Web Service | ISAPI Extension Requests/sec | Your Web site |

## Create the Test Cases

The fourth step in the process of load testing the integrated application is to create the test cases. The test cases are created based on the scenarios identified in Step 1 and workload profiles identified in Step 2. You can create test cases for incremental load testing by increasing the load for a given workload profile until the metrics hit their performance threshold, or you can create a test case for analyzing performance for a specific load of users. Table 9.15 shows a test case with a workload profile and a load of 1,000 simultaneous users.

**Table 9.15: Sample Test Case for the Online Bookstore Sample Application**

| Workload profile | User profile | Percentage | Simultaneous users |
|---|---|---|---|
| | Register | 5 | 50 |
| | Personalize | 10 | 100 |
| | Browse | 35 | 350 |
| | Search | 25 | 250 |
| | Buy Soft Copy | 15 | 150 |
| | Buy Hard Copy | 10 | 100 |
| | Total | 100 | 1000 |
| **Test duration** | 60 minutes | | |
| **Think time** | Random think time of 1 – 10 seconds | | |
| **Expected results** | | | |
| **Throughput** | 100 requests per second (**ASP.NET\Requests/sec** performance counter) | | |
| **Processor\%Processor** | 75 percent | | |
| **Memory\Available Mbytes** | 25 percent of total RAM | | |
| **Average response time** | <3 seconds (on 100 megabits per second [Mbps] LAN) | | |

## Simulate the Load

You can simulate the load by using load simulation tools such as Microsoft Application Center Test (ACT). You should make sure the load generator does not stress the resources (such as processor or memory) on the client computer.

### Analyze the Results

The final step in load testing the application is analyzing results. The metrics are analyzed for any bottlenecks that are hurting application performance and to determine whether the application performance is moving toward or away from the performance objectives.

If you encounter any bottlenecks, you need to identify the cause by either conducting a code review of the module or profiling the code using tools from the debugging toolkit for the Microsoft® Windows® operating system.

If the application block is the cause of the bottleneck, you need to unit test and fix the issues in the application block, and then continue with integration testing.

## Step 4: Perform Stress Testing

The integrated application needs to be stress tested to evaluate application behavior at loads greater than the peak operating capacity while the resources to process the load are being denied. This type of testing reveals issues that arise only under abnormal loads, such as synchronization issues, loss of data during network congestion, memory leaks, and so on.

To stress test the integrated application block, you need to identify scenarios, workload profiles, metrics, and test cases. You should use input from load testing, such as the test scenarios, peak operating capacities, workload allocations, and so on, as well as code review results to formulate the test cases for stress testing.

The steps for stress testing are as follows:

1. Identify the key scenarios.
2. Identify the workloads.
3. Identify the metrics.
4. Create the test cases.
5. Simulate the load.
6. Analyze the results.

The next sections describe each of these steps in more detail.

### Identify the Key Scenarios

The first step in stress testing is to identify the scenarios that are critical to performance of the overall application. This includes operations such as locking, synchronized access to a resource, or extensive disk or network I/O. You can identify these scenarios based on input from load testing. Usually, such scenarios are resource intensive or occur frequently and affect overall performance directly or indirectly.

For example, in the online bookstore sample application, each usage scenario requires different configuration information. If caching is not enabled for the CMAB, the CMAB must repeatedly access the Oracle data store to read and write configuration information. When the number of users increases to very high loads, the number of CMAB requests to the Oracle database also increases significantly. This increase can lead to excessive disk I/O at the database server, which in turn leads to queue buildup at the database server and an increase in response time. The requests on the Web server continue to wait for a response from the database server, and hold on to the threads from the thread pool. This can cause the requests to time out, and the user may receive "time out" or "server too busy" errors.

Another possible scenario deals with enabling caching for the CMAB, and then caching a large amount of data on the Web server. This significantly increases memory utilization on the Web server and, in worst case, may cause the worker process to recycle.

Table 9.16 lists other stress testing scenarios from the sample application.

**Table 9.16: Sample Stress Testing Scenarios for the Online Bookstore Sample Application**

| | | |
|---|---|---|
| Scenario 1 | | Conduct stress testing of various use cases of the online bookstore sample application |
| Priority | | High |
| Comments | | |
| 1.1 | High | Test the Personalize profile for a number of users above the peak operating capacity identified in load testing. |
| 1.2 | High | Test the Buy Soft Copy profile for a number of users beyond the peak operating capacity. |
| 1.3 | High | Test the Buy Hard Copy profile for a number of users beyond the peak operating capacity, with a large number of users ordering hard copies of the book from the publishers. |

## Identify the Workloads

The second step in stress testing is to identify workload profiles based on the scenarios identified in the preceding step. You can use the same workload profiles from load testing but increase the load for the dominant use case significantly or stress test the use case in isolation. The input for the workload identification includes the peak operating capacities taken from load testing for a given workload profile.

The workload is usually increased from a value beyond the maximum operating capacity. For example, in the case of the test scenario, if the maximum operating capacity is 1,000 users, then the workload can be varied between 1,100 to 2,000 or more users.

### Identify the Metrics

Metrics are specific to the test scenarios, and you should select them based on the seriousness and likelihood of potential problems.

For example, in the case of stress testing with a workload and storing a large amount of information in the application block cache, you should monitor the memory-related counters for the system, the ASP.NET worker process, and the base set of counters listed in Table 9.13. Table 9.17 lists the performance counters you would most likely want to monitor for this scenario.

**Table 9.17: Metrics to Be Captured for Stress Testing the Online Bookstore Sample Application**

| Object | Counter | Instance |
|---|---|---|
| ASP.NET Applications | Requests/Sec | <Virtual directory> |
| ASP.NET Applications | Requests Executing | Not applicable |
| ASP.NET Applications | Requests Timed Out | Not applicable |
| ASP.NET Applications | Requests in Application Queue | <Virtual directory> |
| ASP.NET | Request Execution Time | Not applicable |
| ASP.NET | Requests Queued | Not applicable |
| ASP.NET | Requests Execution Time | Not applicable |
| ASP.NET | Requests Rejected | Not applicable |
| .NET CLR Memory | # Bytes in All Heaps | Not applicable |
| .NET CLR Memory | # Gen 0 Collections | Not applicable |
| .NET CLR Memory | # Gen 1 Collections | Not applicable |
| .NET CLR Memory | # Gen 2 Collections | Not applicable |
| .NET CLR Memory | % Time in GC | Not applicable |
| .NET CLR Memory | Large Object Heap Size | Not applicable |

### Create the Test Cases

Document the test cases using the workload patterns and scenarios that you identified based on the results of the previous steps.

Table 9.18 shows the results for the sample application.

**Table 9.18: Test Case Results for the Online Bookstore Sample Application**

| | |
|---|---|
| Scenario | A user orders a hard copy of a book from the publisher |
| Number of users | 2000 simultaneous users |
| Test duration | 1 hour |
| Think time | Random think time is 1 – 10 seconds |
| Expected results | – The ASP.NET worker process should not be recycled.<br>– Throughput should not fall below 30 requests per second. (**ASP.NET\Requests/sec** performance counter)<br>– Response Time for the usage scenario should not exceed 7 seconds (on a 100 Mbps per second LAN)<br>– Server busy errors should not exceed 20 percent of the total response.<br>– Memory utilization should not be more than 60 percent of available memory (**Memory\Private Bytes**) |

## Simulate the Load

Simulate the load as specified in the test cases by using load simulation tools such as ACT.

## Analyze the Results

The final step in stress testing the application is to analyze the results. If you identify any potential bottlenecks caused by the application block rather than the application itself, you need to conduct white box testing of the application block by profiling or debugging the code using various tools, such as debugging tool kit.

### More Information

For more information about analyzing the results of stress testing, see the following resources:

- For more information about tools and techniques for debugging, see "Production Debugging for .NET Framework Applications" at: *http://msdn.microsoft.com/library/en-us/dnbda/html/DBGrm.asp.*
- For more information about debugging deadlocks, see "Scenario: Contention or Deadlock Symptoms" in "Production Debugging for .NET Framework Applications" at: *http://msdn.microsoft.com/library/en-us/dnbda/html/DBGch03.asp.*
- For more information about deadlocks, see Microsoft Knowledge Base article 317723, "INFO: What Are Race Conditions and Deadlocks?" at: *http://support.microsoft.com/default.aspx?scid=kb;en-us;317723.*

## Step 5: Perform Globalization Testing

One requirement of the online bookstore sample application is that the application should be easily localizable for Japanese and German locales. This means that the application should be able to accept, process, store, and display Japanese and German characters. The resulting localized application is expected to meet the requirements of the functional specifications and the performance objectives of the application.

The test cases and test scenarios executed as part of functional testing also apply for the functionality testing in globalization. The test cases must be tested for both German and Japanese locales and cultures in the case of the sample application.

You will need to set up the test environment, which may include one or both of the following:

- You may need a target version of the operating system with the appropriate language group installed. The targeted language groups in the case of the sample application are Japanese and German.
- You may need a localized build target of the operating system, for example, the Windows operating system with Japanese as the UI language.

After the environment is set up, you need to execute the application use cases to ensure that the localized application functions as expected. Even though the application block has been tested for globalization in the unit testing phase, it is important to verify that the use cases that integrate the application block work correctly.

### More Information

For more information about globalization testing, see the following resources:

- For more information about step-by-step globalization testing for integrated components, see "Globalization Step-by-Step" on the Microsoft Global Development and Computing Portal at:
  *http://www.microsoft.com/globaldev/getwr/steps/wrguide.mspx.*
- For more information about best practices for globalization, see "Best Practices for Developing World-Ready Applications" in the *.NET Framework Developer's Guide* on MSDN at:
  *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconbestpracticesforglobalapplicationdesign.asp.*

## Step 6: Perform Security Testing

You should test the security of the sample application after it is integrated with the application block. This testing should focus on identifying any new vulnerabilities that were introduced as a result of the integration. This testing can be done at two levels:

- **Black box integration testing.** The focus of this testing should be on the application block integration. You should test for unauthorized access to ensure that the application is secure after integration. This type of testing is done by simulating the production environment, and it attempts to find issues that could arise at run time. The test cases that were created during the application block black box testing phase can be used here by modifying them to use the changed interface. You can also use any test cases developed earlier for the application to ensure that the application is secure.
- **White box integration testing.** The application threat model should be updated to include application block integration. (It is assumed that the sample application already has a threat model in place.) The new threat model should do the following:
  - Update the application architecture diagram with the application block architecture.
  - Identify any new assets, privilege requirements, and resources accessed by the application block.
  - Based on additional information collected earlier, identify any new threats and update the threat model.
  - Prioritize the threats and suggest countermeasures.

Any threats you identify during security testing can be a result of one or both of the following:

- The application block design and/or code did not follow security best practices, such as not supporting encryption if the application block is used for handling sensitive data or not using the best encryption algorithm available to encrypt sensitive data.
- The application block failed to consider specific requirements of the application, such as the application block depending on a service that makes the application vulnerable.

You should also review the integration-related code to ensure that it follows security-related best practices.

### More Information

For a detailed description of threats and countermeasures and the process of threat modeling, see Chapter 3, "Threat Modeling" in *Improving Web Application Security: Threats and Countermeasures* on MSDN at:
*http://msdn.microsoft.com/library/en-us/dnnetsec/html/THCMCh03.asp*.

## Summary

This chapter presented the processes for unit testing the customizations made to the application block and how to test the application block after you integrate it with your application. The process of integration testing assumes that the application block has been rigorously unit tested and meets all requirements of the application, and that the developer has done a careful analysis before deciding to use the application block.

# Appendix

**Table 9.19: Execution Steps for the Online Bookstore Sample Application Uses Cases**

| Test scenario | Steps to follow |
|---|---|
| Registering with the Web site | – Fill out the logon form or the user profile.<br>– Submit the user profile.<br>– Log on.<br>– Log out. |
| Personalizing the Web pages and content to be displayed | – Log on.<br>– Modify or upload the Web page templates.<br>– Modify the information to be displayed.<br>– Modify the number of titles to be listed per page.<br>– Choose whether the titles displayed should be categorized based on the publisher or the subject matter of the books.<br>– Choose the order in which the titles are listed, either in alphabetical order or in the chronological order.<br>– Choose the publishers whose titles should be listed.–Submit the personalized information.<br>– Choose the method of payment and the bank.<br>–Enter the credit card and or bank account details.<br>– Log out. |
| Browsing through the application | – Log on.<br>– Browse the advertisements and new titles that are listed based on the history of a specific user.<br>– Read the abstract and user preface for each new title.<br>– Browse the latest books from each publisher.<br>– Log out. |
| Searching for a title | – Log on.<br>– Search for a book.<br>– Read the abstract and user preface for the selected book.<br>– Log out. |
| Reserving a book | – Log on.<br>– Search for a book.<br>– Select the distributor.<br>– Reserve the book.<br>– Log out. |
| Buying the soft copy of the book | – Log on.<br>– Browse through the latest offerings from a publisher.<br>– Select a book.<br>– Choose the soft copy format of the book.<br>– Certify and submit the payment details.<br>– Download the soft copy.<br>– Log out. |

| Test scenario | Steps to follow |
|---|---|
| Ordering the hard copy of the book from the publisher | – Log on.<br>– Browse through the latest offerings from a publisher.<br>– Select a book.<br>– Choose the hard copy option of the book.<br>– Choose the option to have the publisher send the book.<br>– Certify and submit the payment details.<br>– Log out. |

### About Microsoft *patterns & practices*

Microsoft *patterns & practices* guides contain specific recommendations illustrating how to design, build, deploy, and operate architecturally sound solutions to challenging business and technical scenarios. They offer deep technical guidance based on real-world experience that goes far beyond white papers to help enterprise IT professionals, information workers, and developers quickly deliver sound solutions.

IT Professionals, information workers, and developers can choose from four types of *patterns & practices*:

- **Patterns**—Patterns are a consistent way of documenting solutions to commonly occurring problems. Patterns are available that address specific architecture, design, and implementation problems. Each pattern also has an associated GotDotNet Community.
- **Reference Architectures**—Reference Architectures are IT system-level architectures that address the business requirements, LifeCycle requirements, and technical constraints for commonly occurring scenarios. Reference Architectures focus on planning the architecture of IT systems.
- **Reference Building Blocks and IT Services**—References Building Blocks and IT Services are re-usable sub-system designs that address common technical challenges across a wide range of scenarios. Many include tested reference implementations to accelerate development. Reference Building Blocks and IT Services focus on the design and implementation of sub-systems.
- **Lifecycle Practices**—Lifecycle Practices provide guidance for tasks outside the scope of architecture and design such as deployment and operations in a production environment.

*Patterns & practices* guides are reviewed and approved by Microsoft engineering teams, consultants, Product Support Services, and by partners and customers. *Patterns & practices* guides are:

- **Proven**—They are based on field experience.
- **Authoritative**—They offer the best advice available.
- **Accurate**—They are technically validated and tested.
- **Actionable**—They provide the steps to success.
- **Relevant**—They address real-world problems based on customer scenarios.

*Patterns & practices* guides are designed to help IT professionals, information workers, and developers:

**Reduce project cost**

- Exploit the Microsoft engineering efforts to save time and money on your projects.
- Follow the Microsoft recommendations to lower your project risk and achieve predictable outcomes.

**Increase confidence in solutions**

- Build your solutions on proven Microsoft recommendations so you can have total confidence in your results.
- Rely on thoroughly tested and supported guidance, but production quality recommendations and code, not just samples.

**Deliver strategic IT advantage**

- Solve your problems today and take advantage of future Microsoft technologies with practical advice.

**patterns & practices: Current Titles**

October 2003

| Title | Link to Online Version | Book |
|---|---|---|
| **Patterns** | | |
| Enterprise Solution Patterns using Microsoft .NET | *http://msdn.microsoft.com/practices/type/Patterns/Enterprise/default.asp* |  |
| Microsoft Data Patterns | *http://msdn.microsoft.com/practices/type/Patterns/Data/default.asp* | |
| **Reference Architectures** | | |
| Application Architecture for .NET: Designing Applications and Services | *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/distapp.asp* |  |
| Enterprise Notification Reference Architecture for Exchange 2000 Server | *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnentdevgen/html/enraelp.asp* | |
| Improving Web Application Security: Threats and Countermeasures | *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnetsec/html/ThreatCounter.asp* |  |
| Microsoft Accelerator for Six Sigma | *http://www.microsoft.com/technet/treeview/default.asp?url=/technet/itsolutions/mso/sixsigma/default.asp* | |
| Microsoft Active Directory Branch Office Guide: Volume 1: Planning | *http://www.microsoft.com/technet/treeview/default.asp?url=/technet/prodtechnol/ad/windows2000/deploy/adguide/default.asp* |  |
| Microsoft Active Directory Branch Office Series Volume 2: Deployment and Operations | *http://www.microsoft.com/technet/treeview/default.asp?url=/technet/prodtechnol/ad/windows2000/deploy/adguide/default.asp* |  |
| Microsoft Content Integration Pack for Content Management Server 2001 and SharePoint Portal Server 2001 | *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncip/html/cip.asp* | |
| Microsoft Exchange 2000 Server Hosting Series Volume 1: Planning | Online Version not available |  |
| Microsoft Exchange 2000 Server Hosting Series Volume 2: Deployment | Online Version not available |  |

| Title | Link to Online Version | Book |
|---|---|---|
| Microsoft Exchange 2000 Server Upgrade Series Volume 1: Planning | *http://www.microsoft.com/technet/treeview /default.asp?url=/technet/itsolutions/guide /default.asp* |  |
| Microsoft Exchange 2000 Server Upgrade Series Volume 2: Deployment | http://www.microsoft.com/technet/treeview /default.asp?url=/technet/itsolutions/guide /default.asp |  |
| Microsoft Solution for Intranets | *http://www.microsoft.com/technet/treeview /default.asp?url=/technet/itsolutions/mso /msi/Default.asp* | |
| Microsoft Solution for Securing Wireless LANs | *http://www.microsoft.com/downloads /details.aspx?FamilyId=CDB639B3-010B-47E7-B23 4-A27CDA291DAD&displaylang=en* | |
| Microsoft Systems Architecture— Enterprise Data Center | *http://www.microsoft.com/technet/treeview /default.asp?url=/technet/itsolutions/edc /Default.asp* | |
| Microsoft Systems Architecture— Internet Data Center | *http://www.microsoft.com/technet/treeview/ default.asp?url=/technet/itsolutions/idc/default.asp* | |
| The Enterprise Project Management Solution | *http://www.microsoft.com/technet/treeview /default.asp?url=/technet/itsolutions/mso/epm /default.asp* | |
| UNIX Application Migration Guide | *http://msdn.microsoft.com/library/default.asp?url= /library/en-us/dnucmg/html/ucmglp.asp* |  |
| **Reference Building Blocks and IT Services** | | |
| .NET Data Access Architecture Guide | *http://msdn.microsoft.com/library/default.asp?url= /library/en-us/dnbda/html/daag.asp* | |
| Application Updater Application Block | *http://msdn.microsoft.com/library/default.asp?url= /library/en-us/dnbda/html/updater.asp* | |
| Asynchronous Invocation Application Block | *http://msdn.microsoft.com/library/default.asp?url= /library/en-us/dnpag/html/paiblock.asp* | |
| Authentication in ASP.NET: .NET Security Guidance | *http://msdn.microsoft.com/library/default.asp?url= /library/en-us/dnbda/html/authaspdotnet.asp* | |
| Building Interoperable Web Services: WS-I Basic Profile 1.0 | *http://msdn.microsoft.com/library/default.asp?url= /library/en-us/dnsvcinter/html/wsi-bp_msdn_ landingpage.asp* |  |
| Building Secure ASP.NET Applications: Authentication, Authorization, and Secure Communication | *http://msdn.microsoft.com/library/default.asp?url= /library/en-us/dnnetsec/html/secnetlpMSDN.asp* |  |

| Title | Link to Online Version | Book |
|---|---|---|
| Caching Application Block | *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/Cachingblock.asp* | |
| Caching Architecture Guide for .Net Framework Applications | *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/CachingArch.asp?frame=true* | |
| Configuration Management Application Block | *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/cmab.asp* | |
| Data Access Application Block for .NET | *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/daab-rm.asp* | |
| Designing Application-Managed Authorization | *http://msdn.microsoft.com/library/?url=/library/en-us/dnbda/html/damaz.asp* | |
| Designing Data Tier Components and Passing Data Through Tiers | *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/BOAGag.asp* | |
| Exception Management Application Block for .NET | *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/emab-rm.asp* | |
| Exception Management Architecture Guide | *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/exceptdotnet.asp* | |
| Microsoft .NET/COM Migration and Interoperability | *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/cominterop.asp* | |
| Microsoft Windows Server 2003 Security Guide | *http://www.microsoft.com/downloads/details.aspx?FamilyId=8A2643C1-0685-4D89-B655-521EA6C7B4DB&displaylang=en* | |
| Monitoring in .NET Distributed Application Design | *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/monitordotnet.asp* | |
| New Application Installation using Systems Management Server | *http://www.microsoft.com/business/reducecosts/efficiency/manageability/application.mspx* | |
| Patch Management using Microsoft Systems Management Server - Operations Guide | *http://www.microsoft.com/technet/treeview/default.asp?url=/technet/itsolutions/msm/swdist/pmsms/pmsmsog.asp* | |
| Patch Management Using Microsoft Software Update Services - Operations Guide | *http://www.microsoft.com/technet/treeview/default.asp?url=/technet/itsolutions/msm/swdist/pmsus/pmsusog.asp* | |
| Service Aggregation Application Block | *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/serviceagg.asp* | |
| Service Monitoring and Control using Microsoft Operations Manager | *http://www.microsoft.com/business/reducecosts/efficiency/manageability/monitoring.mspx* | |

| Title | Link to Online Version | Book |
|---|---|---|
| User Interface Process Application Block | *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/uip.asp* | |
| Web Service Façade for Legacy Applications | *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/wsfacadelegacyapp.asp* | |
| **Lifecycle Practices** | | |
| Backup and Restore for Internet Data Center | *http://www.microsoft.com/technet/treeview/default.asp?url=/technet/ittasks/maintain/backuprest/Default.asp* | |
| Deploying .NET Applications: Lifecycle Guide | *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/DALGRoadmap.asp* | |
| Microsoft Exchange 2000 Server Operations Guide | *http://www.microsoft.com/technet/treeview/default.asp?url=/technet/prodtechnol/exchange/exchange2000/maintain/operate/opsguide/default.asp* |  |
| Microsoft SQL Server 2000 High Availability Series: Volume 1: Planning | *http://www.microsoft.com/technet/treeview/default.asp?url=/technet/prodtechnol/sql/deploy/confeat/sqlha/SQLHALP.asp* |  |
| Microsoft SQL Server 2000 High Availability Series: Volume 2: Deployment | *http://www.microsoft.com/technet/treeview/default.asp?url=/technet/prodtechnol/sql/deploy/confeat/sqlha/SQLHALP.asp* |  |
| Microsoft SQL Server 2000 Operations Guide | *http://www.microsoft.com/technet/treeview/default.asp?url=/technet/prodtechnol/sql/maintain/operate/opsguide/default.asp* | |
| Operating .NET-Based Applications | *http://www.microsoft.com/technet/treeview/default.asp?url=/technet/itsolutions/net/maintain/opnetapp/default.asp* |  |
| Production Debugging for .NET-Connected Applications | *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/DBGrm.asp* | |
| Security Operations for Microsoft Windows 2000 Server | *http://www.microsoft.com/technet/treeview/default.asp?url=/technet/security/prodtech/win2000/secwin2k/default.asp* |  |
| Security Operations Guide for Exchange 2000 Server | *http://www.microsoft.com/technet/treeview/default.asp?url=/technet/security/prodtech/mailexch/opsguide/default.asp* |  |
| Team Development with Visual Studio .NET and Visual SourceSafe | *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/tdlg_rm.asp* | |

 This title is available as a Book