Chapter 9

Testing

Introduction

Along with security, testing is another major topic in software development. Without rigorous testing, there is no possible assertion of the quality of a system, and all organizations involved in software development are aware of the importance of testing. Unfortunately, today, when it comes to address the issues that define the testing discipline, either in a process or in a project, more often than not you will be observing a very craftsman-like situation.

This is paradoxical when you think that 20 or 30 years ago testing was a very rigorous discipline. The explanation is very simple: 30 years ago most software-intensive systems were strategic solutions, enjoying lots of resources and with very stringent requirements on quality. Today, a lot of systems fulfill tactical business objectives, in a very competitive environment, with very tight timeframes and limited budgets. No wonder that over that period, three-quarters of software automation projects have failed, being either over budget, over time, of poor quality, or scrapped altogether (you can find more precise figures in the Standish Group CHAOS report; see `http://www.standishgroup.com/chaos/index.html`). Note, however, that even the well-tested strategic projects from long ago tended to go over budget and over time.

Nonetheless, the most conscious organizations are generally achieving a good level of quality in the related activities, but often at the price of excessive allocation of resources (people and time). The good news is that the situation is generally improving, as the software development industry matures in the six dimensions described in Chapter 6: technology, tools, methods/techniques, processes, people, and organization. This improvement is apparent in the comparison between the 1994 and 1998 figures published by the Standish Group.

The objective of this chapter is to demonstrate how a seemingly peripheral issue like testing can actually be integrated within an overall engineering approach to software development, and how to implement traceability for testing artifacts as for any other type of artifacts. The software engineering approach presented in this book can help you achieve a high level of organizational performance in the testing activities, as defined by two elements:

- Quality of testing, by defining techniques and methods ensuring the complete test coverage of all the requirements of the system, verifiable by the capability to trace the results to the requirements.
- Efficiency, by defining a rigorous process for testing, but also by enabling the automation of a large number of tasks.

This chapter will focus on functional testing, as a means of completing the exploration of the functional aspects of the development of a software system. Other test topics (which we will not discuss) would be:

- Performance testing. For example, the number of concurrent sessions that a single Web server can support at a 1- and 5-second response time for known page composition.
- Stress testing. For example, the number of concurrent sessions that a single Web server can maintain, while "gracefully" degrading its performance. "Gracefully" in this context means without loss of data or emergence of error conditions in the system itself.
- Scalability testing and capacity planning. This involves increasing the load at the same time as augmenting the infrastructure and verifying that the system response is kept within the set targets.
- Availability and resilience:

  - Failover, or how long it takes to happen, whether any transactions are lost/fail in the meantime.
  - That a single failure does not cause loss of all of a certain type of system components.
  - Whether the level of service degradation can be controlled (e.g., max sessions, threads per process, etc.).
- Security testing. For example, technical vulnerability exploitation (brute force attacks, buffer overflows, replay attacks, session hijacking) and denial of service attacks.
- Acceptance testing. Representatives of the user stakeholders test the complete system in order to assess readiness for operation.

Two other topics that will not be discussed are configuration management and defect tracking, which address issues like staging of the test configuration, test versioning, and bug tracking. Though very important and interesting, a discussion on these topics will drive the focus away from the core concern of this book, which is the process of refinement of the functional specifications. Due to the vastness of the subject, it is impossible to cover all of the testing discipline within one single chapter. For further enlightened reading on the topic I recommend *A Practical Guide to Testing Object-Oriented Software*; for in-depth coverage of advanced methods and techniques, see *Testing Object-Oriented Systems*. You will also find important insight on testing as part of the overall software development process in *The Rational Unified Process: An Introduction*.

Functional testing takes a black box or behavioral perspective to testing. This perspective consists of completely disregarding any possible knowledge of the way the unit/system operates internally. This is in contrast to the white box or structural perspective, where the test definition takes into account the understanding of the inner works of the unit/system. Among the tools that are used for white box testing are the code debuggers. The white box approach is often more appropriate in situations where the requirements are not well defined for the unit. This is certainly not the case when you apply the practical process described in this book, but the reality is that to achieve a practical and effective approach to testing, it is a good practice to combine the two approaches.

In this chapter we will consider two stages of functional testing:

- Unit test. A unit is the smallest amount of code that is allocated to one developer. A good practice in an OOAD context is to define a unit to be

the code for one implementation class. The rationale for this choice is that a class defines the smallest self-contained unit, encapsulating its code and data. As such, it is also a good practice to define it as the smallest unit of responsibility for a developer. In unit testing, any call to another implementation class outside the unit is replaced with stubs or simulators. Calls to .NET framework classes or trusted components (external systems with .NET managed interfaces) are maintained as is. Calls by other implementation classes are replaced with drivers. Thus, the unit is tested in isolation. Unit testing is a responsibility of the developers, as it is most likely to take a white box testing approach, making extensive use of debugger tools.

- System test, which considers the whole solution as a unit. Within one specific development iteration, the system test will cover the relevant use cases, as well as all use cases from previous iterations. This entails applying regression tests for the test cases that cover the use cases of the previous iterations. System testing is the responsibility of the testers, defined as a separate team.

As you will see later in this chapter, both unit test and system test artifacts trace back to use cases through the use case scenarios: unit tests through the design of the implementation classes in sequence diagrams, and system tests through the test cases. Both sequence diagrams and test cases are based on use case scenarios. Thus, the two groups of workers involved in these activities have a common reference source of knowledge that defines their objectives, and they can easily achieve a common understanding in their communication.

As a final remark for this Introduction, it is not in the scope of this chapter, or of the book in general, to review any specific test automation tools, either for unit test or system test. The objective is to present you with the concepts involved in the testing activities, as well as practical methods and techniques to apply.

Approach

The approach presented puts the test development effort in a parallel track with the design and code efforts. The input artifacts for test activities, as for all the design activities, are use case descriptions, as we take the perspective of the functional aspects of software development. After all, this is a use case-driven process, and use cases are the original representation of the system

knowledge, defining the system requirements and specifications, the code, the documentation, and the tests. In reality you will see in this chapter that test cases and use cases are tightly related. Indeed, the overall idea of the approach is to map the use cases onto test cases.

Test Cases

As test cases proceed from use cases, the activity of specifying them can start very early in the overall development process, as soon as the use cases of the current iteration are specified in detail. The best timing is to start developing the test cases along the use case detail descriptions. However, because use cases are likely to be updated throughout the development of the analysis model, you must be aware that you will need to maintain the consistency of test cases with the use cases. On the other hand, having the test cases completed and stable, along with the use cases at the end of the analysis model and before developing the design model, will be useful in order to start testing the implementation classes that are developed in parallel with the design model.

Test cases are the basic components of testing. A test case is defined as a set of data inputs, execution conditions, and expected results. You can think of each test case as representing a use case scenario, or a complete path through one use case. Each use case scenario involves some or all steps of the basic flow of events and possibly one or more alternate flow of events. The above definition of test cases is very close to the definition of sequence diagrams in Chapter 6, and a practical approach to develop test cases is to create at least one test case for each use case scenario represented by a sequence diagram. Note that this does not mean using the sequence diagrams to describe the test cases, as this approach would effectively define a white box test. It does mean to base the test case on the same description of the use case scenario that defines the sequence diagram.

As a practical approach, each use case scenario is focused on a particular flow of events (basic or alternates). Consequently, defining test cases from a use case scenario effectively means to create one or more test cases for each documented flow of events of the use case. For this reason, all test cases that correspond to one use case scenario define a class of test cases that specify the same execution conditions, differing only in their input data and the results. At the same time, the activity of creating test cases is also an opportunity to unveil important use case scenarios, beyond those defined by the simple enumeration

of the use case flow of events. In this perspective, test cases are also a very useful validation and consistency check for the use case scenarios.

In summary, a use case will map to one or more test cases that exercise its basic and alternate flow of events. Also, a use case maps to one or more implementation classes and their collaborations that realize its basic and alternate flows of events. Thus, the objective of mapping use cases to test cases is to ensure that each test case exercises the implementation classes that realize one or more flows of events of the related use case. Achieving this objective will ensure that the testing covers the system specification as defined by the use cases. Figure 9-1 depicts the concepts discussed in this section and their relationships. Note that in the selected approach, any use case scenario exercises only one or two flows of events: one when only the basic flow is involved, two when the basic flow and one of the alternate flows are involved. In this diagram you will also notice the concept of Test Scenario, which is a compound of test cases that represent a complex user task or a complete user session. Test scenarios are discussed in their own section later in this chapter.

All test cases of a specific use case scenario share the same definition of execution conditions, which represent the set of data elements that is involved in the corresponding use case scenario. These conditions can be represented in a test case matrix that is specific to the related use case scenario (see the magazine article "Generating Test Cases From Use Cases"). Table 9-1 is the matrix

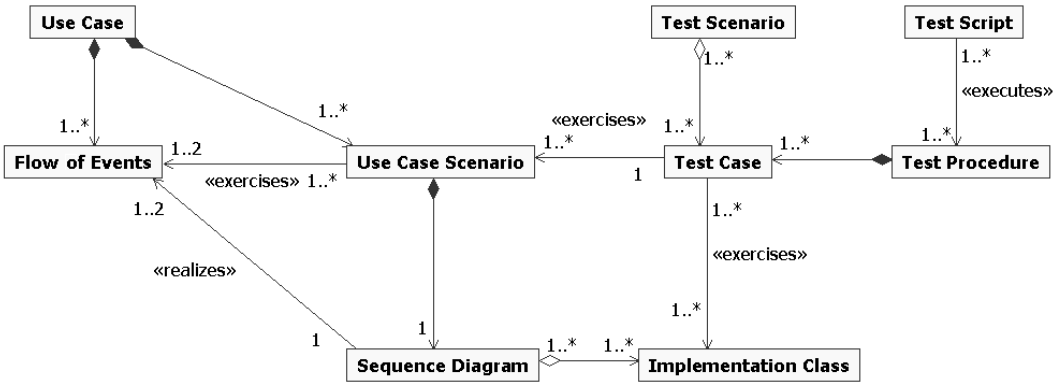

Figure 9-1: Concepts involved in testing and their relationships.

Table 9-1: Test Case Matrix: Definitions of the Execution Condition and Test Results

Use Case	Create Account		
Use Case Scenario	CreateAccount_Entry validation		
Test Case ID	**TC4**	**TC5**	**TC6**
Condition	Username exists	Passwords do not match	Invalid card expiration date
Username	I	N/A	V
Password	V	V	V
Verify Password	V	I	V
E-mail address	V	N/A	V
Name on Credit Card	V	N/A	V
Card Number	V	N/A	V
Card Type	V	N/A	V
Expiration	V	N/A	I
Street Address	V	N/A	V
City	V	N/A	V
State	V	N/A	V
ZIP/Postal Code	V	N/A	V
Country	V	N/A	V
Expected Results	Message displayed: *This user already exists. Please enter another user.*	Message displayed: *The 2 password fields do not match.*	Message displayed: *Expiration date must be between MM1/yyy1 and MM2/yyy2.*

for the "CreateAccount_Entry validation" use case scenario of the Create Account use case. Note that the use case scenario should correspond to a collaboration instance (containing its sequence diagrams) in the design model.

In this matrix we do not enter data values but only an indication of which fields are valid (V), invalid (I), or indifferent (N/A). In TC5 it is sufficient to enter a valid first password and a different verify password to get the expected result, thus all other fields are indifferent. Although there will be other error messages, they are not of any interest for the test case. The above matrix is partial, as it does not cover all the possible error conditions that may occur. More test cases are needed, so that each condition has at least an invalid indication in one test case column.

The next step is to define actual test data in place of the indications on the test conditions. The same matrix template is used for this purpose (see Table 9-2).

In the test case matrix, the expected results describe the state of the system from a user perspective, thus keeping the black box view of the system. They describe in all details the state of the application and any other element of the system environment that is affected as a result of the test activities (e.g., message ABC on the screen, water pump XYZ is now on, message to `abc@xyz.com` sent and will be received within X hours).

Table 9-2: Test Case Matrix: Input Data

Use Case	Create Account		
Use Case Scenario	CreateAccount_Entry validation		
Test Case ID	**TC4**	**TC5**	**TC6**
Condition	Username exists	Passwords do not match	Invalid card expiration date
Username	testbuyer01	N/A	testbuyer10
Password	testbuyer01pwd	testbuyer10pwd	testbuyer10pwd
Verify Password	testbuyer01pwd	Test	testbuyer10pwd
E-mail address	testbuyer01@ softgnsosis.com	N/A	testbuyer10@ softgnsosis.com
Name on Credit Card	testbuyer01	N/A	testbuyer10
Card Number	1111222233330001	N/A	1111222233330010
Card Type	MC	N/A	MC
Expiration	02/04	N/A	02/15
Street Address	1, High Street	N/A	1, High Street
City	London	N/A	London
State		N/A	
ZIP/Postal Code	W1	N/A	W1
Country	UK	N/A	UK
Expected Results	Message displayed: *This user already exists. Please enter another user.*	Message displayed: *The 2 password fields do not match.*	Message displayed: *Expiration date must be between 09/2002 and 09/2007*

Test Procedures

The test case specifications are not enough by themselves to conduct a test run. Test procedures define the steps and environmental conditions (system and application) that need to be in place in order to execute one or more test cases. Test procedures also define the steps to analyze the results. A test procedure can describe the execution of one or more test cases, as depicted in Figure 9-1. The reality is that most of the time you should expect to define a one-to-one map of a test case to a test procedure. In this perspective it is often more effective to create two test procedures that are very similar than to try to merge the two procedures in one.

The various parts of the execution of test procedures are automated by developing test scripts either by coding them or using automated tools to generate them. Automated test tools generate the test scripts by capturing the user actions with the system. Ideally, test procedures should be completely automated.

It is useful to define a test procedure template, as it brings structure to your thinking when describing the steps for testing (see Table 9-3).

The test procedure steps derive directly from the flow of events defining the use case scenario that the test case represents. A simple way to think of a test case is as an instantiation of a use case flow of events, thus effectively using the same description for the execution steps, but imposing on you the requirement to think and document every detail of the state of the system before and after the execution of the test.

The test case matrix effectively defines the combination of data that will be used for a specific test procedure. Note that because a set of test cases within a particular test case matrix correspond to one use case scenario, it is possible that they will also share the same test procedure, and consequently the same test scripts. But in general every test case will potentially have its own test procedure.

In parallel with the test case definitions you must also maintain a repository of the tests executed, in which iteration, and what the result of the test was. If the result did not match the expected state of the system and/or data, an incident will be raised with the developers to investigate the problem. Note that incidents fall into two broad categories:

- True defects, where some code or system configuration has to be modified.
- False alarms, arising from erroneous test case definitions.

In the case of a false alarm, after investigation, the defect along with its assessment shall be transmitted to the test team that will need to amend the test case definition and run the test case again before closing the defect report.

Table 9-3: Test procedure template.

Test procedure	Name of the test procedure *(for identification)*
Related test case(s)	The identification of the test case(s) that this procedure covers *(useful for traceability and assessment of the coverage of functional testing to the system specifications)*
Initial state of the system before the test case starts	Describe in all details the state of the application as well as any element of the system environment that is of importance during the test; e.g., connection to a bar-code reader, particular HTTP port enabled *(useful to avoid false test failure incidents due to inadequacy of the test assumptions with the current system environment and state)*.
State of the data used by the system before the test case starts	Describe in all details the state of the data in the system, as it has to be present before the test is conducted. This may entail describing the steps to prepare the data and reference of any automated script or another document that describes the procedure to prepare the data for the particular test *(useful to ensure that the test does not generate false test failure incidents due to inadequacy of test assumptions with available system data)*.
Test procedure steps	Describe in all details the steps that the user has to follow. Be very specific, referencing the specific buttons or links to click, or input fields to fill. The data itself is one of the data sets from the test case descriptions covered by the procedure.
Expected final state of the system after the test case finishes	Complementary description of the test case results. This after the test case finishes description is optional and focuses on the elements of the system environment that are not specified within the corresponding use case *(useful in order to add a white box perspective to the test)*.
Expected state of the data used by the system after the test case finishes	Describe in every detail the state of the data in the system as a result of the test. This may entail referencing extended documentation on how to extract the data from the system; e.g., a specific SQL query to execute *(useful in order to add a white box perspective to the test. Note that the related scripts need to be created by the developers.)*.

Effectively this procedure consists of handing over the responsibility of correcting the defect, while the overall defect management and resolution procedure stays the same as in the case of a true defect.

Test Coverage Matrix

From the above discussion it is straightforward to trace every flow of event of every use case to a test case. But considering the discussion on test case mapping to implementation classes, a practical question that springs to mind is how to ensure that every operation of every implementation class is participating in a test case.

The indirect and necessary answer to this question is to consider the sequence diagram of a use case scenario, representing a particular flow of events of a use case, and list the implementation classes with their operations involved in the collaboration defined in the design model. As this particular flow of events has a corresponding test case, the implication is that the test case does exercise the corresponding implementation classes and the operations involved in the sequence diagram. A first evaluation of test coverage can be documented by using a simple matrix, named the test coverage matrix, where you represent all implementation classes with all their operations in the first two columns, along with all the test cases on the first row. Then you mark with an X every cell where an implementation class is involved in the test case(s) corresponding to the design model sequence diagram(s) where the implementation class is used.

At the same time, as the matrix effectively references all units of the system, it is the best place to track the completeness of unit test definitions. The other advantage is that you can quickly identify test case data to use for the unit test, as it will be some derivation of the test case data that covers the corresponding class operation. Table 9-4 is a conceptual example of a test coverage matrix.

In the example in Table 9-4 you can identify a problem with the operation OB2 of implementation class CB, as it seems that it is not exercised in any test case. As described in the design model in Chapter 6, the process of discovering class operations is based on developing the sequence diagrams for each possible flow of events of a use case, represented in a use case scenario. Consequently, an operation that is not covered by a test case can only mean that there is a test case missing for each flow of events where the operation is used.

Table 9-4: Test coverage matrix example.

Class	*Operation*	*Unit Test*	*Test Case A*	*Test Case B*	*Test Case C*
CA	**OA1**	X	X		X
	OA2	X		X	X
CB	**OB1**	X		X	
	OB2	X			
	OB3	X		X	
CC	**OC1**		X		

A second problem involves the OC1 operation of implementation class CC, where you can identify that a unit test has yet to be defined. Thus, the test coverage matrix is a powerful validation technique for test coverage. There should be only one test coverage matrix for all the system, in order to use it as a synthetic view, or dashboard of the state of completeness of the testing effort.

Continuing the investigation of the above question on assessing the test coverage of every operation, a direct answer that would complement the above test coverage matrix would be to use a code coverage tool. This type of tool can identify the lines of code in the program that are never executed during testing. The approach is very simple and consists of running the code coverage tool while you execute a particular test case. Then you analyze the results produced by the tool and verify in the test coverage matrix that all of the classes and operations that have an X for the corresponding test case have been executed during the run. If not, you must investigate whether this situation is due to the definition of the test case or the definition of the sequence diagram.

Applying both of the above techniques will give you the necessary and sufficient conditions to positively assert the question of the test coverage of implementation classes. Note that using a test coverage tool is part of a white box approach to testing, as these tools are effectively analyzing the structure of the classes and operations. Nonetheless, the approach described demonstrates the complementarities between the two techniques. Although white box testing is more in the realm of the developers, who have an intimate understanding of the inner workings and structure of the code, you can find code coverage tools like Rational PureCoverage that integrate transparently with automated testing tools in a black box approach to testing, thus effectively bridging the gap

between back box and white box testing and integrating the testing activities of testers and developers.

Test Scenarios

Test cases are used as building blocks for test scenarios, which represent plausible usage of the system by end users in terms of a series of tasks that the user will want to execute. They are useful in describing typical user sessions, involving a sequence of use cases, some of which might be repeated; for

Table 9-5: Test scenario template.

Test Scenario	Name of the test scenario *(for identification)*
Purpose	Describe the rationale why this scenario was defined.
Description	Describe what this scenario does.
Test type	Functional/Load/Stress/Performance/Security…
Test cases used	The list of test cases involved in the scenario. Reference the test case names.
Initial state of the system before the test scenario starts	Describe in all details the state of the application as well as any element of the system environment that is of importance during the test; e.g., connection to a bar-code reader, particular HTTP port enabled.
State of the data used by the system before the scenario starts	Describe in all details the state of the data in the system, as it has to be present before the test is conducted. This may entail describing the steps to prepare the data and reference any automated script or another document that describes the procedure to prepare the data for the particular test.
Test scenario steps	Enumerate the test cases that have to be applied. Identify repeating sets of test cases and how many repetitions; for example: 1. Execute test case: A Execute 2 followed by 3 for 7 times. 2. Execute test case: B 3. Execute test case: C 4. Execute test case: D
Additional directives and information to collect	Any extra details on the test execution and what information to collect or verify in order to assess the success of the test.

example, when the user shops for books, a test scenario can be composed of sign-in, repeat N times (browse catalogue; select books), check-out, sign-out.

Test scenarios are particularly useful when developing load tests where you should create a representative mix of use cases that are likely to happen in parallel during normal system operation. As test cases are the building blocks of test scenarios, these will be composed of one or more test cases. In test scenarios you might also mix the basic flow of events of some use cases, with the alternate flows of other use cases in order to verify that integrity of the data is maintained. Similar to the test cases, it is desirable to define a practical template for test scenarios.

I will not cover test scenarios in any more detail in this book, as they do not bring significantly more insight than test cases to the discussion of functional testing.

Unit Test

A unit test is most likely to entail using a combination of white box and black box approaches. As seen in the Introduction, the black box approach to unit testing mandates that all implementation classes stub all the calls to other implementation classes that are external to the unit under consideration. As a small reminder and a matter of practicality, the strongly typed DataSet classes described in Chapter 6 are considered as .NET infrastructure classes and consequently trusted to be working correctly during unit tests. Stubbing the implementation classes is quite easy to achieve for three reasons:

- The class operation called has already been defined, as per applying the approach of Chapter 6. Thus, the stub structure already exists.
- By virtue of the design pattern described in Chapter 6, the only implementation classes that should ever be returned to the caller are the strongly typed DataSets, as representing system data. As these classes are trusted, you can create instances inside the stub and populate them with the appropriate data needed for the unit test.
- Other classes that need to be returned are .NET framework classes, which are also trusted and can easily be created and populated inside the stub with the appropriate data for the unit test.

The first step for black box unit testing consists of systematically creating stubs for all implementation classes as soon as these classes and their operations are defined in the design model. Figure 9-2 gives an example of a stubbed class structure. Note in the implementation namespace the reference to a ".Stubs" namespace to be used for the definition of the actual class implementation.

Just copy within the same implementation file the structure of the implementation class, but under a different namespace that has a ".Stubs" suffix to the actual implementation class namespace. The calling units will need to reference that namespace in place of the default namespace of the class. This simplifies the work for system integration, as you only need to take out all the ".Stubs" suffixes of the "using" clauses in each code file in order to start calling the actual implementation classes. The other advantage is that it prevents the stub class definitions from getting in the way and hogging the implementation models, as they are kept separate under different namespace packages, as seen in Chapter 7.

All operations definitions within the stub class definition remain the same as for the real class. The code of a stubbed operation has as a primary objective to create and populate return objects with static data. In order to do so, it is very useful to get inspiration from the test case data that is defined for the test cases that cover the same operation, as per the test coverage matrix. At the same time, the structure of the stubbed code also defines the initial structure of the corresponding actual class, with the constraint of not calling to any other actual or stubbed implementation class. Adopting this approach is helpful in two ways:

- It helps you think of the structure of the actual implementation class and its operations. In this perspective it is a complementary design step.
- It helps to create a stub that simulates various normal or abnormal conditions, but sourcing exclusively from the class structure. You should ban any attempt to consider what error conditions are carried over from downstream calls. Defining normal and abnormal conditions is helpful for improving the testing of the calling class.

When all stubs are created, they are made available to all developers to use while they implement the units they have been assigned. The next step is to create unit tests for each unit. It is interesting to make a parallel with what

happens in Extreme Programming (XP), where the process also takes a similar approach in putting even more of an emphasis on specifying test cases as the basis for coding.

Unit tests are effectively realized by code components that represent the concept of test drivers discussed in the Introduction. The general pattern of a test driver is to initialize the required objects, execute a call to the operation

```
namespace BooksREasy.UserAcctMgr.UserAcctMgrBLL
{
    using System;
    using BooksREasy.Common.UtilityClasses;
    using BooksREasy.UserAcctMgr.UserAcctMgrDAL.Stubs;

    public class AccountManager : MarshalbyRefObject
        /// <summary>
        /// getUserAccountByUserId: Locates the user account with the specified userId.
        /// </summary>
        /// <param name="UserId">userID of the user to retrieve user account information for.</param>
        /// <returns>Typed UserInfo dataset that contains the UserAccount record with a matching userID.</returns>
        public UserInfo getUserAccountByUserId (string userId)
        {
            //
            //TODO: implement the operation
            //
        }
    }
}
//Stubs
namespace BooksREasy.UserAcctMgr.UserAcctMgrBLL.Stubs
{
    using BooksREasy.Common.UtilityClasses;
    public class AccountManager
    {
        public UserInfo getUserAccountByUserId (string userId)
        {
            UserInfo dsUser = new UserInfo();
            if (userId=="testbuyer10")
            {
                dsUser.UserAccount.AddUserAccountRow(dsUser.UserAccount.NewUserAccountRow());
                dsUser.UserAccount[0].UserId=testbuyer10;
                dsUser.UserAccount[0].Password="testbuyer10pwd";
                dsUser.UserAccount[0].Email="testbuyer10@softgnosis.com";
                dsUser.UserAccount[0].Addr1="1, High Street";
                dsUser.UserAccount[0].City="London";
                dsUser.UserAccount[0].Country="UK"
                dsUser.UserAccount[0].SecretNumber="1111";
                dsUser.UserAccount[0].State="";
                dsUser.UserAccount[0].Status=CommonKeyWords.ACTIVE;
                dsUser.UserAccount[0].Zip="W1";
            }
            return dsUser;
        }
    }
}
```

Figure 9-2: Defining a ".Stubs" namespace for each implementation class.

under test, and check the state of the returned objects. It is also a good practice to keep the unit test code along with the definition of its corresponding class and stub, using a namespace suffixed with ".Tests" in a similar way as for stubs. Figure 9-3 presents test drivers.

Note that you also need to have a utility application to run these tests and manage results. At this stage it is useful to use a unit test tool to automate this activity, which brings two advantages:

- It defines a structure for documenting the unit test, including the test code.
- It defines an environment to execute the tests and track test results.

At the time of this writing, I have evaluated two unit test tools that were specifically designed for the .NET platform: Nunit and HarnessIt. Both take advantage of the attribute-based programming of .NET.

Having defined the stubbed implementation classes and the test drivers, we are in the position to test the unit in complete isolation, as required by a black box approach to testing.

In reality, it is not always easy or effective to implement a unit test exclusively using a black box approach. This is due mainly to the inherent complexity of the underlying .NET technological framework. As much as it

```
namespace BooksREasy.UserAcctMgr.UserAcctMgrBLL.Tests
{
    using System;
    using System.Diagnostics;
    using BooksREasy.Common.UtilityClasses;

    public class AccountManager_getUserAccountByUserId_Test
    {
        public void success()
        {
            UserInfo dsUser =new AccountManager() .getUserAccountByUserId("testbuyer10");
            Debug.Assert (dsUser.UserAccount.Rows.Count == 1
                && dsUser.UserAccount[0].Status.Equals (CommonKeyWords.ACTIVE), "Success");
        }

        public void fail_Invalid_UserId()
        {
            UserInfo dsUser =new AccountManager().getUserAccountByUserId("dummyUserId");
            Debug.Assert (dsUser.UserAccount.Rows.Count == 0,"Success");
        }
    }
}
```

Figure 9-3: Defining a ".Tests" namespace for each implementation class.

simplifies development, it also forces the designers to rely on the complex .NET mechanisms that work behind the scenes.

As an example, let's consider what happens when a UserControl is passed as an argument to a dispatcher class operation. If you want to apply a pure black box approach to the unit test, you need to completely isolate the class operation by creating a driver to call it. But then it is your responsibility to somehow create and initialize a UserControl class in order to use it as a parameter to the called operation. This is not a very practical and efficient usage of your time. The reality is that specifically for unit tests you have to be practical and not hesitate to take a white box approach to testing when it is appropriate. White box testing involves using debugger tools.

Thus, a definition of an overall approach for unit testing would be to start with a black box approach and complement it with a white box approach for the complex situations. At the same time you should be very conscious to document the unit tests, either by virtue of using a unit test tool or with documents or code annotations describing the tests. This is particularly important for white box testing, which is largely a manual process involving debugging procedures.

You also need to identify and acknowledge some dependencies in the timing of the development of the implementation classes, in order to take advantage of existing parts of the system, as the previous example suggests. Because of the very nature of unit tests, these dependencies often go in the inverse direction of the implementation dependencies defined by the sequence diagrams of the class collaborations (e.g., in the design model, a UserControl class depends on a dispatcher class).

Fortunately, this is not limiting, as during the design of sequence diagrams you will define the operations of the dependent classes based on the needs of the calling classes (e.g., the dispatcher class operation called by the UserControl class will already have been defined by the time you need to code its implementation). Thus, before developing the code of a class operation, you can already define a unit test for that operation, as well as have a test driver to test it. The only apparent limitation is that you do not completely control the test driver, as the call generated is partly the responsibility of the .NET framework. This situation is not limiting either, because of the assumption made in the Introduction of this chapter to trust the .NET components. The only impact is that for documenting the test you have to capture the data created by the .NET framework, instead of defining it beforehand.

System Test

In a simple and sufficient view for the discussion of this chapter, the system test consists of executing all the test cases defined so far for all the iterations of the system. Indeed, as described earlier, test cases are finalized at the same time as use cases. Thus, when the development team has developed the code implementing the use cases of the current iteration, it is possible for the test team to run the corresponding test cases. But at the same time, they also need to rerun all the test cases corresponding to use cases for the previous iterations, in order to ensure that the new state of system development has not introduced any defects to a previously defect-free system. This part of the testing is named regression testing.

It is clear by now that test cases will possibly be run a great number of times. For this reason, it is important to define a strategy to automate the test execution as much as possible. This entails using test automation tools, but also creating scripts that will set up the environment and initialize the system with the correct data. At the same time, you need to think of automation solutions to capture the state of the system and data upon completion of each test case.

Case Study

Applying the above approach to the case study, we will produce the functional test artifacts relating to the Create Account use case, focusing specifically on the basic flow of events.

Test Cases

The following test case matrix defines the three test cases that are sufficient to cover the use case scenario of creating a new user account. The start of the use case scenario specifies that the user accesses the Create Account screen either from a link on the main menu or through a button on the sign-in screen. Thus, the first two test cases, TC1 and TC2, are there to ensure that this part of the use case scenario is tested.

In TC3, the State field is indifferent because the system does not do any complex validation between the Country, the State, and the ZIP/Postal Code fields. This is the result of a simple system specification and not an error in the design (see Table 9-6). If more complex validation becomes a requirement, the

Table 9-6: Test Case Matrix: Definitions of the Execution Condition and Test Results

Use Case	Create Account		
Use Case Scenario	Basic Flow		
Test Case ID	**TC1**	**TC2**	**TC3**
Condition	Create Account is accessible from Sign-In	Create Account is accessible from menu item	Basic flow
Username	N/A	N/A	V
Password	N/A	N/A	V
Verify Password	N/A	N/A	V
E-mail address	N/A	N/A	V
Name on Credit Card	N/A	N/A	V
Card Number	N/A	N/A	V
Card Type	N/A	N/A	V
Expiration	N/A	N/A	V
Street Address	N/A	N/A	V
City	N/A	N/A	V
State	N/A	N/A	N/A
ZIP/Postal Code	N/A	N/A	V
Country	N/A	N/A	V
Expected Results	Create Account page is displayed.	Create Account page is displayed.	Account created and user signed in. Message displayed: *<userX> signed in.*

change will ripple throughout the use case specifications and system design to also impact the test case, which would then need to be changed to account for that new requirement (see Chapter 10 for the impact analysis of a change request involving the validation of the ZIP/Postal Code).

Table 9-7 is the test case matrix with the input data. Notice that we have not defined any data for the State field, as this is not a relevant concept for the United Kingdom.

Table 9-7: Test Case Matrix: Input data.

Use Case	Create Account		
Use Case Scenario	Basic Flow		
Test Case ID	**TC1**	**TC2**	**TC3**
Condition	Create Account is accessible from Sign-In	Create Account is accessible from menu item	Basic flow
Username	N/A	N/A	testbuyer10
Password	N/A	N/A	testbuyer10pwd
Verify Password	N/A	N/A	testbuyer10pwd
E-mail address	N/A	N/A	testbuyer10@ softgnsosis.com
Name on Credit Card	N/A	N/A	testbuyer10
Card Number	N/A	N/A	1111222233330010
Card Type	N/A	N/A	MC
Expiration	N/A	N/A	02/04
Street Address	N/A	N/A	1, High Street
City	N/A	N/A	London
State	N/A	N/A	
ZIP/Postal Code	N/A	N/A	W1
Country	N/A	N/A	UK
Expected Results	Create Account page is displayed.	Create Account page is displayed.	Account created and user signed in. Message displayed: *testbuyer10 signed in.*

Test Procedures

All the three test cases defined in the previous section can be executed within the test procedure defined in Table 9-8. It is a good practice to refer to the test cases within the test procedure steps in order to verify the consistency of the test procedure definition. The right mindset to acquire in order to develop test procedures is to consider every detail to be important.

In the test procedure in Table 9-8, Step 2 references TC3 as the sole test case data that is available for that step. In another situation where more than one test case data could be used for the same test procedure (e.g., TCxyz), a simple way to indicate alternate usage would be to write the step as: *At the*

Table 9-8: Create Account—Basic Flow test procedure.

Test Procedure	Create Account—Basic Flow
Related test case(s)	TC1, TC2, TC3
	■ Initial state of the system before the test case starts The test site is up and running
	■ A Web browser is open on the test machine, pointing to the test site. No user is signed in for this browser session.
State of the data used by the system	■ The user testbuyer10 does not exist in the BooksREasy before the test case starts system.
Test procedure steps	1. The Create Account use case is accessible via the Sign-In page and the Create Account link on the menu: a. (TC1) Click the Sign-In link and verify that the system displays the Sign-In page. Click Create Account, and then verify that the system displays the Add Account Information page. b. (TC2) Click the Create Account link and then verify that the system displays the Add Account Information page. 2. At the Create Account page enter test case data TC3 and click the "Submit" button. Verify that the system displays the Verify Account page. 3. Verify that the data on this page corresponds to test case data TC3, and click on the "Yes" button. 4. Verify that you are signed in as "testbuyer10."
Expected final state of the system after the test case finishes	The Account Home page is displayed and contains the following string: *testbuyer10 signed in*.
Expected state of the data used by the system after the test case finishes	The system adds a record to the UserAccount table. The system adds a record to the CreditCard table. The system adds a record to the UserGroup table. Verify that the following query on the BooksREasy database returns one row with the columns matching the test case data TC3: SELECT * FROM UserAccount, CreditCard, UserGroup WHERE CreditCard.UniqueId=UserAccount.UniqueId AND UserGroup.UniqueId=UserAccount.UniqueId AND UserAccount.UserId='<username>' where you replace <username> with the actual value of the username in the test case data; e.g., testbuyer10 for TC3 Also verify that in the returned row the GroupId column contains "Registered Buyer."

Create Account page enter test case data TC3/TCxyz and click the "Submit" button. Verify that the system displays the Verify Account page. The expected results are defined by each test case. But in general if the two test cases define different system environment and data states, it is easier to copy the test procedure and adapt its definition.

Test Coverage Matrix

Because there is only one test coverage matrix for all the system, Table 9-9 is an excerpt that presents only the implementation classes participating in the sequence diagrams for the use case scenario defining the basic flow of events. As a reminder, you need to consider the sequence diagrams of the design model, because this is the only model that references implementation classes. Notice that the UserGroupInfo class, which appears on the "Detail:Assign User to Group" sequence diagram (see Figure 6-27 in Chapter 6), is notably absent from the test coverage matrix. The reason is that this is a strongly typed DataSet class, thus it is considered an infrastructure class, which does not need to be assessed with any test case. To create this matrix you need to consider the sequence diagrams that correspond to the use case scenario covered by the referenced test cases. These diagrams were presented in Figures 6-23, 6-25, 6-26, and 6-27 in Chapter 6.

In Table 9-9 we can make the following remarks:

- CreateAccount_Entry class: Strictly considering the sequence diagrams, we cannot identify that InitializeComponent and OnInit are tested in the listed test cases. But because we know the way that .NET executes these operations, we also know that they are called before the Page_Load operation.
- CreateAccount_Entry class: Because the btnCancel_Click operation is not covered by any of the specified test cases, it should be covered by test cases corresponding to its use case scenario. Indeed, this operation relates to the "User Cancels Request" alternate flow of events for the Create Account use case.
- CreateAccount_Verify class: Same remark for the btnCancel_Click operation as in CreateAccount_Entry class.

Table 9-9: Test coverage matrix excerpt.

Class	*Operation*	*Unit Test*	*TC1*	*TC2*	*TC3*
CreateAccount_Entry	btnCancel_Click				
	btnSubmit_Click				X
	InitializeComponent		X	X	X
	IsCreditCardValid				X
	IsExpirationValid				X
	IsUserUnique				X
	OnInit		X	X	X
	Page_Load		X	X	X
CreateAccount_Verify	btnCancel_Click				
	btnYes_Click				X
	InitializeComponent				X
	OnInit				X
	Page_Load				X
Home_Account	InitializeComponent				X
	OnInit				X
	Page_Load				X
CreateAccountDispatcher	ProcessEvents				X
	ShowEntry				X
	ShowVerify				X
	SubmitEntry				X
	SumbitVerify				X
AccountManager	activateUserAccount				
	CreateAccount				X
	DeactivateUserAccount				
	GetCreditCardInfo				
	getCreditCardInfoByCardNumber				
	getUserAccountByUniqueId				
	GetUserAccountByUserId	X			X
	GetUserGroupByUniqueId				
	SetUserUIDToGroupId				X
	UpdateAccount				
	ValidateAccount				
CreditCardManager	chargeTransactionFee				
	IsCreditCardValid				X

Class	*Operation*	*Unit Test*	*TC1*	*TC2*	*TC3*
UserAccount	Create				X
	findByUserId				X
	findByUniqueId				
	Update				X
CreditCard	Create				X
	findByCardNumber				
	findByUniqueId				
	Update				X
UserGroup	Create				X
	findByUniqueId				
	Update				X

- CreateAccountDispatcher class: As its name implies, the operations of this class are very specific to the Create Account use case, so it should be no surprise that all its operations are covered by the selected test cases.
- AccountManager class: In contrast with CreateAccountDispatcher this is a general class, and this is the reason that only a few operations are covered by the selected test cases. Other test cases from other use case scenarios of the same or other use cases will need to cover the rest of the operations.
- CreditCardManager, CreditCard, and UserGroup: Same remark as for AccountManager class.

Unit Tests

In the test coverage matrix of the previous section, the only operation of implementation classes that is marked is getUserAccountByUserId, because, as a matter of conciseness, this is the only operation for which I will review the unit tests. In fact the stub and unit tests of the setUserUIDToGroupId operation have already been reviewed in Figures 9-2 and 9-3, respectively. In the stub definition of Figure 9-2 you can notice the test on the UserId argument. This test enables the support of the abnormal condition that occurs when the requested user account is not in the database. In this case, the specification of the operation is to return an empty DataSet. The actual code of the operation

```
namespace BooksREasy.UserAcctMgr.UserAcctMgrBLL
{
    using System;
    using BooksREasy.Common.UtilityClasses;
    using BooksREasy.UserAcctMgr.UserAcctMgrDAL.Stubs;

    public class AccountManager : MarshalByRefObject
        /// <summary>
        /// getUserAccountByUserId: Locates the user account with the specified userId.
        /// </summary>
        /// <param name="userId">userID of the user to retrieve user account information for.</param>
        /// <returns>Typed UserInfo dataset that contains the UserAccount record with a matching userID.</returns>
        public UserInfo getUserAccountByUserId (string userId)
        {
            return new UserAccount() .findByUserId(userId);
        }
```

Figure 9-4: Definition of the actual getUserAccountByUserId implementation class.

is presented in Figure 9-4, where the only difference with the definition of Figure 9-2 is that we have replaced the "TODO:" with the call to the findByUserId of the UserAccount class in the Data Access Layer (DAL).

In fact, as the UserAccount class defines another unit, the call is directed to the corresponding stub, thanks to the "using" close of the class definition. This is transparent to the developer, who can go about and code the operation definitions as though all the external units were already completed. Figure 9-5 presents the stub of this DAL class. You can notice that it has the same definition as the stub for the getUserAccountByUserId operation of the AccountManager class. This is sensible as the only thing that getUserAccountByUserId does is to delegate to findByUserId. Nevertheless, as explained in the Approach section, you cannot call the findByUserId stub from the getUserAccountByUserId stub.

The unit tests for the getUserAccountByUserId operation were presented in Figure 9-3, where you can notice in the fail_Invalid_UserId test how a cleverly coded stub can help you test various conditions.

Summary

In this chapter we reviewed the activities and artifacts related to the functional testing of the system. With these activities you produce six artifacts: test cases,

```
namespace BooksREasy.UserAcctMgr.UserAcctMgrDAL.Stubs
{
    using System;
    using BooksREasy.Common.UtilityClasses;

    public class UserAccount
    {
        public UserInfo findByUserId (string userId)
        {
            UserInfo dsUser = new UserInfo();
            if (userId=="testbuyer10")
            {
                dsUser.UserAccount.AddUserAccountRow(dsUser.UserAccount.NewUserAccountRow());
                dsUser.UserAccount[0].UserId=userID
                dsUser.UserAccount[0].Password="testbuyer10pwd";
                dsUser.UserAccount[0].Email="testbuyer10@softgnosis.com";
                dsUser.UserAccount[0].Addr1="1, High Street";
                dsUser.UserAccount[0].City="London";
                dsUser.UserAccount[0].Country="UK"
                dsUser.UserAccount[0].SecretNumber="1111";
                dsUser.UserAccount[0].State="";
                dsUser.UserAccount[0].Status=CommonKeyWords.ACTIVE;
                dsUser.UserAccount[0].Zip="W1";
            }
            return dsUser;
        }
```

Figure 9-5: Definition of the UserAccount stub and its findByUserId operation.

test procedures, test scenarios, test coverage matrix, unit stubs, and unit tests. Within this type of testing, we have specifically discussed the two stages of unit test and system test, with a focus on black box testing. Functional testing consists of asserting that the system operates according to its functional specifications as captured in the use cases. The cornerstone of functional testing is the test case, which corresponds to one use case scenario and represents one possible flow of events of a use case.

The test coverage matrix helps to validate that all the classes and all their operations are allocated to at least one test case, and that unit tests have been defined for each one. A complete validation uses a code coverage tool to verify that each class operation is actually exercised within the test case in which it is allocated. Test cases are used to build the test procedures, which need to be followed in order to execute the tests. Test cases can be compounded into test scenarios in order to represent and test complex user sessions on the system.

Unit tests are most likely to use a combination of white box and black box approaches. White box testing involves using debugger tools. In the black box approach to unit testing, you first systematically create stubs as soon as imple-

mentation classes and their operations are defined in the design model. These stubs are then accessible by all developers, who first create unit tests for the units they have responsibility for. Only then can developers start coding the implementation of each operation, using stubbed calls to external units that are not yet available.

Creating stubs and unit tests is also useful as an additional level of design, before starting to code the actual operations. When the unit implementation is complete, the unit tests will assess that the unit operates to its specifications. Using the .NET namespaces feature to your advantage, you can write code that you can easily integrate with just a change of the namespace referenced within the definition of an implementation class.

System tests take place at the end of the development iteration and consist of executing all the test cases of all the iterations before and including the current one. Regression testing is the act of repeating the execution of test cases that have passed in a previous iteration, in order to assert that the new development has not induced any defects in any part of the system previously tested. For this purpose, it is important to use automated testing tools to support the testing activities.