Introduction

It has been said before but it bears repeating: Writing defect-free software is exceedingly difficult. Proof of correctness of real systems is still well beyond our abilities, and specification of behavior is equally challenging. Predicting future needs is a hit or miss affair—we'd all be getting rich on the stock market instead of building software systems if we were any good at it!

Automated verification of software behavior is one of the biggest advances in development methods in the last few decades. This very developer-friendly practice has huge benefits in terms of increasing productivity, improving quality, and keeping software from becoming brittle. The very fact that so many developers are now doing it of their own free will speaks for its effectiveness.

This chapter introduces the concept of test automation using a variety of tools (including **xUnit**), describes why you would do it, and explains what makes it difficult to do test automation well.

Feedback

Feedback is a very important element in many activities. Feedback tells us whether our actions are having the right effect. The sooner we get feedback, the more quickly we can react. A good example of this kind of feedback is the rumble strips now being ground into many highways between the main driving surface and the shoulders. Yes, driving off the shoulder gives us feedback that we have left the road. But getting feedback *earlier* (when our wheels first enter the shoulder) gives us more time to correct our course and reduces the likelihood that we will drive off the road at all.

Testing is all about getting feedback on software. For this reason, feedback is one of the essential elements of "agile" or "lean" software development. Having feedback loops in the development process is what gives us confidence in the software that we write. It lets us work more quickly and with less paranoia. It lets us focus on the new functionality we are adding by having the tests tell us whenever we break old functionality.

Testing

The traditional definition of "testing" comes from the world of quality assurance. We test software because we are sure it has bugs in it! So we test and we test and we test some more, until we cannot prove there are still bugs in the software. Traditionally, this testing occurs after the software is complete. As a result, it is a way of measuring quality—not a way of building quality into the product. In many organizations, testing is done by someone other than the software developers. The feedback provided by this kind of testing is very valuable, but it comes so late in the development cycle that its value is greatly diminished. It also has the nasty effect of extending the schedule as the problems found are sent back to development for rework, to be followed by another round of testing. So what kind of testing should software developers do to get feedback earlier?

Developer Testing

Rare is the software developer who believes he or she can write code that works "first time, every time." In fact, most of us are pleasantly surprised when something does work the first time. (I hope I am not shattering any illusions for the nondeveloper readers out there!)

So developers do testing, too. We want to prove to ourselves that the software works as we intended it to. Some developers might do their testing the same way as testers do it: by testing the whole system as a single entity. Most developers, however, prefer to test their software unit by unit. The "units" may be larger-grained components or they may be individual classes, methods, or functions. The key thing that distinguishes these tests from the ones that the testers write is that the units being tested are a consequence of the design of the software, rather than being a direct translation of the requirements.[1]

Automated Testing

Automated testing has been around for several decades. When I worked on telephone switching systems at Nortel's R&D subsidiary Bell-Northern Research in the early 1980s, we did automated regression and load testing of

[1] A small percentage of the **unit tests** may correspond directly to the **business logic** described in the requirements and the **customer tests**, but a large majority tests the code that surrounds the business logic.

the software/hardware that we were building. This testing was done primarily in the context of the "System Test" organization using specialized hardware and software that were programmed with test scripts. The test machines connected to the switch being tested as though it were a bunch of telephones and other telephone switches; it made telephone calls and exercised the myriad of telephone features. Of course, this automated testing infrastructure was not suitable for **unit testing**, nor was it generally available to the developers because of the huge amounts of hardware involved.

In the last decade, more general-purpose test automation tools have become available for testing applications through their user interfaces. Some of these tools use scripting languages to define the tests; the sexier tools rely on the "robot user" or "record and playback" metaphor for test automation. Unfortunately, many of the early experiences with these latter tools left the testers and test managers less than satisfied. The cause was high test maintenance costs caused by the "fragile test" problem.

The "Fragile Test" Problem

Test automation using commercial "record and playback" or "robot user" tools has gained a bad reputation among early users of these tools. Tests automated using this approach often fail for seemingly trivial reasons. It is important to understand the limitations of this style of test automation to avoid falling victim to the pitfalls commonly associated with it—namely, behavior sensitivity, interface sensitivity, data sensitivity, and context sensitivity.

Behavior Sensitivity

If the behavior of the system is changed (e.g., if the requirements are changed and the system is modified to meet the new requirements), any tests that exercise the modified functionality will most likely fail when replayed.[2] This is a basic reality of testing regardless of the test automation approach used. The real problem is that we often need to use that functionality to maneuver the system into the right state to start a test. As a consequence, behavioral changes have a much larger impact on the testing process than one might expect.

[2] A change in behavior could occur because the system is doing something different or because it is doing the same thing with different timing or sequencing.

Interface Sensitivity

Testing the business logic inside the **system under test** (**SUT**) via the user interface is a bad idea. Even minor changes to the interface can cause tests to fail, even though a human user might say the test should still pass. Such unintended interface sensitivity is partly what gave test automation tools such a bad name in the past decade. Although the problem occurs regardless of which user interface technology is being used, it seems to be worse with some types of interfaces than with others. Graphical user interfaces (GUIs) are a particularly challenging way to interact with the business logic inside the system. The recent shift to Web-based (HTML) user interfaces has made some aspects of test automation easier but has introduced yet another problem because of the executable code needed within the HTML to provide a rich user experience.

Data Sensitivity

All tests assume some starting point, called the **test fixture**; this **test context** is sometimes called the "pre-conditions" or "before picture" of the test. Most commonly, this test fixture is defined in terms of data that is already in the system. If the data changes, the tests may fail unless great effort has been expended to make them insensitive to the data being used.

Context Sensitivity

The behavior of the system may be affected by the state of things outside the system. These external factors could include the states of devices (e.g., printers, servers), other applications, or even the system clock (e.g., the time and/or date of execution of the test). Any tests that are affected by this context will be difficult to repeat deterministically without getting control over the context.

Overcoming the Four Sensitivities

The four sensitivities exist regardless of which technology we use to automate the tests. Of course, some technologies give us ways to work around these sensitivities, while others force us down a particular path. The xUnit family of test automation frameworks gives us a large degree of control; we just have to learn how to use it effectively.

Uses of Automated Tests

Thus far, most of the discussion here has centered on regression testing of applications. This is a very valuable form of feedback when modifying existing applications because it helps us catch defects that we have introduced inadvertently.

Tests as Specification

A completely different use of automated testing is seen in **test-driven development** (**TDD**), which is one of the core practices of agile methods such as **eXtreme Programming**. This use of automated testing is more about specification of the behavior of the software yet to be written than it is about **regression testing**. The effectiveness of TDD comes from the way it lets us separate our thinking about software into two separate phases: what it should do, and how it should do it.

Hold on a minute! Don't the proponents of agile software development eschew waterfall-style development? Yes, indeed. Agilists prefer to design and build a system feature by feature, with working software being available at every step to prove that each feature works before they move on to develop the next feature. That does not mean we do not do design; it simply means we do "continuous design"! Taking this to the extreme results in "emergent design," where very little design is done upfront. But development does not have to be done that way. We *can* combine high-level design (or architecture) upfront with detailed design on a feature-by-feature basis. Either way, it can be useful to delay thinking about how to achieve the behavior of a specific class or method for a few minutes while we capture what that behavior should be in the form of an executable specification. After all, most of us have trouble concentrating on one thing at a time, let alone several things simultaneously.

Once we have finished writing the tests and verifying that they fail as expected, we can switch our perspective and focus on making them pass. The tests are now acting as a progress measurement. If we implement the functionality incrementally, we can see each test pass one by one as we write more code. As we work, we keep running all of the previously written tests as regression tests to make sure our changes have not had any unexpected side effects. This is where the true value of automated unit testing lies: in its ability to "pin down" the functionality of the SUT so that the functionality is not changed accidentally. That is what allows us to sleep well at night!

Test-Driven Development

Many books have been written recently on the topic of test-driven development, so this one will not devote a lot of space to that topic. This book focuses on what the code in the tests looks like, rather than when we wrote the tests. The closest we will get to talking about how the tests come into being is when we investigate **refactoring** of tests and learn how to refactor tests written using one pattern into tests that use a pattern with different characteristics.

I am trying to stay "development process agnostic" in this book because automated testing can help any team regardless of whether its members are doing TDD, **test-first development**, or **test-last development**. Also, once people learn how to automate tests in a "test last" environment, they are likely to be more inclined to experiment with a "test first" approach. Nevertheless, we do explore some parts of the development process because they affect how easily we can do test automation. There are two key aspects of this investigation: (1) the interplay between *Fully Automated Tests* (see page 26) and our development integration process and tools, and (2) the way in which the development process affects the testability of our designs.

Patterns

In preparing to write this book, I read a lot of conference papers and books on xUnit-based test automation. Not surprisingly, each author seems to have a particular area of interest and favorite techniques. While I do not always agree with their practices, I am always trying to understand *why* these authors do things a particular way and *when* it would be more appropriate to use their techniques than the ones I already use.

This level of understanding is one of the major differences between examples and prose that merely explain the "how to" of a technique and a pattern. A pattern helps readers understand the *why* behind the practice, allowing them to make intelligent choices between the alternative patterns and thereby avoid any unexpected nasty consequences in the future.

Software patterns have been around for a decade, so most readers should at least be aware of the concept. A pattern is a "solution to a recurring problem." Some problems are bigger than others and, therefore, too big to solve with a single pattern. That is where the **pattern language** comes into play; this collection (or grammar) of patterns leads the reader from an overall problem step by step to a detailed solution. In a pattern language, some of the patterns will necessarily be of higher levels of abstraction, while others will focus on lower-level details. To be useful, there must be linkages between the patterns so that we

can work our way down from the higher-level "strategy" patterns to the more detailed "design patterns" and the most detailed "coding idioms."

Patterns versus Principles versus Smells

This book includes three kinds of patterns. The most traditional kind of pattern is the "recurring solution to a common problem"; most of the patterns in this book fall into this general category. I do distinguish between three different levels:

- "Strategy"-level patterns have far-reaching consequences. The decision to use a *Shared Fixture* (page 317) rather than a *Fresh Fixture* (page 311) takes us down a very different path and leads to a different set of test design patterns. Each of the strategy patterns has its own write-up in the "Strategy Patterns" chapter in the reference section of the book.
- Test "design"-level patterns are used when developing tests for specific functionality. They focus on how we organize our test logic. An example that should be familiar to most readers is the *Mock Object* pattern (page 544). Each test design pattern has its own write-up and the patterns are grouped into chapters in the reference section of the book based on topics such as *Test Double* patterns.
- Test "coding idioms" describe different ways to code a specific test. Many of these are language specific; examples include using **block closures** for *Expected Exception Tests* (see *Test Method* on page 348) in Smalltalk and anonymous inner classes for Mock Objects in Java. Some, such as *Simple Success Test* (see *Test Method*), are fairly generic in that they have analogs in each language. These idioms are typically listed as implementation variations or examples within the write-up of a "test design pattern."

Often, several alternative patterns *could* be used at each level. Of course, I almost always have a preference for which patterns to use, but one person's "anti-pattern" may be another person's "best practice pattern." As a result, this book includes patterns that I do not necessarily advocate. It describes the advantages and disadvantages of each of those patterns, allowing readers to make informed decisions about their use. I have tried to provide linkages to those alternatives in each of the pattern descriptions as well as in the introductory narratives.

The nice thing about patterns is that they provide enough information to make an intelligent decision between several alternatives. The pattern we choose may be affected by the goals we have for test automation. The goals describe

desired outcomes of the test automation efforts. These goals are supported by a number of principles that codify a belief system about what makes automated tests "good." In this book, the goals of test automation are described in Chapter 3, *Goals of Test Automation*, and the principles are described in Chapter 5, *Principles of Test Automation*.

The final kind of pattern is more of an anti-pattern [AP]. These **test smells** describe recurring problems that our patterns help us address in terms of the symptoms we might observe and the root causes of those symptoms. **Code smells** were first popularized in Martin Fowler's book [Ref] and applied to xUnit-based testing as test smells in a paper presented at XP2001 [RTC]. The test smells are cross-referenced with the patterns that can be used to banish them as well as the patterns[3] that are more likely to lead to them.[4] In addition, the test smells are covered in depth in their own section: Part II, *The Test Smells*.

Pattern Form

This book includes my *descriptions* of patterns. The patterns themselves existed before I started cataloging them, by virtue of having been invented independently by at least three different **test automaters**. I took it upon myself to write them down as a way of making the knowledge more easily distributable. But to do so, I had to choose a pattern description form.

Pattern descriptions come in many shapes and sizes. Some have a very rigid structure defined by many headings that help the reader find the various sections. Others read more like literature but may be more difficult to use as a reference. Nevertheless, all patterns have a common core of information, however it is presented.

My Pattern Form

I have really enjoyed reading the works of Martin Fowler, and I attribute much of that enjoyment to the pattern form that he uses. As the saying goes, "Imitation is the sincerest form of flattery": I have copied his format shamelessly with only a few minor modifications.

The template begins with the problem statement, the summary statement, and a sketch. The italicized problem statement summarizes the core of the problem

---

[3] Some might want to call these patterns "anti-patterns." Just because a pattern often has negative consequences, it does not imply that the pattern is *always* bad. For this reason, I prefer not to call these anti-patterns; I just do not use them very often.

[4] In a few cases, there are even a pattern and a smell with similar names.

that the pattern addresses. It is often stated as a question: "How do we . . . ?" The boldface summary statement captures the essence of the pattern in one or two sentences, while the sketch provides a visual representation of the pattern. The untitled section of text immediately after the sketch summarizes why we might want to use the pattern in just a few sentences. It elaborates on the problem statement and includes both the "Problem" and "Context" sections from the traditional pattern template. A reader should be able to get a sense of whether he or she wants to read any further by skimming this section.

The next three sections provide the meat of the pattern. The "How It Works" section describes the essence of how the pattern is structured and what it is about. It also includes information about the "resulting context" when there are several ways to implement some important aspect of the pattern. This section corresponds to the "Solution" or "Therefore" sections of more traditional pattern forms. The "When to Use It" section describes the circumstances in which you should consider using the pattern. This section corresponds to the "Problem," "Forces," "Context," and "Related Patterns" sections of traditional pattern templates. It also includes information about the "Resulting Context," when this information might affect whether you would want to use this pattern. I also include any "test smells" that might suggest that you should use this pattern. The "Implementation Notes" section describes the nuts and bolts of how to implement the pattern. Subheadings within this section indicate key components of the pattern or variations in how the pattern can be implemented.

Most of the concrete patterns include three additional sections. The "Motivating Example" section provides examples of what the test code might have looked like before this pattern was applied. The section titled "Example: {Pattern Name}" shows what the test would look like after the pattern was applied. The "Refactoring Notes" section provides more detailed instructions on how to get from the "Motivating Example" to the "Example: {Pattern Name}."

If the pattern is written up elsewhere, the description may include a section titled "Further Reading." A "Known Uses" section appears when there is something particularly interesting about those applications. Most of these patterns have been seen in many systems, of course, so picking three uses to substantiate them would be arbitrary and meaningless.

Where a number of related techniques exist, they are often presented here as a single pattern with several variations. If the variations are different ways to implement the same fundamental pattern (namely, solving the same problem the same general way), the variations and the differences between them are listed in the "Implementation Notes" section. If the variations are primarily a different reason for using the pattern, the variations are listed in the "When to Use It" section.

Historical Patterns and Smells

I struggled mightily when trying to come up with a concise enough list of patterns and smells while still keeping historical names whenever possible. I often list the historical name as an alias for the pattern or smell. In some cases, it made more sense to consider the historical version of the pattern as a specific variation of a larger pattern. In such a case, I usually include the historical pattern as a named variation in the "Implementation Notes" section.

Many of the historical smells did not pass the "sniff test"—that is, the smell described a root cause rather than a symptom.[5] Where an historical test smell describes a cause and not a symptom, I have chosen to move it into the corresponding symptom-based smell as a special kind of variation titled "Cause." *Mystery Guest* (see *Obscure Test* on page 186) is a good example.

Referring to Patterns and Smells

I also struggled to come up with a good way to refer to patterns and smells, especially the historical ones. I wanted to be able to use both the historical names when appropriate and the new aggregate names, whichever was more appropriate. I also wanted the reader to be able to see which was which. In the online version of this book, hyperlinks were used for this purpose. For the printed version, however, I needed a way to represent this linkage as a page number annotation of the reference without cluttering up the entire text with references. The solution I landed on after several tries includes the page number where the pattern or smell can be found the first time it is referenced in a chapter, pattern, or smell. If the reference is to a pattern *variation* or the *cause* of a smell, I include the aggregate pattern or smell name the first time. Note how this second reference to the *Mystery Guest* cause of *Obscure Test* shows up without the smell name, whereas references to other causes of *Obscure Test* such as *Irrelevant Information* (see *Obscure Test*) include the aggregate smell name but not the page number.

Refactoring

Refactoring is a relatively new concept in software development. While people have always had a need to modify existing code, refactoring is a highly

[5] The "sniff test" is based on the diaper story in [Ref] wherein Kent Beck asks Grandma Beck, "How do I know that it is time to change the diaper?" "If it stinks, change it!" was her response. Smells are named based on the "stink," not the cause of the stink.

disciplined approach to changing the design *without* changing the behavior of the code. It goes hand-in-hand with automated testing because it is very difficult to do refactoring without having the safety net of automated tests to prove that you have not broken anything during your redesign.

Many of the modern integrated development environments (IDEs) have built-in support for refactoring. Most of them automate the refactoring steps of at least a few of the refactorings described in Martin Fowler's book [Ref]. Unfortunately, the tools do not tell us when or why we should use refactoring. We will have to get a copy of Martin's book for that! Another piece of mandatory reading on this topic is Joshua Kerievsky's book [RtP].

Refactoring tests differs a bit from refactoring production code because we do not have automated tests for our automated tests! If a test fails after a refactoring of the test, did the failure occur because we made a mistake during the refactoring? Just because a test passes after a test refactoring, can we be sure it will still fail when appropriate? To address this issue, many test refactorings are very conservative, "safe refactorings" that minimize the chance of introducing a change of behavior into the test. We also try to avoid having to do major refactorings of tests by adopting an appropriate test strategy, as described in Chapter 6, *Test Automation Strategy*.

This book focuses more on the target of the refactoring than on the mechanics of this endeavor. A short summary of the refactorings does appear in Appendix A, but the process of refactoring is not the primary focus of this book. The patterns themselves are new enough that we have not yet had time to agree on their names, content, or applicability, let alone reach consensus on the best way to refactor to them. A further complication is that there are potentially many starting points for each refactoring target (pattern), and attempting to provide detailed refactoring instructions would make this already large book much larger.

Assumptions

In writing this book, I assumed that the reader is somewhat familiar with object technology (also known as "object-oriented programming"); object technology seemed to be a prerequisite for automated unit testing to become popular. That does not mean we cannot perform testing in procedural or functional languages, but use of these languages may make it more challenging (or at least different).

Different people have different learning styles. Some need to start with the "big picture" abstractions and work down to "just enough" detail. Others can understand only the details and have no need for the "big picture." Some learn best by hearing or reading words; others need pictures to help them visualize

a concept. Still others learn programming concepts best by reading code. I've tried to accommodate all of these learning styles by providing a summary, a detailed description, code samples, and a picture wherever possible. These items should be Skippable Sections [PLOPD3] for those readers who won't benefit from that style of learning.

Terminology

This book brings together terminology from two different domains: software development and software testing. As a consequence, some terminology will inevitably be unfamiliar to some readers. Readers should refer to the glossary when they encounter any terms that they do not understand. I will, however, point out one or two terms here, because becoming familiar with these terms is essential to understanding most of the material in this book.

Testing Terminology

Software developers will probably find the term "system under test" (abbreviated throughout this book as SUT) unfamiliar. It is short for "whatever thing we are testing." When we are writing unit tests, the SUT is whatever class or method(s) we are testing; when we are writing customer tests, the SUT is probably the entire application (or at least a major subsystem of it).

Any part of the application or system we are building that is *not* included in the SUT may still be required to run our test because it is called by the SUT or because it sets up prerequisite data that the SUT will use as we exercise it. The former type of element is called a **depended-on component** (**DOC**), and both types are part of the test fixture. This is illustrated in Figure I.1.

Language-Specific xUnit Terminology

Although this book includes examples in a variety of languages and xUnit family members, **JUnit** figures prominently in this coverage. JUnit is the language and xUnit framework that most people are at least somewhat familiar with. Many of the translations of JUnit to other languages are relatively faithful ports, with only minor changes in class and method names needed to accommodate the differences in the underlying language. Where this isn't the case, Appendix B, *xUnit Terminology Cross-Reference,* often includes the appropriate mapping.

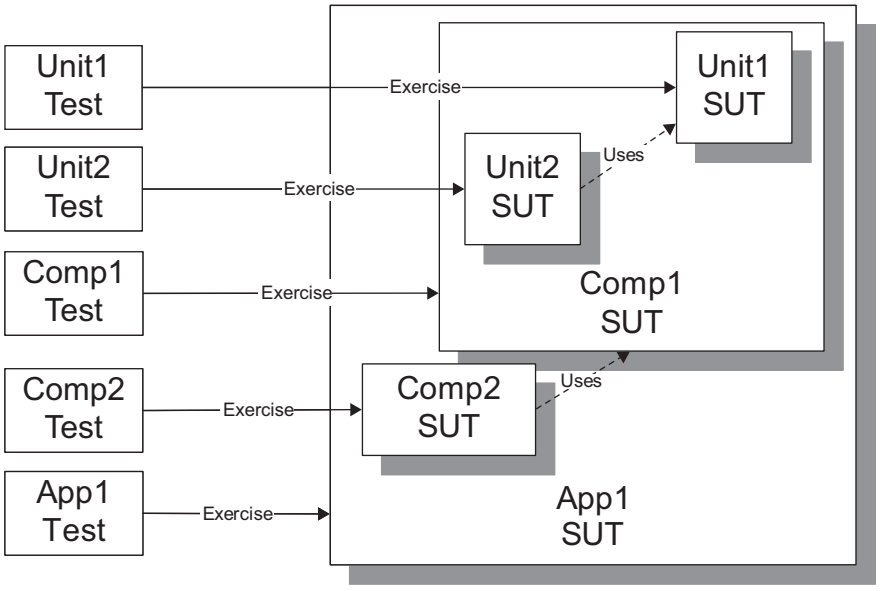

Figure I.1. *A range of tests each with its own SUT. An application, component, or unit is only the SUT with respect to a specific set of tests. The "Unit1 SUT" plays the role of DOC (part of the fixture) to "Unit2 Test" and is part of the "Comp1 SUT" and the "App1 SUT."*

Using Java as the main sample language also means that in some discussions we will refer to the JUnit name of a method and will not list the corresponding method names in each of the xUnit frameworks. For example, a discussion may refer to JUnit's `assertTrue` method without mentioning that the **NUnit** equivalent is `Assert.IsTrue`, the **SUnit** equivalent is `should:`, and the **VbUnit** equivalent is `verify`. Readers are expected to do the mental swap of method names to the SUnit, VbUnit, **Test::Unit**, and other equivalents with which they may be most familiar. The Intent-Revealing Names [SBPP] of the JUnit methods should be clear enough for the purposes of our discussion.

Code Samples

Sample code is always a problem. Samples of code from real projects are typically much too large to include and are usually covered by nondisclosure agreements that preclude their publication. "Toy programs" do not get much respect because "they aren't real." A book such as this one has little choice except to use "toy programs," but I have tried to make them as representative as possible of real projects.

Almost all of the code samples presented here came from "real" compilable and executable code, so they should not (knock on wood) contain any compile errors unless they were introduced during the editing process. Most of the Ruby examples come from the XML-based publishing system I used to prepare this book, while many of the Java and C# samples came from courseware that we use at ClearStream to teach these concepts to ClearStream's clients.

I have tried to use a variety of languages to illustrate the nearly universal application of the patterns across the members of the xUnit family. In some cases, the specific pattern dictated the use of language because of specific features of either the language or the xUnit family member. In other cases, the language was dictated by the availability of third-party extensions for a specific member of the xUnit family. Otherwise, the default language for examples is Java with some C# because most people have at least reading-level familiarity with them.

Formatting code for a book is a particular challenge due to the recommended line length of just 65 characters. I have taken some liberties in shortening variable and class names simply to reduce the number of lines that wrap. I've also invented some line-wrapping conventions to minimize the vertical size of these samples. You can take solace in the fact that your test code should look a lot "shorter" than mine because you have to wrap many fewer lines!

Diagramming Notation

"A picture is worth a thousand words." Wherever possible, I have tried to include a sketch of each pattern or smell. I've based the sketches loosely on the Unified Modeling Language (**UML**) but took a few liberties to make them more expressive. For example, I use the aggregation symbol (diamond) and the inheritance symbol (a triangle) of UML class diagrams, but I mix classes and objects on the same diagram along with associations and object interactions. Most of the notation is introduced in the patterns in Chapter 19, *xUnit Basics Patterns*, so you may find it worthwhile to skim this chapter just to look at the pictures.

Although I have tried to make this notation "discoverable" simply through comparing sketches, a few conventions are worth pointing out. Objects have shadows; classes and methods do not. Classes have square corners, in keeping with UML; methods have round corners. Large exclamation marks are **assertions** (potential **test failures**), and a starburst is an error or exception being raised. The fixture is a cloud, reflecting its nebulous nature, and any components the SUT depends on are superimposed on the cloud. Whatever the sketch is trying to illustrate is highlighted with heavier lines and darker shading. As a result, you should be able to compare two sketches of related concepts and quickly determine what is emphasized in each.

Limitations

As you use these patterns, please keep in mind that I could not have seen every test automation problem and every solution to every problem; there may well be other, possibly better, ways to solve some of these problems. These solutions are just the ones that have worked for me and for the people I have been communicating with. Accept everyone's advice with a grain of salt!

My hope is that these patterns will give you a starting point for writing good, robust automated tests. With luck, you will avoid many of the mistakes we made on our first attempts and will go on to invent even better ways of automating tests. I'd love to hear about them!