Testing

CHAPTER

3

IN THIS CHAPTER

Although most system architecture books include a chapter on testing toward the end of the book, I believe that testing should be considered in the beginning of every project and at every step along the way. Therefore, I am including a chapter dedicated to testing early on in this book, in the hope that you will plan and design your projects with testing in mind.

Often, in real-world projects, testing isn't used as much as it should be and, when it is used, it is considered extremely time-consuming. Meanwhile, developers often claim they need a third-party testing tool to get started on their testing work. However, testing is more about organizational attitude than about fancy tools. If you value testing in the beginning of a project, it will help you in the long run, regardless of the tool you use. I firmly believe that the amount of testing you do is inversely related to the stress level in your project. In addition, you probably create more bugs under pressure than when you are working in a calm environment.

Most of this chapter will be devoted to making testing automatic to inexpensively increase the long-term quality of your application. We will start by looking at the various types of testing available. I will then present a proposal to automate testing that you can adapt to your own projects. Next we will look at how assertions can be used to make the code test itself. Finally, we will discuss a diagnose/monitor tool as well as important testing tips. I will conclude the chapter by evaluating the automatic testing and assertions proposals I have made based on the factors I presented in Chapter 2, "Factors to Consider in Choosing a Solution to a Problem."

A Short Introduction to Testing

This section is not meant to be used as a guideline for the Quality Assurance folks; rather, it is a recap of testing issues for developers—both considering the testing level and the type of testing needed—before we start getting our hands dirty.

Considering the Testing Level

Before you begin testing any project, you must first consider the level at which you want your testing efforts to be applied. You could test the entire application (and you definitely should) but it's important to consider individually the parts of the whole and to test them early on. If you postpone testing until the application is done, you're in for trouble. It will be too late to fix serious problems, and you most likely won't be able to conduct enough testing due to lack of time. I highly recommend you test the subsystems as they are being built, but even that is too granular and too "late" a level if it is the first one at which you test. Start testing directly at the unit level for components and stored procedures, and test all of them.

A couple of years ago, I had a heated discussion with my wife, who was working as a project manager for a test group at the time. She said that her group was in a wait state because they couldn't start testing their subsystem because it was dependent on another group's subsystem

that hadn't been built yet. I told her that they should create stubs to simulate the delayed subsystem, giving the throughput that was to be expected and so on. (Stubs are empty methods but with the signature of the real methods. They are used to simulate the finished methods, to make dependent parts testable early in the process.) However, my wife thought it would be too expensive for her group to create the stubs and so wanted to wait until the other group's subsystem was complete. As you might guess, I didn't agree with this. I think that creating the stubs would have been cheaper than delaying the tests. (On the other hand, as I tried to remind myself, the less you know about a problem, the easier it is to solve. I guess I ought to say this should my wife ever read this book.) The bottom line is: Test at the unit level, and then at the subsystem level (by testing all the components in that subsystem), and finally at the complete system level, with all the subsystems working together.

Determining the Type of Testing

Depending on the level at which you will test your system, you can and should use different types of testing. It's important to note that no single technique will be the only one needed. Let's take a look at the different types of testing you should consider.

Development Testing

As developers work with a component, they obviously must test the component from different perspectives. Does it work in this case? Do I get the result I expect for a bunch of different combinations of input? And so on.

Historically, VB has been a great tool for writing code, testing it, and correcting problems as you test, continuing to write more code as you go. This is a very productive approach. What is *not* a productive approach is to have only the final User Interface (UI) as the test driver when you build the components. There are several problems with this, including the following:

- The UI might not be finished yet.
- Clicking through several dialogs before you get to where you can test your method is not just time consuming and frustrating, it can also lead to carpal-tunnel-syndrome-like symptoms.
- It will seldom be easy to test all the functionality of your component, as a lot of it will be "hidden" by the UI. (This is especially true for error-handling code.)

Most developers conduct development testing as they go along. However, the process of development testing may need to be formalized. We'll discuss this in more detail later in the chapter.

Code Reviewing

One powerful testing technique is simply to review your code. This is especially efficient if you ask someone else to review your code. In my first real-world project, a mentor came in to

check on us now and then. One time he said to me, "Jimmy, show me your best function." I quickly guessed that he wanted to see some code with a lot of comments, and I remembered one function I had recently written that was heavily commented. (Yes, all my code was commented, but this function was more so than average.) I proudly showed him that sample. After half an hour or so, I wasn't so proud of the code anymore. He had given me feedback on how I could improve every single aspect of the function. Perhaps he was too tough on me, given that it was so early in my career, but on the other hand, it helped me improve the quality of my code greatly. Not only did his ideas inspire me, I was also afraid that he would ask me to show him my best function again!

A few organizations have formalized code reviewing so that every component must be signed by a reviewer, for example. On the other hand, the majority of organizations have only just begun to discuss implementing a code-reviewing policy. I recommend you start conducting code reviewing if you haven't already done so. Let somebody read your code for an hour; you will be surprised by the amount of potential problems that someone other than yourself can see easily.

I've encountered many examples of the importance of code reviewing. One in particular stands out. A colleague of mine once built an application that was to be used on a national television show that was collecting money for charity. My colleague had a hard time convincing the customer that it was important to test the application, and so he was very anxious about his application failing during prime time. Therefore, he asked me to review the code. Although he is the most skilled programmer I know, and had reviewed the code several times himself and carried out stress tests and such, nevertheless, in two hours I was able to point out three major risks that he was able to remove before using the application. The application ran with no problems on the show. (Perhaps it still would have without my review, but it was a much smaller "perhaps" with the review.)

Code review does involve some problems you have to watch out for. For example, you must take care that programmers don't feel personally insulted by the review process. Also, be careful that the code review doesn't focus solely on checking formalism, such as that the naming convention is followed and so on. Such reviewing is important, of course, but it is less important than finding parts of code that may be risky, especially if all the review time is spent debating on how a vague naming rule should be applied.

Finally, most of what I've said about code review can be valuable and should be used for reviewing the project's design as well. As you know, the earlier a problem is found, the easier and cheaper it is to fix.

Integration Testing

Although each unit should be tested separately, they must obviously work together when you integrate them. Therefore, it's important to test the components after they have been integrated.

I suggest that you schedule integration tests at specific points in time—perhaps once a week, perhaps as a nightly build with automatic integration—by automatically extracting the code from the version control system you use. If you don't schedule such testing, you risk getting into the situation I've been in several times. Say I'm about to integrate my code with a fellow programmer's. I go over to his or her office and ask if he or she is ready for integration. The response is that he or she needs another hour, and so I go back to my office and start fixing something. Two hours later, my fellow programmer comes to my office and asks me if I'm finished. I say that I need another hour, and so we continue like this for a week or more. Avoid this trap by scheduling integration tests, and stick to your schedule.

It's important to note that, as is the case in other levels of testing, if integration testing is not done until the end of the project, you are asking for trouble. Start doing integration tests as soon as the components take shape—they don't have to be finished. In fact, they don't even have to be programmed at all except for the interfaces and a minimal amount of fake code. This will allow for integration as early as possible.

Stress Testing

Yet another type of test that you definitely shouldn't put off until the end of the development cycle is stress testing. (In reality, it's very common that this isn't done until after the system has been shipped and the customer complains about performance and/or scalability problems.) A good approach is to make a proof of concept stress test for one or two key use cases as soon as the design is set. If you can't reach your throughput goals at that point, you probably won't for the other use cases either, so you'll have to rethink the design.

Stress testing is also a good opportunity to conduct recovery testing. What happens if I pull the plug? When does the system hit the wall, and is there a convenient interval between the expected load and the wall? Will the system continue working at peak when it hits the wall? Asking these questions and testing for answers now will save time later.

A few years ago, Microsoft hired Vertigo to create the sample application Fitch and Mather Stocks,[2] which simulates a stock trading Web site. The purpose of creating the sample application was not only to see how well an application built with Microsoft tools such as ASP, VB6, COM+, and SQL Server could scale, but also to show developers around the globe the best practices for how to write scalable applications with these tools. When Vertigo thought they were finished, they shipped the application to National Software Testing Labs (NSTL) for the initial stress tests. The stress testing was done with four IIS/COM+ servers and one database server. NSTL found that the application crashed at 16 simultaneous users. After some fine-tuning by Vertigo, the application could handle approximately 15,000 simultaneous users with less than one-second response time, still using the same hardware. In this specific case, most of the tuning effect was reached because Vertigo used a workaround for an NT bug, but that's not my point here. My point is that if you don't do early stress testing, you don't know what the limits of your application's scalability are. One bottleneck, and you're in trouble.

Regression Testing

Sound object-oriented and component-based design will, thanks to a high degree of information hiding, make it more possible to change systems without breaking other parts of the system. Still, as soon as you change anything in an application, there is a risk that something else will stop working. You have probably said something like, "The only thing I changed was X, but that doesn't have anything to do with this new problem" only to find that the change actually was the reason. The more encapsulated your system is, the less often you encounter a problem like this, but for every n^{th} change, you will create a problem. Therefore, when you change something, you should perform regression testing to see that nothing else has been tampered with. Most often, you have to decide to test only the closest parts of the code change; otherwise, it would take too long, at least if you have to test by hand. In my experience, regression testing is one of the most important forms of testing, but many programmers do not take much time to do it.

This brings us to the next section: support for automatic testing.

Support for Automatic Testing with a Standardized Test Bed

Do you know the most common reason developers give for why they haven't tested their components? Well, it depends on when they are asked. Early on in the project, they may say, "It's no use testing my own code. Someone else has to do it." If they are asked after the project is completed, they may say, "There wasn't enough time." I've used both of these responses myself at one point in time or another. And although I know it *is* more likely that someone else will find problems with my components than I will, on the other hand, if I leave easily detectable errors to the tester, he or she won't have time to find the more important, nasty problems. So of course, as a programmer, I must remove as many simple bugs as possible as early as possible. Perhaps the singlemost efficient way to achieve this is to add support for automatic testing to your components.

NOTE

Although in this chapter I discuss a custom solution for making testing more automatic, there are several third-party solutions that are very interesting too. Two common examples are the tools from Rational and Compuware.

When you build components, the final UI often isn't available, and it would take too long to use it for all your testing because it often takes several actions in the final UI before your component is used. A common approach is to build a dummy UI instead with 48 buttons that have

to be pressed in a certain order to test your component. Unfortunately, only the original developer will know how to use this dummy UI, and not even he or she will know how to use it after a couple months. Why not skip that specially built UI as a test driver and use a standardized test bed instead? It doesn't have to be very fancy. The important thing is that you have something that you (and your colleagues) can use. While you're building the components, you also write a test driver for each component, so the test suite for the test bed is being created too. Bugs will be found early on and the test suite can be used repeatedly. You will be most happy about it when you make a small change in a component and re-execute all the tests, just in case, and the tests tell you about a newly introduced bug.

Test Driver, Test Bed, and Test Suite

Before we get any further in our discussion, it's important you understand the meaning of the following terms:

- *Test driver*—The test driver is a program that executes some tests of one or more components or one or more stored procedures.
- *Test bed*—The test bed is a container or a controller that can execute the test drivers.
- *Test suite*—The test suite is a couple of test drivers that are related in some fashion.

Developers of the source code are usually the best persons to write a test driver for the component. This test driver should be able to handle as many cases as possible and should expose the most sensitive parts of the code in the component, something with which the original developers are familiar. The developers can also complete this faster than anybody else because they know the code inside out, and it will rarely be as quick or efficient to write the test driver as when the code itself is being written.

NOTE

When the tester knows about the implementation and doesn't just see the unit as a black box, it is commonly referred to as white-box testing. (The opposite is black-box testing.) If the developer of a component also writes the test driver, white-box testing is being used.

It's important to note that there is no silver bullet when it comes to testing. Creating test drivers according to some rules for all the components will not solve every problem. However,

doing this will increase the quality of your application if your organization isn't using something similar to it already. By using a test bed like the one I will discuss in this chapter, you will take care of development testing, integration testing, stress testing, and regression testing. The test drivers will also provide great documentation, showing you how the components should be used, for example. After all, an example is often the most helpful documentation.

NOTE

Remember this simple rule: The component or stored procedure isn't finished until there is a well-written and successfully executed test driver that follows the rules established by the standardized test bed.

Working with the Test Bed

Let's look at a simple test bed that I've used to test the code in this book. The test bed is called `JnskTest`. The general requirements that I had for the test bed were that it should

- Log the test results in a database
- Log who executed the tests, when they were started, and when they ended
- Be usable for testing stored procedures, .NET components, and components written in VB6
- Have test drivers that are possible to use (without logging) from another test tool that logs data on its own, such as Microsoft's Application Center Test 2000 (ACT)
- Make it possible to schedule execution, for example, after the nightly build

In this section, I will demonstrate how you can work with the test bed. Please note that the tool itself isn't really important; it is the principle I hope will inspire you or give you some code to start with that you could then expand. I also want to make it clear that the following few pages aren't typical of the book. I'm *not* going to fill the book with a lot of screenshots. I'm including these screenshots only as a means to communicate the ideas behind this tool.

NOTE

You'll find all the source code for this chapter on the book's Web site at `www.samspublishing.com`.

In the following sections, I'm going to show you each of the forms in the test bed application `JnskTest`. The first is the overview form, which is the main form. Keep this in mind when you

read through the explanation of the tool. The idea is that you create test drivers for your components and stored procedures. Those test drivers are orchestrated to test executions. The test executions are listed in the overview form.

The Overview Form for the Test Bed

The `JnskTest` tool is "operated" through the overview form, where test executions are listed. Here, you can set a filter to see only certain executions, such as those that were executed during a certain time interval or those that haven't been executed yet. Before the form shows, you are asked for ordinary login information and what database to use. Figure 3.1 shows how the overview might appear. Here you can see the test executions based on the current filter.

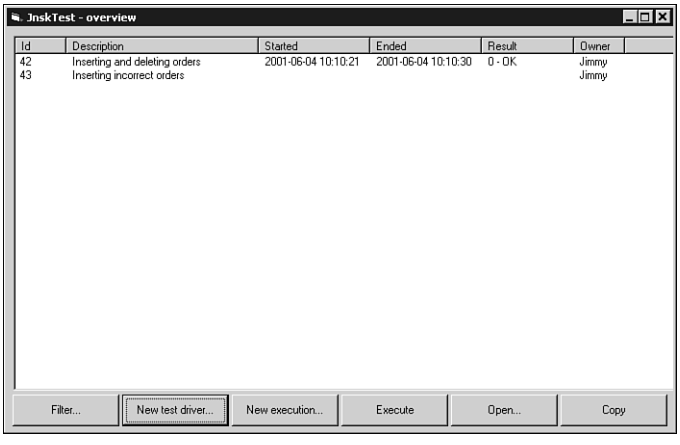

FIGURE 3.1

An overview of test executions based on the current filter.

The Form for Registering a Test Driver

Before you can get started, you must register your test drivers. (You also must have your test drivers written, but we will discuss that shortly.) It is possible for the application to autoregister test drivers by examining the database for specific suffixes among the stored procedures and checking the classes at the machine if they implement `Jnsk.Test.ITestDriver`. However, I have skipped that for now and instead will show you how to register the test drivers by hand. Figure 3.2 shows how the appropriate dialog appears. In the example, a stored procedure test driver is being registered.

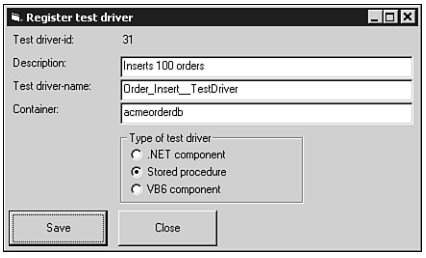

FIGURE 3.2
Registering a stored procedure test driver.

Registration and Execution

After you have registered your test drivers, you can register an execution by pointing out which test drivers to use. The registration of an execution is shown in Figure 3.3.

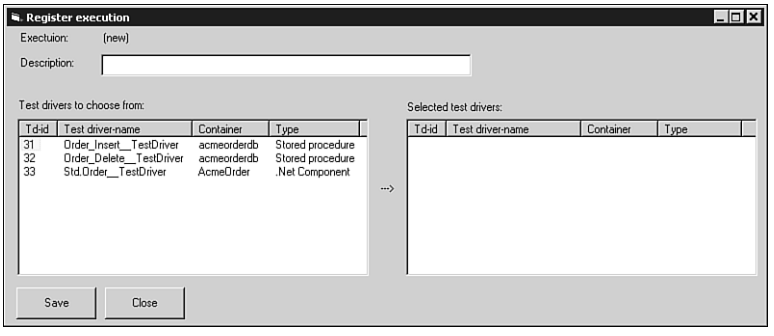

FIGURE 3.3
Registering the test drivers that should be used for an execution.

When you have registered one or more executions, you can execute them from the overview form (refer to Figure 3.1) by selecting the desired rows and clicking the Execute button. After a while, you will get the result of the execution. At this point, you should ask yourself how long the execution took and whether it worked out as you expected. You can then open each execution to get more details about the test driver used in the execution. Figure 3.4 shows the subsequent dialog in which you can add a comment to the result as documentation for the future.

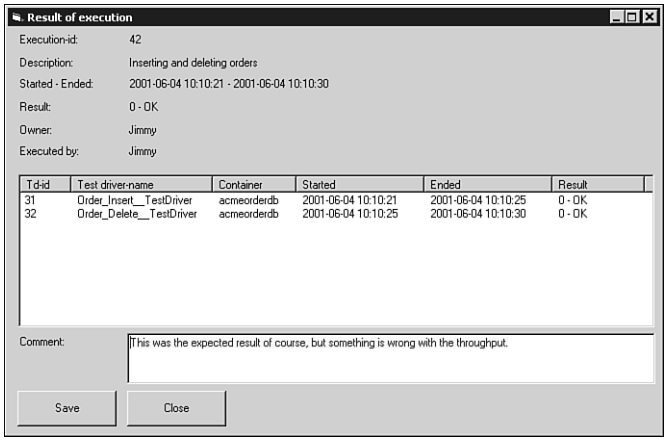

FIGURE 3.4
Viewing the result for each used test driver in a specific execution.

NOTE

The GUI calls stored procedures as the technique to access the database. The stored procedure interface can be used directly by the user of the test bed, as can the underlying tables (which will be described in the following section). For more information about this topic, you can download the tool from the book's Web site at `www.samspublishing.com`.

Understanding the Database Schema for the Test Bed

Whenever I design a component, I try to keep in mind the idea that simplicity is beautiful. You will find that the database schema I use for the test bed is quite simple. The tables are as follows:

- `jnsktest_td` In this table, each test driver (regardless of whether it is a test driver for a stored procedure or for a component) will be registered before it can be used.
- `jnsktest_execution` In this table, a certain execution request will be registered. A row in this table is the master to the detail rows in `jnsktest_executionrow`. After the test has been executed, you will find the overall result about the execution here.
- `jnsktest_executionrow` A row in this table carries information about one step of a certain execution request and what test driver to use. Afterward, you will find the result of each test driver used in the execution in this table.

The database schema just described is shown in Figure 3.5.

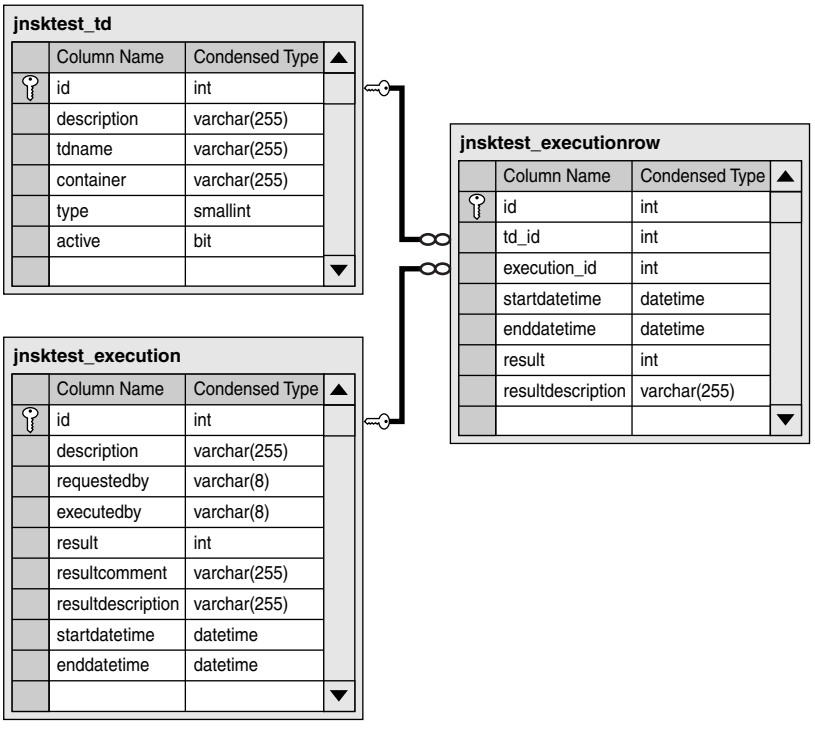

FIGURE 3.5
Database schema for test bed.

There are several points that need to be made about using the database for logging data. For one, I've asked myself more than once how long a certain scenario took in an earlier version. If I have the data about that easily accessible, I have an answer to this question. Another advantage is that it is easy to run statistic reports to determine, for example, whether the problem frequency is decreasing over time.

Using the Test Driver to Test Stored Procedures

It's simple to write a test driver to a stored procedure. Just write a stored procedure that returns 0 for success and another `INTEGER` value, typically the error code, for failure. (You should also use `RAISERROR()` to give some contextual information as text, such as exactly where the problem occurred.) Take care that you stop execution if you find a problem and return the error code. If you execute another test instead, there is a risk that the second test will be successful and you will never find out that there was a problem.

Listing 3.1 displays the first part of the code for a sample test driver. Some standard declarations of local variables are done first, followed by a few initializations. (This will be discussed in depth in Chapter 5, "Architecture.") After that, a specific local variable is declared, called `@anOrderId`, that will be used to receive the order ID of a newly added order.

LISTING 3.1 Example of a Test Driver for Stored Procedures: Part 1

```
CREATE PROCEDURE Order_InsertMaster__TestDriver AS
  DECLARE @theSource uddtSource
  , @anError INT
  , @anErrorMessage uddtErrorMessage
  , @aReturnValue INT

  SET NOCOUNT ON
  SET @anError = 0
  SET @anErrorMessage = ''
  SET @theSource = OBJECT_NAME(@@PROCID)

  ------------------------------------------------

  DECLARE @anOrderId INT
```


In the second part of the test driver, shown in Listing 3.2, the stored procedure to be tested is called. In this case, `@aReturnValue` of `0` is expected, meaning that an order registration for customer `42` at a certain date executed just fine. As you can see by the error-trapping model, the execution of the test driver will be ended by a `GOTO` to the `ExitHandler` if there is an error.

LISTING 3.2 Example of a Test Driver for Stored Procedures: Part 2

```
  EXEC @aReturnValue = Order_InsertMaster 42
  , '2001-04-30', @anOrderId OUTPUT

  SET @anError = @@ERROR
  IF @anError <> 0 OR @aReturnValue <> 0 BEGIN
    SET @anErrorMessage =
    'Problem with inserting order (1).'
    IF @anError = 0 BEGIN
      SET @anError = @aReturnValue
    END
    GOTO ExitHandler
  END
```

In the third part of the test driver, shown in Listing 3.3, the stored procedure to be tested (`Order_InsertMaster()`) is called again, but this time it is called with parameters that force an

error. If `@anError` equals to `8114`, everything is fine and it won't be treated as an error by the test driver. Otherwise, there was an unexpected error, which will be reported by the test driver.

LISTING 3.3 Example of a Test Driver for Stored Procedures: Part 3

```
EXEC @aReturnValue = Order_InsertMaster 42
, 'xx', @anOrderId OUTPUT
SET @anError = @@ERROR

IF @anError = 8114 BEGIN
  SET @anError = 0
END

IF @anError <> 0 OR @aReturnValue <> 0 BEGIN
  IF @anError = 0 BEGIN
    SET @anErrorMessage =
    'Problem with inserting order (2).'
    SET @anError = @aReturnValue
  END
  ELSE BEGIN
    SET @anErrorMessage =
    'Unexpected value for @@ERROR'
  END
  GOTO ExitHandler
END

------------------------------------------------
```

In the fourth part of the test driver, shown in Listing 3.4, you can see the `ExitHandler`. If an error has been found, it is reported through a centralized stored procedure called `JnskError_Raise()`. Finally, the stored procedure is ended with `RETURN()` of the error code.

NOTE

The error-trapping model will be discussed in detail in Chapter 9, "Error Handling and Concurrency Control."

LISTING 3.4 Example of a Test Driver for Stored Procedures: Part 4

```
ExitHandler:
  IF @anError <> 0 BEGIN
    EXEC JnskError_Raise @theSource
```

LISTING 3.4 Continued

```
    , @anError, @anErrorMessage
  END

  RETURN @anError
```

I suggest that you give your test driver the name of the stored procedure that is tested with a `__TestDriver` suffix. That way, you can easily see the stored procedures that lack a test driver.

NOTE

I originally had two versions of the controller for the execution: one for stored procedures and one for components. I have dropped that design and let the overview form in Figure 3.1 be the controller for all types of test drivers instead. This is easier and cleaner. To make it as easy as possible to write the test drivers, the controller will take care of all the logging.

Using the Test Driver to Test .NET Components

When you are testing .NET components with the test driver, you will use the same tables as were discussed earlier and shown in Figure 3.5. The test drivers will be Visual Basic .NET components, a sample of which is shown in Listing 3.5. In the sample, a method called `SalesOfToday()` is tested in an `Order` class.

LISTING 3.5 Example of a Component Test Driver

```
Public Class Order__TestDriver
    Implements Jnsk.Test.ITestDriver

    Public Sub Execute() _
    Implements Jnsk.Test.ITestDriver.Execute
        Dim anOrder As New Order()

        Try
            If anOrder.SalesOfToday() < 100000 Then
                Throw New ApplicationException _
                ("Strange SalesOfToday result.")
            End If

        Finally
            anOrder.Dispose()
```

LISTING 3.5 Continued

```
        End Try
    End Sub

End Class
```

As you can see, the test driver component must implement `Jnsk.Test.ITestDriver`, but except for this, there is nothing else special required. You can even skip `Try`, `Catch`, and `Finally` if you want, because the controller will take care of unhandled problems. If the test driver catches exceptions, it's important to `Throw()` an exception also, so it is popped to the controller.

Change of Style

For several years, I've been a fan of prefixing variables with data types and adding prefixes for the scope of variables and for `ByVal`/`ByRef` of parameters. For the .NET languages, Microsoft recommends camelCase (first letter lowercase, initial letter in the next word uppercase) as the naming convention for parameters—no data-type prefixes. (Microsoft actually only comments on public elements. Variables and so on aren't important in Microsoft's view of standardizing.)

Because VB6 did not have strong typing, data-type prefixes were very important. We now have strong typing at our disposal, so I typically add an `a`, `an`, or `the` prefix for variables. (For parameters, I follow Microsoft's recommendation.) For member variables, I use `m_` as the prefix instead. (I also use `btn` for buttons, and so on.) This is my choice, and you should follow the standards you think are appropriate. However, I strongly recommend standardizing public elements if you are creating a coding standard at your company, and it's probably a good idea to follow Microsoft's recommendation then.

I use the same convention for my parameters as for my stored procedures. In the past, I used the same data-type prefixes in my stored procedures as for my VB6 components. That was probably not good for database administrators, because it is not at all a standard in the T-SQL field.

NOTE

You could (and it's not uncommon) add your test driver code as a method to the class itself. Then you wrap that method with a conditional compilation directive so it's not in the release version of the code. I prefer *not* to use this solution because if I have

the test driver as a method in a separate class, I get the interface tested too because this question will be answered: Can the test driver call the other component and its methods? In the COM world, it has helped me on numerous occasions to have the test drivers code in separate classes.

Understanding the Controller of the Test Bed

The UI will act as the controller of the test bed (refer to Figure 3.1). The controller will call `JnskTest*` stored procedures to fetch information about test drivers, log what is going on, and save the final result. The same goes for calling the stored procedure test drivers. The technique that is used for calling component test drivers by name is found in the snippet from the controller code starting in Listing 3.6. In VB6, you could accomplish the same thing easier using `CreateObject()`, but if you use `CreateObject()` in Visual Basic .NET, you pass the interoperability layer and use the test driver as a COM object.

In Listing 3.6, the assembly with the test driver is loaded by using the name of the assembly for where the test driver is located (`container`), for example, `AcmeOrder`. Then, the classname (for example, *`Std`*`.Order__TestDriver`, where *`Std`* is the namespace) is used for getting the class.

LISTING 3.6 Sample Controller Code: Part 1

```
Dim anAssembly As System.Reflection.Assembly = _
System.Reflection.Assembly.Load _
 (container)


Dim aType As Type = _
anAssembly.GetType(testDriverName, True, True)
```

NOTE

The `testDriversAssembly` parameter may contain a fully specified reference to an assembly, for example,

```
AcmeOrder, Culture=neutral, PublicKeyToken=abc123..., Version=1.0.0.0
```

Then, as you see in Listing 3.7, the test driver class is instantiated and then cast to the `Jnsk.Test.ITestDriver` interface. Finally, the `Execute()` method is used to run the test scenario in the test driver.

LISTING 3.7 Sample Controller Code: Part 2

```
Dim aTestDriver As Jnsk.Test.ITestDriver = _
CType(Activator.CreateInstance(aType), _
Jnsk.Test.ITestDriver)

aTestDriver.Execute()
```

NOTE

This tool is fully usable for components written in VB6 as well as for components written in another .NET language, such as C#. You'll find test driver templates on the book's Web site at `www.samspublishing.com`.

Writing the Test Drivers: Issues to Consider

So far we have discussed how to use a test bed and write the test drivers. Now it's time to discuss some issues you should think about when you write test drivers to make them as useful as possible. But before we do that, I'd like to share a tip for choosing test cases in addition to the testing you do for each and every component and stored procedures. The use cases are a great source to use. If each and every part of your use cases—both the normal and those with expected exceptions—works fine, you have come a long way.

I've often heard that the part of the system that is most often full of errors is the error-trapping code. Although it is rarely used, it is still important that it works correctly. Make sure that you handle different error situations, including unexpected ones, when creating the error-trapping code.

In addition, you should set a goal for how much code should be executed, or covered, in the tests. Because you will write the test driver together with the component, it is the best opportunity to ensure such code coverage. You should also check it with a tool. (There will soon be several third-party tools to use with .NET components.) When it comes to stored procedures, I don't know of any streamlined tools to use. One proposal I came across was to add uniquely numbered trace calls after each line of code that can be executed.[3] That way, you can collect all the trace calls and calculate the percentage of the code that was visited.

NOTE

I will discuss tracing in depth in Chapter 4, "Adding Debugging Support."

It's also important that you test at the limits if there is an interval of possible values. Try the smallest value, the largest value, and the values just outside the interval. Another example of where it is important to test at the limits is `Arrays`. This is due to Microsoft's decision to let developers define the size of `Arrays` in Visual Basic .NET the same way that they defined them in VB6. I'm sure that will lead to bugs, especially among the many programmers who don't program exclusively in Visual Basic .NET, but also in C#.

You should be ruthless when you write test drivers. Remember that a good test driver is one that finds a problem, not one that says that everything seems to be running smoothly.

"Combinatorial bugs" are hardest to find in my opinion. By combinatorial bugs, I mean bugs that only happen when there is more than one circumstance occurring. Assume you have two variables, A and B. The bug is only occurring when A equals 10 and B equals 20, and never in any other situation. Finding that bug is, of course, much harder than if it occurs both when A equals 10 *or* B equals 20. Think about how to find such bugs when you write the test drivers so that you don't only test each case in isolation. When a bug is found, you should try to extend your test drivers so that you catch similar bugs the next time.

Test your test drivers with test drivers that you test with… Sorry, I got carried away there. What I mean is that you may want to introduce saboteurs that check that your tests work well. Insert some bugs in your stored procedures and components. In your components, I recommend you wrap the saboteurs within a specific compiler directive so that you don't forget to change the problem back. Unfortunately, a similar feature to compiler directives doesn't exist for stored procedures.

Test the transactional semantics as much as possible. Simulate problems in the middle of transactions and check that the system handles this gracefully. Also be sure to test concurrency situations. For example, you can add wait states in your components/stored procedures and execute two or more test drivers simultaneously to force the testing of concurrency situations.

Create fake components that your component is dependent on to make early testing possible. You can, of course, do the same with ADO.NET and `DataSets`. To make this productive, spend an hour or so writing a tool to generate the fake code because writing it by hand can be tedious. In sum, it's a good idea to achieve parallelism in your development project. For example, the UI group can immediately get components that return data (even if it is only fake data). When the components have real implementations, they are switched and the UI works just fine.

Write your test drivers (at least some of them) so that they can also be used for stress testing and not just for functional testing. That way, you can use the same test drivers in several different scenarios.

Finally, write a couple of generators that can create stubs for test drivers very quickly. That way, you can increase the productivity even further for the testing.

Discovering Problems with the Test Bed Proposal

Of course, test drivers do introduce some problems. One problem is that you will find yourself making code changes in two places—first in the code itself, and then in the test drivers. If you don't change the test drivers as the system evolves, they will quickly become obsolete.

Another problem is that in the kind of systems I build, I interact a lot with the database. Therefore, the tests, too, will be database-centric, so they must know something about what data to expect. If you delete all the rows in a table, for example, the test drivers will probably not work as expected. You could take proactive steps in the test drivers and add and delete rows to the database so it appears as you want it to before the real tests start, but then it will be more time consuming to write the test driver. I believe a better approach is to keep several instances of the database running, with one version specifically for the test suite. That way, most of the experiments will be run in other developer-specific databases instead, and the test database can be quite stable. You need to test your upgraded script anyway, so the test database will work fine for that. Just don't forget to make backups of the test database—it's as valuable as your development databases.

In the proposal for a test bed that I've given here, I haven't said anything about parameters to the test drivers. It might be a good idea to add support for this. That way you can, for example, let a test driver call another test driver with specific parameters to get a specific task done. When it comes to stress testing, it's important not to read the same piece of data over and over again (if this isn't a realistic situation). Otherwise, you will only hit the data that SQL Server is caching and you get an erroneous test result. This is yet another situation for which parameters could be handy, using a client that creates random values for the parameters. Of course, you could use a similar technique within the test drivers themselves to create random customer ID criterions, for example. In any case, it's important to add support for logging the exact values that are used so that a detected problem can be duplicated easily.

Finally, because the components and stored procedures will evolve, it could be wise to add some version information to the test results so that you don't compare apples and oranges. Otherwise, you will get strange comparisons when you see that a certain test driver took 1 second in one instance and 10 seconds in the next. How come? Even the test drivers will evolve, and, most probably, they will execute more tests over time. Thus, there are actually two version series to track.

NOTE

As you saw before, it's possible to give version information for the assembly of a test driver in the version of the test bed that I have discussed here.

Assertions

If you expect a variable at a certain place to have a value between 1 and 4, for example, you can express this in the code. In VB6, the `Debug` object has a method called `Assert()` that you can use to test your assumptions. Unfortunately, that method will not be compiled into the executable, and, in my opinion, this makes the method useless. Because of this, I wrote my own version of `Assert()` for my VB6 components.

The .NET Framework (and therefore Visual Basic .NET) has a new version of `Assert()` in the `System.Diagnostics.Debug` and `Trace` classes. (Use the `Trace` class if you want the calls to stay in the release version.) This new version is much better than the VB6 equivalent. You can use the .NET version, as shown in Listing 3.8. In this example, the `someParameter` parameter is expected to be different from zero. Otherwise, there is a bug.

LISTING 3.8 Example of How the Built-In `Assert()` Can Be Used

```
Trace.Assert(someParameter <> 0, "someParameter <> 0")
```

NOTE

There is another, security-related instance of the `Assert()` method in the Base Classes Library (BCL).

Although the built-in `Assert()` is quite good, I have written my own one as well. This is because

- *I want the tool to integrate with my tracing solution*—When you sit listening to your application (by using tracing), it's important to get broken assertions as trace calls too. (I will discuss my tracing solution in depth in the next chapter.)
- *I prefer a similar tool for both Visual Basic .NET components and stored procedures (and for VB6 components as well)*—Although this feature is not paramount in importance, it is useful.

- *I want a centralized solution that I can easily change*—For example, I like a broken assertion to terminate the application for the user and I want to have the problem logged. I also prefer to have it integrated with my own logging solution.

NOTE

For server-side development, using the built-in assertion solution can be a real show-stopper. You will get a `MsgBox()`, and the execution waits for somebody to accept the message. Because the message will pop up at the server, and typically at a virtual screen, nobody will press Enter.

John Robbins presents an ASP.NET-specific solution to this problem in his article called "Bugslayer: Handling Assertions in ASP.NET Web Apps."[4]

According to the documentation, you can also disable broken assertion signals to pop up, by adding a row to the `<switches>` section of the `.config` file as follows:

```
<assert assertuienabled="false" />
```

I will discuss `.config` files more in Chapter 4. In that chapter, I will also show a solution for how to handle configuration settings on-the-fly that comes in handy for my own assert solution.

Before I describe my solution for assertions in more detail, let's discuss the assertions in general. Although I spend a lot of time describing my own solution for assertions, the most important thing is that you use the concept somehow.

Getting the Basic Idea

Regardless of whether you use my assertion tool or the built-in one, you must consider where you should use assertions in your source code. I suggest you create a plan for when assertions will be used.

Before doing so, let's take a step back for a minute. In another programming language called Eiffel, created by Bertrand Meyer[5], a concept called design by contract (DBC) is used a great deal. The idea of DBC is to set up contracts between consumers and servers (a consumer uses a class and its methods; a server is the class). The server says what it expects (require) and what it promises (ensure). The consumer promises to fulfill the server's expectations; otherwise, the server doesn't have to fulfill its promises.

Assume that you are the consumer and the U.S. Post Office is the server. The contract between you and the Post Office is that you want them to deliver a certain item of mail to the correct person. For you to get that service, they expect to receive the mail before 5 p.m. that day, a valid stamp of the correct value on the letter, and the correct address on the envelope. If all this

is fulfilled, the mail will be delivered to the correct person, and, hopefully, on time. Although in reality you actually don't know when the mail will be delivered and don't really have any form of contract between you and the Post Office, this example expresses the general idea well.

What we want is to transfer this contract-based thinking to software construction. The software can check itself to see if any nasty bugs have been introduced, and the contracts, which are often implicit, will be made explicit to show which part is responsible for what. This way, you can avoid defensive programming. In defensive programming, everything is checked everywhere. Both the consumer and the server must check the value of the parameter. With DBC, the server can set a requirement and then expect that to be OK. Only the client has to perform a test; the server doesn't. That way the code will be robust but with far fewer lines of code, and the number of code lines is often directly correlated to the number of bugs.

NOTE

As a side note, Eiffel is going through a renaissance right now, at least when you consider how often it's mentioned in the Microsoft community compared to before. I guess that's because it's an example of a language that is CLR-compliant, but probably also because it's a very solid language.

To make the design-by-contract concept a little bit more concrete, a method called `Deliver()` in a `Mail` class is shown in Listing 3.9. There you can see that all the expectations of the method are checked before anything else is done. Then, before the method is exited, the promise is checked too.

LISTING 3.9 Example of How Assertions Can Be Used

```
Public Sub Deliver()
  'Require--------------------
  Trace.Assert(OnTime(), "OnTime()")
  Trace.Assert(CorrectAddress(), _
  "CorrectAddress()")
  Trace.Assert(CorrectStamp(),"CorrectStamp()")
  '---------------------------

  'Do the real delivery stuff
  '...

  'Ensure---------------------
  Trace.Assert(CorrectlyDelivered(), _
```

LISTING 3.9 Continued

```
  "CorrectlyDelivered()")
  '---------------------------
End Sub
```

You could naturally use ordinary `If` statements instead and `Throw()` exceptions when you find a problem, but the code in Listing 3.9 is cleaner. You also have the ability to disable the checks without deleting all the code lines. This is very important because one of the major positives with assertions is their value when you maintain the code. They will tell you if you make a change that breaks some other code.

Understanding Class Invariants

Bertrand Meyer's DBC includes a concept called "class invariants." If we take the U.S. Post Office example discussed earlier, we will find that there are "social" contracts and laws that say we're not allowed to send, for example, narcotics and explosives by mail. However, this does not mean that every time you send a letter you need to sign such a contract. Indeed, the process would be extremely long if we had to repeat those contracts and laws over and over again. The same goes for classes in software. If a class for a person has a member variable called `m_Age`, for example, the variable may never have a value of less than 0 and more than 125. This condition shouldn't have to be expressed repeatedly in all the methods of the class. Still, it should be checked in every Require and Ensure section to help discover any nasty bugs. How do we arrange for this? A possible solution would be to have a sub, as is shown in the class in Listing 3.10.

LISTING 3.10 Example of Class Invariants Helper

```
Private Sub AssertInvariants _
(ByVal source As String) _
Implements Jnsk.Instrumentation.IAssertInvariants.AssertInvariants
    'Call Assert() for each contract-part to check.
End Sub
```

The sub in Listing 3.10 should be called from the Require and Ensure sections in all the `Public` methods. You can add the calls through a utility to a copy of the code, or you could add it manually. The implementation of `AssertInvariants()` is a no-brainer—just call `Jnsk.Instrumentation.Assert.Assert()` once for every part of every social contract.

Most assertions in your code will be quite basic—for example, checking that a value is always in an interval or that there is a certain relationship between some variables. Sometimes, however, you need to create complete functions that return `True` and `False` to provide more

advanced logical expressions. I will discuss conditional compilation and similar techniques in the next chapter.

Examining My Solution for Assertions in Visual Basic .NET

As you have probably suspected by now, I centralize my assertions to `Jnsk.Instrumentation.Assert.Assert()` to get several positive benefits (such as those you usually get out of generalized code). The call then looks like that shown in Listing 3.11.

LISTING 3.11 Call to Customized `Assert()`

```
Jnsk.Instrumentation.Assert.Assert _
(a < b, "a < b", exeOrDllName, theSource)
```

My customized `Assert()` could just wrap the ordinary `System.Diagnostics.Trace.Assert()` or it could do customized work. (If I use the ordinary version, I will get one extra level in the call stack, but that is not such a big problem.) My current version appears in Listing 3.12.

LISTING 3.12 Customized `Assert()` Method

```
Public Class Assert
    Public Shared Sub Assert _
    (ByVal resultOfLogicalExpression As Boolean, _
    ByVal logicalExpression As String, _
    ByVal exeOrDllName As String,
    ByVal source As String, _
    ByVal userId As String, _
    ByVal raiseException As Boolean)
        If Not resultOfLogicalExpression Then
            Dim aMessage As String = _
            String.Format _
            ("Broken Assertion: {0} ({1}.{2}) For {3}.", _
            logicalExpression, _
            exeOrDllName, source, userId)


            'Send a trace call.
            Jnsk.Instrumentation.Trace.TraceAssert _
            exeOrDllName, source, logicalExpression)

            'Instantiate a custom exception.
            Dim anException As _
            New ApplicationException(aMessage)
```

LISTING 3.12 Continued

```
            'Log to applications own error log.
            Jnsk.Instrumentation.Err.Log _
            (exeOrDllName, source, anException, userId)
            'Log to the usual event log.
            EventLog.WriteEntry(source, aMessage, _
            EventLogEntryType.Error)

            'Show message box, depending upon
            'the current configuration.
            ShowMessageBox(aMessage)

            If raiseException Then
                Throw (anException)
            End If
        End If
    End Sub

End Class
```

As you see in Listing 3.12, the output from a broken assertion will be written to the event log, the log for the application, as a trace call, and as a `MsgBox()`. John Robbins created a solution that you could use to determine if the application is executed from the IDE or not.[6] The solution I have used instead uses the same solution for handling configuration data as is discussed in the next chapter.

NOTE

In Listing 3.12, several methods were used, such as `TraceAssert()` and `WriteToLog()`, which will be discussed in depth in Chapter 4.

Also note that I `Throw()` an `ApplicationException` instead of a custom `Exception`. The reason for this is that I don't want the consumer to have to reference my `Instrumentation` assembly (or another assembly with my custom `Exceptions`).

Finally, there is an overloaded version of the `Assert()` method that is used most often. It doesn't have the `raiseException` parameter, and it calls the method shown in Listing 3.12 with `raiseException` as `True`.

Examining My Solution for Assertions in Stored Procedures

Unfortunately, T-SQL doesn't have built-in support for assertions, but you can easily come up with a solution on your own. Listing 3.13 shows an example of an assertion being checked.

LISTING 3.13 Call to Stored Procedure Version of `Assert()`

```
IF NOT (@someParameter <> 0) BEGIN
  EXEC JnskAssert_Assert
  '@someParameter <> 0', @theSource
END
```

I expect that `@someParameter` will always be different from 0 at that particular point in the code. Therefore, I call my homegrown `JnskAssert_Assert()` stored procedure. As you know, I can't have the logical expression directly in the procedure call in T-SQL. Therefore, I have to evaluate the logical expression before I make the stored procedure call. Because of this, the assertion looks a bit unintuitive. Please observe that I use `NOT` in the `IF` clause. Doing it like this gives an assertion solution that doesn't require that many lines of code per assertion. The `JnskAssert_Assert()` procedure is shown in Listing 3.14.

LISTING 3.14 Stored Procedure Version of `Assert()`

```
CREATE PROCEDURE JnskAssert_Assert
(@logicalExpression VARCHAR(255)
, @source uddtSource
, @userId uddtUserId = '') AS

  DECLARE @anError INT
  , @aMessage VARCHAR(600)
  , @theNow DATETIME

  SET NOCOUNT ON

  --Trace the problem.
  EXEC JnskTrace_Assert @source
  , @logicalExpression, @userId

  --Log the problem.
  SET @anError = 99999
```

LISTING 3.14 Continued

```
SET @aMessage = 'Assert broken: '
+ @logicalExpression

SET @theNow = GETDATE()

EXEC JnskError_Log @source, @anError
, @aMessage, @theNow, 'unknown', 'unknown'
, @userId

--Finally, raise an error that closes the connection.
SET @aMessage = @aMessage + ' (' + @source + ')'
RAISERROR(@aMessage, 20, 1) WITH LOG

RETURN @anError
```

As you see in Listing 3.14, the same output will be used as for the Visual Basic .NET solution (except for a message box). The call to `RAISERROR()` will make broken assertions visible in the SQL Server Query Analyzer, and because severity level 20 is used, the connection is closed. The reason for that degree of violence is to directly terminate the operation, but also to make it easy to use calls to `JnskAssert_Assert()` because no error checking is necessary after the calls, as you saw in Listing 3.13.

Microsoft recommends that you check arguments for validity and raise `ArgumentException` (or a subclass exception). Listing 3.15 gives an example where a parameter called `size` isn't allowed to be less than zero.

LISTING 3.15 How to Check for an `ArgumentOutOfRangeException`

```
If size < 0 Then
    Throw New ArgumentOutOfRangeException _
    ("size must be >=0.")
End If
```

As I see it, this is defensive programming. Still, to comply with the standard, I do this in the methods that are entry points for the consumer. With methods in layers that are only internal to my own code, I use assertions instead of checking arguments. That seems to be a happy medium.

Creating Documentation

A bonus to the methods we have just discussed is that the contracts written in the code will provide great documentation. I've seen the use of contracts even without the automatic and built-in testing that I've discussed here, just because the contracts express much of the semantics of the classes and their methods. You can, for example, easily build a utility that drags out the method signatures together with the assertions to create one important part of the documentation of your application. This documentation is very valuable for public components.

The same goes for cooperation between different developers. Those contracts express responsibilities in a natural way. The assertions may also be a handy instrument at design time. They help you to be detailed about the responsibilities of the methods without writing all the code.

Checking for Traps

The most common trap is that your assert checks change the execution in any way. Be careful of this. A typical example is that the assert code moves the "cursor" in a `DataSet`, for example. Make sure you use a clone in that case. No matter what, there will be some minor differences when you use assertions and when you don't, because the code isn't the same. Just be careful that you're not affected.

Summing Up: Final Thoughts About Assertions

I mentioned earlier that I prefer the consumer application to terminate if an assertion has been broken. In this case, I don't want the application to continue with unpredictable and probably damaging results. To make termination happen, I `Throw()` an exception in the `Jnsk.Instrumentation.Assert.Assert()` method and I use `RAISERROR()` in `JnskAssert_Assert()` stored procedure with a severity level that closes the connection. In both cases, the consumer can continue to run, but hopefully, he won't want to continue when unexpected error codes are coming in.

Should the assertions be left in the compiled code so that they execute at runtime? There is a cost in overhead, of course. On the other hand, most often the cost is quite small, and it's great to have the system tell you when it's not feeling good and even what is wrong sometimes. Let's face it, most development projects are run in Internet time, and there aren't six months of testing before the components are rolled out. Why not distribute both a version with assertions active and one without them to the customer? You can also use a configuration solution, similar to the one I discuss in the next chapter, to activate and deactivate assertions on-the-fly. In the case of stored procedures, it's easy to toggle whether assertions should be active by just commenting the body of the `JnskAssert_Assert()` central routine. (You can also customize the behavior, of course. For example, you may want to do this if you don't want the connection to

be closed and no error to be raised, but want only the broken assertions to be logged.) Anyway, it's important to leave the assertions in the code. You will need them again!

NOTE

I will show you several real-world examples of assertions in the coming chapters. In addition, you can download the code samples from the book's Web site at `www.samspublishing.com`.

The Diagnose/Monitor Tool

Another tool that comes in handy when you have a problem that is hard to understand is a diagnosing tool. This can help you capture basic and essential information to get you on the right track.

Most developers I know have at least once spent hours thinking about a problem only to find, for example, that the customer has changed something vital in the environment. Or, perhaps you have had to ask a customer to check this and that before you start thinking about the problem. In my opinion, a much better approach is just to execute a diagnosing tool to collect all the interesting information to a file, which you can then analyze before you do anything else. Typical information to gather is

- Performance monitor metrics
- Items in the event log, SQL Server's error log, and your application's error log
- Version information for certain DLLs, OS, and so on
- Hardware information (for example, the number of CPUs)

Most of this information is quite easily caught with the help of `System.Diagnostics` namespace. You can also reuse some of the test drivers from the in-house testing that checks diagnostic matters. Why not ship a complete test database and all the test drivers to check the complete test suite at the customer location? It may also be wise to expand these thoughts a little, not only to execute the diagnosing tool to find information about the site after a problem has occurred, but also to monitor the site all the time. Log some of this information every hour or so. That way, you will easily see trends and can take action before a problem occurs.

Of course, this is a lengthy discussion, but hopefully I have whetted your appetite.

Miscellaneous Tips

Finally, the following are two additional tips for testing:

- Security issues are often "forgotten" by developers. The same applies to testing too. Spend some extra time on testing security issues and the deployment will run more smoothly.
- Perhaps the most important tip of all regarding testing is to have a similar testing environment—hardware, communication, size of database, and so on—to the one you will have in the real situation. A common problem is that the customer has a Symmetric MultiProcessing (SMP) machine—a machine with several processors—but you don't have that in your testing environment. It's common to find problems that only show up with an SMP machine. At the same time, it's unusual that companies have these kinds of machines in their testing environment. I can't say this enough: It is important to have a realistic testing environment. Period.

Evaluation of Proposals

Now it's time to wrap up this chapter by evaluating the proposals I have presented against the factors we discussed in Chapter 2. I hope it is clear that all the proposals are good in terms of testability because they all have this as their primary focus. Apart from that, let's dig into the evaluation of each of the proposals in more detail.

Evaluation of the Test Bed Proposal

The test bed will not directly improve the performance and scalability of your project. However, it won't decrease it either, because the test drivers are not in the binary that will be used by the application—the test drivers are separate units. On the other hand, you do get a tool that can help you to get on the right track and to find out bottlenecks in different situations. You can test to increase performance and scalability, and you can carry out the testing with good productivity.

The test bed is also good for easily checking how physical restrictions are affecting the application and to see that the application is working correctly, say, in a farm environment. Meanwhile, reliability is definitely addressed with the test bed because you can check a lot of "totally" impossible situations, for example. You can, and should, unplug network cables, the power supply, and such to see that the components act as expected.

In terms of productivity, it probably won't increase in the short run. In fact, it can actually suffer. But it will most often increase in the long run. The possible maintenance side effects are addressed greatly.

The test bed can help improve debuggability because it can be hard to set up a certain problem situation with only the application's ordinary UI. You can also use the test bed to localize a problem when you know that something has stopped working. I think even reusability could be improved because you can easily test to see how a component behaves in another context. Security problems can also be found by using the test bed. And when it comes to coexistence and interoperability, I've emphasized making the test bed possible to use for both COM and .NET components as well as for stored procedures.

Even if I'm only partially correct, can you afford not to give these ideas a try?

Evaluation of Assertions Proposal

As with the test bed, assertions will increase testability and debuggability a great deal. In a way, assertions are to finding bugs as the compiler is to finding syntax errors. I also think reusability will benefit because the assertions will tell you if the reused component doesn't work correctly in the new context.

The price for my solution to assertions is some added overhead. Interoperability with VB6 and stored procedures is addressed because the solution works in all three cases. (Actually, there isn't anything stopping this from being used with old ASP either.) Productivity can be both positive and negative, but in the long run, I believe it will be positive. Contracts as documentation will increase maintainability, and reliability is addressed by ending the scenario in case of broken assertions.

What's Next

In the next chapter, I will discuss a topic closely related to testing, namely debugging. When you have proved that something is broken with the help of testing, debugging will help you to determine what is wrong. I will also focus on how to prepare for debugging.

References

1. K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley; 1999.
2. `http://msdn.microsoft.com/library/techart/fm2kintro.htm`.
3. DMS Reengineering Toolkit at `http://www.semdesigns.com/Company/Publications/TestCoverage.pdf`.
4. J. Robbins. "Bugslayer: Handling Assertions in ASP.NET Web Apps," *MSDN Magazine*; October 2001; Microsoft.
5. B. Meyer. *Object-Oriented Software Construction, Second Edition*. Prentice Hall; 1997.
6. J. Robbins. "Bugslayer: Assertions and Tracing in .NET;" *MSDN Magazine*; February 2001; Microsoft.