Preface

We're all in the business of software development. Code is written and then deployed. Once we've deployed the code, our customers will express pleasure or, depressingly often, displeasure.

For the last few decades, much has been written about how to minimize this displeasure. We have countless languages, methodologies, tools, management techniques, and plain old-fashioned mythology to help address this issue.

Some of these approaches are more effective than others. There has certainly been a renewed emphasis and focus on testing lately, along with the pleasures said testing would bring to developers and users alike.

Much has been written extolling the virtues of testing. It can make your code better, faster, and lighter. It can add some sorely needed spice to the drudgery of coding. It's exciting and new (and therefore worth trying), not to mention the feeling of responsibility and accountability it imparts; there's something mysteriously satisfying about adding a feature and having a test to prove you did it right.

Unfortunately, religion has also crept into the science of testing. You won't have to look far to find holy commandments or Persons of Authority handing down instructions either applauding or scolding certain testing behavior.

This book attempts to distill some of the wisdom that has emerged over the last few years in the realm of Java testing. Neither of us has ever had a job where we're paid to sell testing, nor has testing been forced on us. Neither of us works at a place where one methodology has been proclaimed the "winner" and must be followed religiously.

Instead, we're pragmatic testers. Testing to us is simply another valuable tool that helps us as part of the software development cycle. We're not particularly "test infected," the term coined by JUnit early on that's gained so much adoption. We write tests when and where it makes sense; testing is a choice and not an infectious disease for us.

As a result of this approach, we've noticed a rather large hole in our testing arsenal: Very few tools seem to be practical and to lend themselves to the sort of tests we'd like to write. The dominant force in Java testing is JUnit, and in many cases, it's easy and intuitive to think of a test we'd like to run. The main obstacle, however, ends up being the tooling and its inability to capture concepts that are second nature to us in the code we'd like to test—concepts such as encapsulation, state sharing, scopes, and ordering.

JUnit, for all its flaws, really brought the concept of testing to the forefront. No longer was it an ad hoc approach. Instead, tests now had a framework and a measure of standardization applied. JUnit-based tests could be easily automated and replayed in a variety of environments using any number of visualization tools. This ease of use encouraged its massive adoption and the increased awareness of Java testing in general.

Its success has also spilled over to a number of other languages, with ports to other languages all based on the same underlying concepts.

As with any successful tool, however, the success came at a price. A subtle shift took place where instead of testing being the concern, and JUnit a tool to help achieve that, JUnit became the main focus, with testing that didn't fit in its narrow confines resulting in doubts about the test, rather than the tool.

Many will proclaim that a test that cannot be easily expressed in a simple "unit" is a flawed test. It's not a unit test since it has requirements beyond the simplistic ones that JUnit provides for. It's a functional test that happens later, after having built the unit building blocks. We find this argument perplexing, to say the least. Ultimately there is no one right way to do testing. It would be equally ridiculous to proclaim that development must start from implementing small units to completion first, before thinking of higher-level concerns. There are cases where that makes the most sense, just as there are many where it doesn't. Testing is a means to an end, and the end is better software. It's crucial to keep this in mind at all times.

Why Another Book about Testing?

This is a book about Java testing. Every chapter and section you will read in the following pages will discuss testing in some way or another. Regardless of what testing framework you use or whether you use tools that we don't cover, our goal is to show you some practices that have worked for us in some way. We also tried to draw general conclusions from our own experiences and use these to make recommendations for future scenarios that might arise.

Even though we use TestNG in this book to illustrate our ideas, we firmly believe that you will find some of it useful, whether or not you use JUnit—even if you're not programming on the Java platform. There are plenty of TestNG/JUnit-like frameworks for other languages (C# and C++ come to mind), and the ideas used in a testing framework are usually universal enough to transcend the implementation details that you will encounter here and there.

This book is about pragmatic testing. You will not find absolute statements, unfounded religious proclamations, and golden rules that guarantee robust code in this book. Instead, we always try to present pros and cons for every situation because ultimately you, the developer, are the one with the experience and the knowledge of the system you are working with. We can't help you with the specifics, but we can definitely show you various options for solving common problems and let you decide which one fits you best.

With that in mind, let's address the question asked above: Why another book about testing?

There are plenty of books (some very good) about Java testing, but when we tried to look more closely, we came to the conclusion that hardly any books covered a broad topic that we found very important to our day-to-day job: modern Java testing.

Yes, using the adjective *modern* in a book is dangerous because, by nature, books don't remain modern very long. We don't think this book will be the exception to this rule, but it is clear to us that current books on Java testing do not properly address the challenges that we, Java developers, face these days. As you can see in the table of contents, we cover a very broad range of frameworks, most of which have come into existence only in the last three years.

In our research on prior art, we also realized that most books on Java testing use JUnit, which, despite its qualities, is a testing framework that has barely evolved since its inception in 2001.[1] It's not just JUnit's age that we found limiting in certain ways but also its very design goal: JUnit is a unit testing framework. If you are trying to do more than unit testing with JUnit (e.g., testing the deployment of a real servlet in an application server), you are probably using the wrong tool for the job.

Finally, we also cover a few frameworks that are quite recent and are just beginning to be adopted (e.g., Guice) but that we believe have such a

1. JUnit 4, which came out in 2006, was the first update in five years that JUnit received, but at the time of writing, its adoption is still quite marginal, as most Java projects are still using JUnit 3.

potential and open so many doors when used with a modern testing framework such as TestNG that we just couldn't resist writing about them. Hopefully, our coverage of these bleeding-edge frameworks will convince you to give them a try as well.

Throughout the book, we have tried hard to demonstrate a pragmatic application of testing. Many patterns are captured in these pages. It's not an explicit list that we expect to be recited; rather, it's more of a group of examples to ensure you develop the right approach and way of thinking when it comes to testing code.

We achieve this through two separate approaches, the first of which is TestNG usage specifics. We discuss most of its features, explaining how and why they arose, as well as practical real-world examples of where they might be applicable. Through this discussion, we'll see how testing patterns can be captured by the framework and what goes into a robust maintainable test suite (and more importantly, what doesn't!).

The second aspect is showing how TestNG integrates with your existing code and the larger ecosystem of Java frameworks and libraries. Few of us are lucky enough to work on projects that are started completely from scratch. There are always components to reuse or integrate, legacy subsystems to invoke, or backward compatibility concerns to address. It would be equally foolish to demand redesigns and rewrites just to enable testing. Instead, we try to show how it's possible to work with existing code and how small incremental changes can make code more testable and more robust over time. Again, through this approach, a number of patterns emerge, along with more practices on how to write tests and approach testing in general.

We hope you enjoy reading this book as much as we enjoyed writing it. We feel very strongly about testing, but we feel equally strongly that it isn't a golden hammer in a world of nails. Despite what many would like to believe, there are no solutions or approaches that absolve you from the need to think and the need to understand your goals and ensure that your testing journey is a rational and well-considered one, where both the downsides and upsides have received equal consideration.

Audience

So, what is this book and who is it for? In short, it's for any Java developer who is interested in testing.

We are also targeting developers who have been testing their code for quite a while (with JUnit or any other framework) but still find themselves sometimes intimidated by the apparent complexity of their code and, by extension, by the amount of effort needed to test it. With the help of the TestNG community over these years, we have made a lot of progress in understanding some of the best practices in testing all kinds of Java code, and we hope that this book will capture enough of them that nobody will ever be stuck when confronted with a testing problem.

This book uses TestNG for its code samples, but don't let that intimidate you if you're not a TestNG user: A lot of these principles are very easy to adapt (or port) to the latest version of JUnit.

Whether you use TestNG or not, we hope that once you close this book, you will have learned new techniques for testing your Java code that you will be able to apply right away.