1

Preliminary Concerns

The term "software bug" is a common term that even beginning computer users know to be a defect or imperfection in a software application. Software users have become accustomed to finding problems with software. Some problems have workarounds and are not severe, whereas others can be extremely problematic and, in some cases, costly. Sadly, as users, we have come to expect this from software. However, in recent years the quality of software has generally increased as software teams spend countless hours identifying and eliminating these problems before the software reaches the user. The process of identifying these bugs is known as *testing*.

There are many different types of testing that can be performed on your web applications, including functionality of the application, security, load/stress, compliance, and accessibility testing.

If you are new to testing, don't worry: we will explain the fundamental concepts and guide you to the correct actions you'll need to consider for each type of testing discipline. If you already have experience with testing applications, then this book will identify the key areas of the ASP.NET family and pair them with the correct approaches and the tools available to successfully test your web application.

This book is not intended to be the definitive guide to any particular type of testing, but a thorough overview of each type of web testing discipline. Its goal is to get you started using best practices and testing tools, and provide you with resources to master that particular testing discipline. It's our aim as authors to help the reader navigate to a section of this book and learn what and how they should be testing at any point in the development of a web-based application.

Although existing books cover different testing disciplines in depth, this book is unique because it applies today's best testing approaches to the ASP.NET family, including WebForms, ASP.NET MVC Framework, Web Services, Ajax, Silverlight, and ADO.NET Data Services, ensuring that the key technologies relevant today are able to be tested by the reader.

The History of Testing Tools

Tools for testing have been around for as long as developers have been writing code. In the early years of software development, however, there wasn't a clear distinction between testing and debugging. At the time, this model worked. Some argue that this model worked because the system was closed; most companies who needed software had the developers on staff to create and maintain the systems. Computer systems were not widespread, even though developers worked very closely with customers to deliver exactly what was required. In the years between 1970 and 1995, computer systems started becoming more popular, and the relationships between developers and customers became distant, often placing several layers of management between them.

> *What is the difference between debugging and testing you might ask?* Testing *is the process of finding defects in the software. A defect could be a missing feature, a feature that does not perform adequately, or a feature that is broken.* Debugging *is the process of first tracking down a bug in the software and then fixing it.*

Many of the tools developers used for testing in the early days were internal tools developed specifically for a particular project and oftentimes not reused. Developers began to see a need to create reusable tools that included the patterns they learned early on. Testing methodologies evolved and tools started to become standardized, due to this realization. In recent years, testing methodologies have become their own, very strict computer science discipline.

During the past 12 years, many tools have been developed to help make testing easier. However, it's essential to learn about the past and the tools we had previously before diving into the set of tools we have now. It's important to notice that the tools tend to evolve as the process evolves.

> *The term "debugging" was made popular by Admiral Grace Murray Hopper, a woman who was working on a Mark II computer at Harvard University in August 1945. When her colleagues discovered a moth stuck in a relay, and realized it was causing issues with the system, she made the comment that they were "debugging the system."*

The sUnit Testing Framework

It is said that imitation is the sincerest form of flattery; that said, most modern unit testing frames are derived from the principals set forth in the sUnit testing framework primary developed by Kent Beck in 1998. Below are just a small number of frameworks which have built upon Beck's original concept.

- **sUnit.** Created by Kent Beck for Small Talk, sUnit has become known as the "mother of testing frameworks." Many popular unit testing frameworks such as jUnit and nUnit are ports of sUnit. The key concepts of the sUnit testing framework were originally published in Chapter 30 of Kent Beck's *Guide to Better Smalltalk* (Cambridge University Press, 1998).
- **jUnit.** A port of sUnit for Java created by Kent Beck and Erich Gamma in late 1998. jUnit helped bring automated unit testing into the main stream.
- **nUnit.** In late 2000, all the great things about jUnit were ported to .NET allowing C# developers to write jUnit style unit tests against their C# code.
- **qUnit.** This is the unit test running for the jQuery Framework. In May 2008, qUnit was promoted to a top-level application in the jQuery project. qUnit allows web developers to run unit tests on JavaScript.

- **WCAT.** First included in the IIS 4 resource kit in 1998, the Web Capacity Analysis Tool (WCAT) is a freely distributed command-line tool that allows a server to be configured with agents to perform load/stress testing on websites.
- **Web Application Stress Tool.** In 1999, Microsoft released a free tool to create GUI browser stress tests. This tool recorded a browser session and scripted the actions into a Visual Basic 6 script that could be modified. Because the tool generated scripts that could be modified, many web developers used the tool not only for stress testing but modified the scripts for user interface functional testing.
- **Microsoft Application Center Test.** Included in Visual Studio 2001 Enterprise Edition, Microsoft ACT improved upon the Web Application Stress tool. Microsoft ACT provided a schema in which the tests could be distributed among agents for large-scale load testing.
- **Framework for Integrated Test (FIT).** Created by Ward Cunningham in 2002, FIT is a tool for automated customer tests. Examples of how the software should perform are provided by the customer, and automated Test fixtures are created by the developer. The goal of FIT is to help integrate the work of developers, customers, tests, and analysts.
- **Fitnesse.** Ported to .NET in 2006 by David Chelimsky and Mike Stockdale, Fitnesse combines a web server, Wiki, and the FIT software acceptance testing framework together. This provides an acceptance testing framework which allows users to define input that can be interpreted by a Test fixture allowing for non-technical users to write tests.
- **Watir.** In May 2002, the Web Application Testing in Ruby Watir (pronounced "Water"), a library to automate browser acceptance tests in Ruby, is released.
- **Selenium.** Selenium provides a suite of tools for automated user interface testing of web applications. In 2004, Jason Huggins of Thoughtworks created the core "JavaScriptTestRunner" mode for automation testing of a time and expense system.
- **WatiN.** In May 2006, Watir, the popular browser acceptance testing framework, is ported to .NET as the WatiN (pronounced "Watt in") project.
- **Visual Studio 2005 Test Edition.** In 2005, Microsoft released a version of Visual Studio that included a new unit testing framework created by Microsoft called MS Test. Along with this new unit testing framework, what was known previously as Microsoft Application Center Test (Microsoft ACT) was integrated into this version as Web Unit Tests and Web Load Tests.
- **Visual Studio 2008 Professional.** In 2008, the professional version of Visual Studio 2008 included MSTest.

Testing Terminology

As with many different aspects in programming, testing disciplines have their own unique vocabulary. However, because of the number of terms, the barrier to entry is high and can scare new developers. This section is intended to get the reader up to speed on some common terms that will be used throughout the remainder of this book. The terms shown next are only intended to be a brief explanation. Each term will be discussed thoroughly in their respective chapters.

- **Test.** A *test* is a systematic procedure to ensure that a particular unit of an application is working correctly.

- **Pass.** A *pass* indicates that everything is working correctly. When represented on a report or user interface (UI), it is represented as green.
- **Fail.** In the case of a *fail*, the functionality being tested has changed and as a result no longer works as expected. When represented on a report, this is represented as red.
- **xUnit.** *xUnit* refers to the various testing frameworks which were originally ported from sUnit. Tools such as jUnit, qUnit, and nUnit fall into the xUnit family.
- **Test Fixture.** *Test fixtures* refer to the state a test must be in before the test can be run. Test fixtures prepare any objects that need to be in place before the test is run. Fixtures ensure a known, repeatable state for the tests to be run in.
- **Test Driven Development (TDD).** Test Driven Development is an Agile Software Development process where a test for a procedure is created before the code is created.
- **Behavior Driven Development (BDD).** Building on top of the fundamentals of TDD, BDD aims to take more advantage of the design and documentation aspects of TDD to provide more value to the customer and business.
- **Test Double.** When we cannot, or choose not, to use a real component in unit tests, the object that is substituted for the real component is called a *test double*.
- **Stub.** A test *stub* is a specific type of test double. A stub is used when you need to replicate an object and control the output, but without verifying any interactions with the stub object for correctness. Many types of stubs exist, such as the responder, saboteur, temporary, procedural, and entity chain, which are discussed in more depth in Chapter 2.
- **Mock.** *Mock* objects are also a form of test double and work in a similar fashion to stub objects. Mocks are used to simulate the behavior of a complex object. Any interactions made with the mock object are verified for correctness, unlike stub objects. Mock objects are covered in depth in Chapter 2.
- **Fake.** *Fake* objects are yet another type of test doubles. Fake objects are similar to test stubs, but replace parts of the functionality with their own implementation to enable testing to be easier for the method.
- **Dummy Objects.** *Dummy* objects are used when methods require an object as part of their method or constructor. However, in this case the object is never used by the code under test. As such, a common dummy object is null.
- **Unit Test.** A *unit test* is a method used to verify that a small unit of source code is working properly. Unit tests should be independent of external resources such as databases and files. A unit is generally considered a method.
- **Developer Test.** This is another term for a unit test.
- **Integration Test.** This is similar to a unit test; however, instead of being an isolation unit, these test cross-application and system boundaries.
- **Functional Test.** *Functional tests* group units of work together to test an external requirement. Testing disciplines such as graphical user interface testing and performance testing are considered functional tests.
- **GUI Test.** *GUI tests* test the graphical user interface. GUI tests are considered functional tests. Applications are used to simulate users interacting with the system such as entering text into a field or clicking a button. Verifications are then made based on the response from the UI or system.

- **Customer Test.** This is another term for an acceptance test.
- **System Test.** The term *system test* is a term to indicate an "End To End" test of the system. System tests include unit testing, security testing, GUI testing, functional testing, acceptance testing, and accessibility testing.
- **Load Test.** A large amount of connections are made to the website to determine if it will scale correctly. This type of testing is to ensure that the website can handle the peak load expected when the website is used in production without any errors or failures.
- **Stress Test.** This is another name for a load test.
- **Performance Test.** *Performance testing* measures the response of a system in normal use and when it's placed under load. A common metric for Web Applications is Time To First Byte (TTFB) and Requests Per Second (RPS).
- **Acceptance Test.** This is a formal test to indicate if a function of a software project conforms to the specification the customer expects.
- **Black Box Test.** A *black box test* is a test created without knowing the internal workings of the feature being tested. The only information you have to base your tests on is the requirements.
- **White Box Test.** A *white box test* is a test created with knowledge of the inner workings of the code being tested. By using your internal knowledge of the system you can adapt the inputs you use to ensure high test coverage and correctness of the system.
- **Regression Test.** A *regression test* is a test created to ensure that existing functionality was working correctly previously and is still working as expected.

Testing Myths

Some developers are required to explain every development practice and tool they'll need to create a piece of software to their managers; it's this manager who will then decide if the practice or tool is prudent for use. These managers are often developers that have been promoted, and their focus is no longer on development but managing. Former developers do not always make for the best managers; many times they don't keep their development skills sharp, and they can sometimes deny the use of new techniques and tools just because it's not the way that they do things. These situations do not make sense and are often hard for developers to handle, especially junior developers who are very eager to learn the latest and greatest technology.

> *Unit testing frameworks have been mainstream for roughly 10 years, but still, many managers fight developers who ask to implement unit testing frameworks. This section explores some of the popular myths around testing and helps give advice to the developer who is having issues implementing a testing regiment into their organization.*

Testing Is Expensive

Frederick Brooks stated in his book of essays, *The Mythical Man-Month*, that "A bug found in the field can easily cost one thousand times more to resolve then one found in development."

If this is an argument that your manager uses, create a test to verify the functionality of the method that contains the bug and use it the next time the bug occurs — and time yourself. Then, write the fix

for the bug and time yourself again. In most cases, you'll find that it only takes a few minutes to write a test for the functionality and now your argument to your manager can be, "If I was permitted to spend X amount of time creating a test, the customer would have never encountered this bug." Most managers will pay more attention to your requests if you have data to back up your reasons for wanting to do something.

If you continue on the path of creating tests for your system, over time you will form a comprehensive set of test cases. These test cases can then be executed during the development of your system, allowing you to catch regression bugs earlier in the process. By having the tests catch these bugs, you will save time and money in terms of re-testing the application but also in maintaining the system in production.

Only Junior Developers Should Create Tests

This claim is very far from the truth. It's important for junior developers to write tests along with senior developers. This claim is often an excuse for a manager to stick a junior developer on a project just to create a bunch of test plans. However, the test plans and automated testing a junior developer creates is often useless, because of a lack of training and guidance. It's important for senior developers to work closely with junior developers to teach them what makes good tests. Testing is easy; *good* testing is hard. It's difficult to learn how to write good tests and create test plans from books; books help with the concepts, but nothing matches sitting down and pair programming with an experienced developer for the day.

If you are the senior developer and you notice this, take it upon yourself to help educate the junior developer. Ask the manager if you can be responsible for a portion of the tests and help educate the junior developer about the process. In the long run it will make your job easier having multiple people on a team that can perform the task well.

If you are the junior developer, your career is ultimately your responsibility. You should speak to the manager and request that a senior developer work with you for a portion of the tests. If the manager disagrees, take it upon yourself to get the training you need to perform the task. Start with reading as much about testing as you can, then try going to a local user group. Local user groups often have "hack nights," where they get together and write code, allowing you to learn with your peers.

Tests Cannot Be Created for Legacy Code

Testing legacy code is often more difficult, but it's not impossible. Often, legacy code has not been architected in a way that allows unit tests to be created, but in most scenarios functional or acceptance tests can be created for them. During the past few years, many patterns have emerged that make testing legacy code much easier. In the testing world, a code base that contains no tests is oftentimes referred to as legacy code. This means that if you're not writing tests as you write your code, all you are doing is simply creating legacy code from day one.

This myth is generally just another excuse for managers that are uneducated about testing patterns. It's a great idea to write tests for legacy code. Oftentimes the developer who wrote the code is not around to maintain it anymore and creating a test suite for the application will help a new developer learn about the app and give other developers a safety net if they need to make changes in the future.

Tests Are Only for Use with Agile Software Development

Unit testing and Test Driven Development are fundamental processes for XP and many other Agile Software Development methodologies. Just because one process uses a great tool doesn't mean it won't fit into your process.

If your manager doesn't like the word *Agile*, don't refer to the practice of creating unit tests as an Agile process. Just call it unit testing. As a web developer, you may be familiar with Lorem Ipsum. Lorem Ipsum is used as a placeholder for text and is from the Latin work of Cicero's De Finibus Bonorum et Malorum. Lorem Ipsum is intended to have no meaning because customers often focus on the text rather than the layout of the text. Use this trick on your manager. Some managers are not comfortable with Agile processes because of a lack of knowledge or misunderstandings about how the processors work .

Tests Have to Be Created Before the Code Is Written

Test Driven Development (TDD) is a process that we will explore briefly later in this book in Chapter 3, but for now all you need to know is that it's a process where a unit test for a given functionality is created before the code for the functionality is written.

For a TDD purist, this is not a myth and a test has to be created before the code is created. For someone who is new to testing, TDD can be hard. Before a developer thinks about getting into TDD they should first learn what makes a good unit test. In Chapter 4, we will explore what makes good tests and explore both unit testing and TDD.

- **It's hard to maintain tests.** As you begin creating tests for your code, your test suites often become quite large and unwieldy. If your tests are written poorly then they will be hard to maintain just as with other code. With the correct architecture and considerations in the correct place as described in this book, you should find that your tests and the overall system are easier to maintain.

Having a large suite of tests that fully test your application is an investment. It's your job as a developer to convince your manager of the importance of having a test suite that fully tests your application.

- **You can't automate user interface code.** In the past, creating code to automate user interface code has been difficult. Tools such as the MS Test Web Test and WatiN make automated user interface testing possible.

If a manager states that automated user interface testing is not possible, simply point them to the tools. Be aware, though, that automated user interface tests are often brittle and require more time for maintenance than unit tests. In Chapter 5 you will explore automated user interface testing. With the correct knowledge and time, automated user interface testing can be achieved.

- **Unit tests remove the need for manual testing.** Although unit tests will become a valuable resource for your system, they alone do not replace the need for the system to also be manually tested. Unit tests verify isolated blocks, but we need to ensure that the system works as a whole. Manual testing involves ensuring that the system works as expected when everything is deployed correctly. When manually testing the system, not only will you be ensuring that the UI is rendered correctly, but you will also be verifying that the application and UI are actually useable.

Although you can automate different parts of the application, it is not yet possible to automate so that the application can be understood by the end-user and is easy to use. This is a very important stage and has to

be performed manually. Without this stage you could create a perfectly working application, however, the end-user may never be able to use it because they simply don't understand the layout and process.

- **Unit testing is the latest development fad.** In the last two years, testing has become a very hot topic with new blogs appearing every few days featuring "experts" who preach that if you don't write tests you are writing shoddy code. With the quick rise in popularity, it's understandable why one would perceive this development practice as a fad. This simply is not true with many of the concepts being promoted today originating many years earlier.

As with the Testing Is Expensive myth, most managers listen to facts. Gather facts and present a strong case about why testing is important. In most situations, someone who is telling you that your code is shoddy because you don't practice a certain testing development practice is not someone you should trust in blind faith. The true experts understand why testing is important and will not try to force-feed it to you. They understand why code is written without tests and will help guide you in the right direction if you let them.

- **Developers only need to write unit tests.** Unit tests are great to have as they help you design code that is testable and the tests act as a safety net when making architectural changes. However, they should never be considered as a substitution to integration tests, functional tests, and acceptance tests. When a tester/developer understands the purpose of each type of test, thinking that only unit tests are needed will not be an issue.

If someone tells you this, you can assume they do not fully understand the differences between unit tests, functional tests, and acceptance tests. Explain to them the differences, what each type of test is, and where they apply.

- **Testing is easy.** Creating tests is very easy, but creating tests that are meaningful is another story. As with many other disciplines learned throughout your development career, the ability to create good tests will come with time and experience. One of the best ways to learn how to create good tests is to do pair programming with a developer who is experienced with creating tests. There are many schools of thought on pair programming: what it is and how best to perform this method of software development. The method I found that works best for me is to work together with another software developer on the same workstation with two monitors, two keyboards and two mice attached. One developer writes the test and the other implements the code to make the test pass. If one developer doesn't agree with exactly what the other is trying to accomplish, they are able to take control of the work station and explain their reasoning. I find that sitting face-to-face works best to keep conversation flowing.

If you have convinced your manager that creating tests is a good practice, it should not be very hard for you to convince them that you need to learn how to create good tests. Chapter 2 includes a section on what makes good tests. Also, there are many training classes put on by top-notch trainers to assist in this area.

Iterative Processes

Have you worked at a company that only performed employee performance reviews yearly? I'm sure most developers have, but during the yearly review have you had a manager say, "On June 13, 2008, you forgot to shut off the kanooter value, and every Friday since then you have not been shutting it off!" or something similar? I'm sure many developers have been in this situation. If the manager had indicated that you were doing something wrong you could have taken measures to correct the issue. This same

paradigm happens with software development. Project managers spend a great deal of time working with customers to define exactly what a piece of software should accomplish. However, it's common to find that after the project manager has completed collecting the requirements for an application, the customer doesn't see the application again until development is "finished." This is the reason many software projects fail. In the last few years, Agile Software Development processes have became very popular among developers. One of the key principles in Agile Software processes is iterative development.

Iterative development groups the development of new features together and delivers small chunks of a project to a client, rather than waiting a long time to deliver a very large monolithic system. Agile teams try to keep the iteration period low, perhaps two weeks to a month, and at the end of the iteration they always have something to show. Some attribute the success of Agile to short iterations and constant communication with clients.

Agile developers and managers often speak about reducing the cost of change. Cost of change simply put is this: the longer it takes you to find a defect the more expensive it will be to fix. The cost of change topic is discussed heavily in the project management world; some attribute low cost of change to a very close feedback loop with customers, managers, and developers.

Testing suites can provide many entry points for gaining different types of feedback. At a very low level, testing provides feedback from the code itself about its current state and whether or not it is working correctly. At higher levels, acceptance testing gives developers feedback if the system was developed to specification.

Why Should I Learn about Testing?

You're reading a book about testing for the web, so you're heading in the correct direction.

For Developers

Developers tend to be busy people, what with new technology trends emerging and customers constantly demanding new features. At times, developing software is a very stressful profession. Worrying if the new feature you added this morning is going to break existing functionality should not be something developers are spending time thinking about. Having a test suite that spans from unit tests to customer acceptance tests will help a developer sleep better at night, at peace in the knowledge that the software is functioning as designed.

A well-written test suite will provide the proverbial safety net. With this safety net in place developers are more willing to refactor (changing the internal structure of how code works without modifying its external behavior or functionality) sections of code, experiment more often when getting ready to add more functionality into a system, and be more Agile. The end result is an improved code base and fewer areas within your code base that you are scared to make changes to.

One aspect of unit testing that new developers to automated testing often overlook is the fact that testable code in most circumstances is also well-designed code. Design patterns such as Inversion of Control (IoC), and simple development concepts such as Don't Repeat Yourself (DRY), together with a strict separation of concerns are needed to create testable code (concepts that will be discussed thoroughly in Chapter 2).

A developer who is experienced in writing tests will find that they are more productive actually writing a test as opposed to using the debugger to nail down an issue. Let's take a step back for a moment and think about this. How much time is wasted each day waiting for the browser to load a page just to test a simple change?

> *An informal study conducted by Web Ascender (*`http://www.webascender.com`*) found that web developers spend 15 minutes per day on average just waiting for web pages to load.*

By investing time in load tests, it will enable developers to choose the correct server infrastructure such as web farms and caching services earlier in the lifecycle. This information is also useful to plan for fiscal year budgeting. Loading tests that are run frequently and appropriately defined will help identify bottlenecks and negate negative customer feedback of a poorly performing website.

The concept of acceptance testing reinforces the importance of communication between developers and customers. The end result of acceptance testing ensures that the customer was delivered what they expected for the system being developed. In most cases, many customers are just happy that a developer cares that they are delivering quality software that is on par with what the customer wanted.

Many web developers are unaware of the accessibility laws that their site must conform to. Countries such as the United States have laws that forbid discrimination against people with disabilities. Accessibility is not just for government websites and in many cases simple accessibility testing could have avoided lawsuits .

For Managers

Different projects and different managers will have different criteria for what they feel makes a project successful. Good managers will see the necessity for each discipline of testing discussed in this book.

- **A project should be delivered on time.** Many projects wait until the last possible minute to test an application to ensure it is working properly. One of us has been in a situation where waiting to the last minute to test has caused a project to be delayed because of performance, accessibility, and acceptance issues, all of which could have been avoided if testing was completed through the development process.

  This may make sense to many managers; surely you can't test until the system is completed? Plenty of projects have been delivered to customers with very little customer interaction.

- **A project should be of high quality.** Managers should put a great deal of time into defining metrics to measure if a project is of high quality. Having a suite of tests is a very strong metric to measure the quality of software. Having a full range of tests means that code is "exercised" often and has a far less chance of breaking because of neglect. How are you currently measuring the quality of the software being delivered? Many managers take the approach, "If it ain't broke, don't fix it," but in reality how do they know it's not broken?

- **A project should have low risks.** Managers often speak about the risks of projects. Great managers will manage risks to rewards and only take on projects that they know their team will have success with. In the real world, there is money to be made on projects that are higher risk and many times the highest risk is the delivery date.

  Having a range of testing will help lower project risk by ensuring software works correctly conforms to specifications; and in many cases, testing helps keep the customer in the loop of what is going on with a project.

- **Have a happy client.** A common measure of success is a happy client. Most clients are very happy when you spend the time with them to make sure you are developing the system they want, not the system you or a manager/developer want. Getting the customer involved early on in the testing is key. Customers should be involved in acceptance testing and can also be leveraged in running manual test plans.

Where to Start

Getting the terminology down is the first hurdle. In the following chapters we will discuss the terminology and different types of testing disciplines. Where applicable, templates will be provided to help get you started. The best way to start testing is to jump in headfirst. Pick a project you're familiar with and create some tests.

Generally, most companies use a common library or framework for various projects. These frameworks take the form of logging, data access, message handling, and so on. A good way to learn how to write good unit tests is to start unit testing this library. You may see that the library you thought had a good architecture is not as easy to test as you expected. Nowadays, most of these frameworks also include their own unit tests that you can use as reference material. Another great way to start is by looking at other peoples' tests with open source projects being a goldmine of useful information. By seeing how other people created tests for these projects, you will be able to pick up some tricks to help you when you start writing tests.

Getting Started with Test Driven Development

Before you dive into TDD, you should spend a great deal of time working with code that was not designed to be tested via automation. Focus on techniques and patterns to write good tests for the code you are able to test. Slowly start to refactor code that is not testable into new abstractions that are testable. Spend time learning how to break dependencies. When you have hit the point where you understand how to abstract code out to make it testable, you will then see that Test Driven Development is a horrible name for the development and it should be named something such as Well Designed Code That Can Be Tested Driven Development (WDCTCBTDD).

Test Driven Development will be covered in more depth in Chapter 2, but readers should understand it's very difficult to decide they are going to practice TDD when they don't have the fundamental testing and software design skills in place.

> *A study commissioned in 2002 by the U.S. Department of Commerce's National Institute of Standards and Technology, found software bugs, or errors, are so prevalent and so detrimental that they cost the U.S. economy an estimated $59.5 billion annually, or about 0.6 percent of the gross domestic product.*

When Should Testing Be Automated?

The discussion about whether and when testing should be automated has been going back and forth for many years. When considering this argument, it is important to remember that testing actually covers a number of different techniques, and as such, this question cannot result in a simple yes or no answer. Instead, you need to look at every feature and type of testing which needs to be performed and decide on a case-by-case basis. But how do you decide?

First, you need to consider what you are asking. You are trying to decide if a test you need to perform should be done using script\code and made into an automatable, repeatable test or if you want to manually perform the test each time you change something. You then need to consider the type of testing you want to perform and consider the cost of automating the process as opposed to the cost of repeating the test. You need to consider the cost of maintaining the automated test compared to our manual test. Finally, you need to think about the cost of it going wrong and a bug ending up in production. With these factors considered, you can decide if you think the cost of automating is worth the cost and effort compared to manual testing.

With that in mind, let's consider some initial problem domains and the general thought process when faced with the question of whether or not to automate.

When unit testing, our main focus will be on the business logic and process. Arguably, this is the most important part of any application as this defines what happens and how it will happen. If this logic is incorrect, it can cause a lot of pain because it generally will only appear in production. We have already discussed that bugs found in production are the most expensive to fix. Having the production that the unit tests in place provide against bugs making their way into production is definitively an important factor. With business logic, if you have designed your system in a testable fashion then you should find that the tests are easy to write and maintain. If you need to make changes to the underlying logic due to the design of the system being testable, then the cost of maintaining the costs should be minimal, especially when combined with the lower cost of ownership of the core system.

Though your automated tests will verify the business logic is correct, you could also manually test this. However, there are a few major blockers to this idea. Firstly, to test the business logic you would need to go via a UI, either the main applications or the custom wrote UI. This has two possible problems: the main application might have bugs, which blocks you from testing the logic. Or, it hasn't been written yet. Combine this with the fact that after every code change you would need to manually re-enter all these values and edge cases into the UI and verify the result was as you expected based on either a specification or previous experience. As far as we're concerned, this makes automating the business logic essential.

The next major parts of most applications are the data access layer and interacting with a database. The levels of interaction with a database can vary greatly. For example, you could be using the database as a simple data store which your application will pull all the information required from and process within the business logic. The other case is that all your business logic actually takes place within stored procedures and your application simply pulls the data and renders it on-screen. In either case, you have similar issues to what you had with the business logic. If you were to manually test the database, you would need to write some custom queries or UI to execute your logic which is expensive in itself. If you were to automate you could insert the required test data automatically and test your logic, and then verify the results in a more effective fashion.

However, database tests have a series of problems. For starters, the tests are a lot slower than if they were simply running in memory, something that we'll discuss in more depth in Chapter 4. Secondly, if our database schema changes, the tests are more likely to break without re-factoring tools to support us as with business logic. Even with this additional overhead, automating your database tests generally still have a lower cost than manual testing.

The final level is the GUI. This is the area that causes most debate about whether it should be automated or not. The GUI has some interesting problems when it comes to automated testing which we will dive into in Chapter 5. The main problem is that when you are automating the UI you are verifying for correctness; however, because you are already testing the business logic, you are actually only verifying the behavior of the UI. Verifying this behavior is an expensive process because you are trying to replicate the UI's logic as steps within the test code. Although it is possible and certainly can lead to success, it can be expensive. If the behavior changes then it means the tests also need to be manually updated.

If you compare this to manual testing, then you are still performing the behavior verification, and, as before, are also performing usability testing, — verifying the rendering is correct — and checking text for spellings. All this is very expensive to automate, and as such, doing it manually could prove to be more effective as you also generally need to perform it less. There is some benefit to automating the UI behavior tests; it can be made more effective and more appealing, which we will discuss in Chapter 5.

But remember, if you don't automate your tests then you will manually need to perform each test for each release. If you think this will be a painful and costly process, then automate.

Introducing the ASP.NET Family

When ASP.NET was released in 2002, there was not much of a "family" to speak of. During the past six years, many new components have been developed under the ASP.NET product umbrella and warrant a brief discussion. Throughout this book, we will apply the testing topics and tools for each type of testing discussed in this book — many of the products in the ASP.NET family.

- **ASP.NET WebForms.** When people think of ASP.NET, developers think of WebForms. WebForms consist of an ASPX file, designer file, and a code-behind file. The majority of ASP.NET applications are built using this technology. In fact, WebForms is the View Rendering template engine which lives on top of the core ASP.NET runtime built using WebForms.
- **ASP.NET MVC.** The ASP.NET MVC Framework is an implementation of the Model View Controller (MVC) architectural pattern that isolates business logic, data logic, and user interface logic. In the past, Microsoft developers have thought of this type of architecture as layers, but MVC is more about components than layers.
- **Silverlight.** In 2007, Microsoft released Silverlight, which is a cross-platform/browser, multimedia browser platform for developing Rich Internet Applications (RIA). The user interface is created using the XAML markup language similar to Windows Presentation Foundation (WPF).
- **ASP.NET Web Services.** ASP.NET Web Services are components installed on a web server that allow applications to make HTTP requests to the component and receive data back in a standard format such as XML or SOAP.
- **ASP.NET AJAX Framework.** Included with Visual Studio 2008 by default, and Visual Studio 2005 as a standalone extension, Microsoft has provided ASP developers with an AJAX framework to aid in asynchronous JavaScript calls .
- **ADO.NET Data Services.** Code named Astoria, ADO.NET Data Services is a framework of patterns and libraries that enable creation and consumption of ADO data in the cloud (the Internet). ADO.NET data services interfaces to data with well-known formats such as JSON and XML.

In 1997, soon after Internet Information Systems 4 (IIS) shipped, Mark Anders and Scott Guthrie began work on the predecessor of ASP 3.0, what many developers now call "classic ASP." One of the main goals of this new project is to address separation of concern much better than the previous version of ASP. Through a few name changes (XSP and ASP+), what we now call ASP.NET 1.0 WebForms was released to the world on January 5, 2002.

Summary

The aim of this chapter is to provide you with some context to the book itself and provide you with an introduction into testing. We introduced you to the various testing terminologies, attempted to solve some of the testing myths, and provided you with some initial starting points to get you on your way.

In the next chapter we will focus more on developing the application and starting to think about the design and testability of applications to ensure they are testable.