Preface

The Value of Self-Testing Code

In Chapter 4 of *Refactoring* [Ref], Martin Fowler writes:

> *If you look at how most programmers spend their time, you'll find that writing code is actually a small fraction. Some time is spent figuring out what ought to be going on, some time is spent designing, but most time is spent debugging. I'm sure every reader can remember long hours of debugging, often long into the night. Every programmer can tell a story of a bug that took a whole day (or more) to find. Fixing the bug is usually pretty quick, but finding it is a nightmare. And then when you do fix a bug, there's always a chance that anther one will appear and that you might not even notice it until much later. Then you spend ages finding that bug.*

Some software is very difficult to test manually. In these cases, we are often forced into writing test programs.

I recall a project I was working on in 1996. My task was to build an event framework that would let client software register for an event and be notified when some other software raised that event (the Observer [GOF] pattern). I could not think of a way to test this framework without writing some sample client software. I had about 20 different scenarios I needed to test, so I coded up each scenario with the requisite number of observers, events, and event raisers. At first, I logged what was occurring in the console and scanned it manually. This scanning became very tedious very quickly.

Being quite lazy, I naturally looked for an easier way to perform this testing. For each test I populated a `Dictionary` indexed by the expected event and the expected receiver of it with the name of the receiver as the value. When a particular receiver was notified of the event, it looked in the `Dictionary` for the entry indexed by itself and the event it had just received. If this entry existed, the receiver removed the entry. If it didn't, the receiver added the entry with an error message saying it was an unexpected event notification.

After running all the tests, the test program merely looked in the `Dictionary` and printed out its contents if it was not empty. As a result, running all of my tests had a nearly zero cost. The tests either passed quietly or spewed a list of test failures. I had unwittingly discovered the concept of a *Mock Object* (page 544) and a *Test Automation Framework* (page 298) out of necessity!

My First XP Project

In late 1999, I attended the OOPSLA conference, where I picked up a copy of Kent Beck's new book, *eXtreme Programming Explained* [XPE]. I was used to doing iterative and incremental development and already believed in the value of automated unit testing, although I had not tried to apply it universally. I had a lot of respect for Kent, whom I had known since the first PLoP[1] conference in 1994. For all these reasons, I decided that it was worth trying to apply eXtreme Programming on a ClearStream Consulting project. Shortly after OOPSLA, I was fortunate to come across a suitable project for trying out this development approach—namely, an add-on application that interacted with an existing database but had no user interface. The client was open to developing software in a different way.

We started doing eXtreme Programming "by the book" using pretty much all of the practices it recommended, including pair programming, collective ownership, and test-driven development. Of course, we encountered a few challenges in figuring out how to test some aspects of the behavior of the application, but we still managed to write tests for most of the code. Then, as the project progressed, I started to notice a disturbing trend: It was taking longer and longer to implement seemingly similar tasks.

I explained the problem to the developers and asked them to record on each task card how much time had been spent writing new tests, modifying existing tests, and writing the production code. Very quickly, a trend emerged. While the time spent writing new tests and writing the production code seemed to be staying more or less constant, the amount of time spent modifying existing tests was increasing and the developers' estimates were going up as a result. When a developer asked me to pair on a task and we spent 90% of the time modifying existing tests to accommodate a relatively minor change, I knew we had to change something, and soon!

When we analyzed the kinds of compile errors and test failures we were experiencing as we introduced the new functionality, we discovered that many of the tests were affected by changes to methods of the system under test (SUT). This came as no surprise, of course. What *was* surprising was that most of the impact was felt during the fixture setup part of the test and that the changes were not affecting the core logic of the tests.

This revelation was an important discovery because it showed us that we had the knowledge about how to create the objects of the SUT scattered across most of the tests. In other words, the tests knew too much about nonessential

[1] The Pattern Languages of Programs conference.

parts of the behavior of the SUT. I say "nonessential" because most of the affected tests did not care about *how* the objects in the fixture were created; they *were* interested in ensuring that those objects were in the correct state. Upon further examination, we found that many of the tests were creating identical or nearly identical objects in their test fixtures.

The obvious solution to this problem was to factor out this logic into a small set of *Test Utility Methods* (page 599). There were several variations:

- When we had a bunch of tests that needed identical objects, we simply created a method that returned that kind of object ready to use. We now call these *Creation Methods* (page 415).
- Some tests needed to specify different values for some attribute of the object. In these cases, we passed that attribute as a parameter to the *Parameterized Creation Method* (see *Creation Method*).
- Some tests wanted to create a malformed object to ensure that the SUT would reject it. Writing a separate *Parameterized Creation Method* for each attribute cluttered the signature of our *Test Helper* (page 643), so we created a valid object and then replaced the value of the *One Bad Attribute* (see *Derived Value on page* 718).

We had discovered what would become[2] our first test automation patterns.

Later, when tests started failing because the database did not like the fact that we were trying to insert another object with the same key that had a unique constraint, we added code to generate the unique key programmatically. We called this variant an *Anonymous Creation Method* (see *Creation Method*) to indicate the presence of this added behavior.

Identifying the problem that we now call a *Fragile Test* (page 239) was an important event on this project, and the subsequent definition of its solution patterns saved this project from possible failure. Without this discovery we would, at best, have abandoned the automated unit tests that we had already built. At worst, the tests would have reduced our productivity so much that we would have been unable to deliver on our commitments to the client. As it turned out, we were able to deliver what we had promised and with very good quality. Yes, the testers[3] still found bugs in our code because we were definitely missing some tests. Introducing the changes needed to fix those bugs, once we had figured

[2] Technically, they are not truly patterns until they have been discovered by three independent project teams.

[3] The testing function is sometimes referred to as "Quality Assurance." This usage is, strictly speaking, incorrect.

out what the missing tests needed to look like, was a relatively straightforward process, however.

We were hooked. Automated unit testing and test-driven development really did work, and we have been using them consistently ever since.

As we applied the practices and patterns on subsequent projects, we have run into new problems and challenges. In each case, we have "peeled the onion" to find the root cause and come up with ways to address it. As these techniques have matured, we have added them to our repertoire of techniques for automated unit testing.

We first described some of these patterns in a paper presented at XP2001. In discussions with other participants at that and subsequent conferences, we discovered that many of our peers were using the same or similar techniques. That elevated our methods from "practice" to "pattern" (a recurring solution to a recurring problem in a context). The first paper on test smells [RTC] was presented at the same conference, building on the concept of code smells first described in [Ref].

My Motivation

I am a great believer in the value of automated unit testing. I practiced software development without it for the better part of two decades, and I know that my professional life is much better with it than without it. I believe that the xUnit framework and the automated tests it enables are among the truly great advances in software development. I find it very frustrating when I see companies trying to adopt automated unit testing but being unsuccessful because of a lack of key information and skills.

As a software development consultant with ClearStream Consulting, I see a lot of projects. Sometimes I am called in early on a project to help clients make sure they "do things right." More often than not, however, I am called in when things are already off the rails. As a result, I see a lot of "worst practices" that result in test smells. If I am lucky and I am called early enough, I can help the client recover from the mistakes. If not, the client will likely muddle through less than satisfied with how TDD and automated unit testing worked—and the word goes out that automated unit testing is a waste of time.

In hindsight, most of these mistakes and best practices are easily avoidable given the right knowledge at the right time. But how do you obtain that knowledge without making the mistakes for yourself? At the risk of sounding self-serving, hiring someone who has the knowledge is the most time-efficient way of learning any new practice or technology. According to Gerry Weinberg's

"Law of Raspberry Jam" [SoC],[4] taking a course or reading a book is a much less effective (though less expensive) alternative. I hope that by writing down a lot of these mistakes and suggesting ways to avoid them, I can save you a lot of grief on your project, whether it is fully agile or just more agile than it has been in the past—the "Law of Raspberry Jam" not withstanding.

Who This Book Is For

I have written this book primarily for software developers (programmers, designers, and architects) who want to write better tests and for the managers and coaches who need to understand what the developers are doing and why the developers need to be cut enough slack so they can learn to do it even better! The focus here is on developer tests and customer tests that are automated using xUnit. In addition, some of the higher-level patterns apply to tests that are automated using technologies other than xUnit. Rick Mugridge and Ward Cunningham have written an excellent book on Fit [FitB], and they advocate many of the same practices.

Developers will likely want to read the book from cover to cover, but they should focus on skimming the reference chapters rather than trying to read them word for word. The emphasis should be on getting an overall idea of which patterns exist and how they work. Developers can then return to a particular pattern when the need for it arises. The first few elements (up to and include the "When to Use It" section) of each pattern should provide this overview.

Managers and coaches might prefer to focus on reading Part I, *The Narratives,* and perhaps Part II, *The Test Smells.* They might also need to read Chapter 18, *Test Strategy Patterns,* as these are decisions they need to understand and provide support to the developers as they work their way through these patterns. At a minimum, managers should read Chapter 3, *Goals of Test Automation.*

About the Cover Photo

Every book in the Martin Fowler Signature Series features a picture of a bridge on the cover. One of the thoughts I had when Martin Fowler asked if he could "steal me for his series" was "Which bridge should I put on the cover?" I thought about the ability of testing to avoid catastrophic failures of software

[4] The Law of Raspberry Jam: "The wider you spread it, the thinner it gets."

and how that related to bridges. Several famous bridge failures immediately came to mind, including "Galloping Gertie" (the Tacoma Narrows bridge) and the Iron Workers Memorial Bridge in Vancouver (named for the iron workers who died when a part of it collapsed during construction).

After further reflection, it just did not seem right to claim that testing might have prevented these failures, so I chose a bridge with a more personal connection. The picture on the cover shows the New River Gorge bridge in West Virginia. I first passed over and subsequently paddled under this bridge on a whitewater kayaking trip in the late 1980s. The style of the bridge is also relevant to this book's content: The complex arch structure underneath the bridge is largely hidden from those who use it to get to the other side of the gorge. The road deck is completely level and four lanes wide, resulting in a very smooth passage. In fact, at night it is quite possible to remain completely oblivious to the fact that one is thousands of feet above the valley floor. A good test automation infrastructure has the same effect: Writing tests is easy because most of the complexity lies hidden beneath the road bed.

Colophon

This book's manuscript was written using XML, which I published to HTML for previewing on my Web site. I edited the XML using Eclipse and the XML Buddy plug-in. The HTML was generated using a Ruby program that I first obtained from Martin Fowler and which I then evolved quite extensively as I evolved my custom markup language. Code samples were written, compiled, and executed in (mostly) Eclipse and were inserted into the HTML automatically by XML tag handlers (one of the main reasons for using Ruby instead of XSLT). This gave me the ability to "publish early, publish often" to the Web site. I could also generate a single Word or PDF document for reviewers from the source, although this required some manual steps.