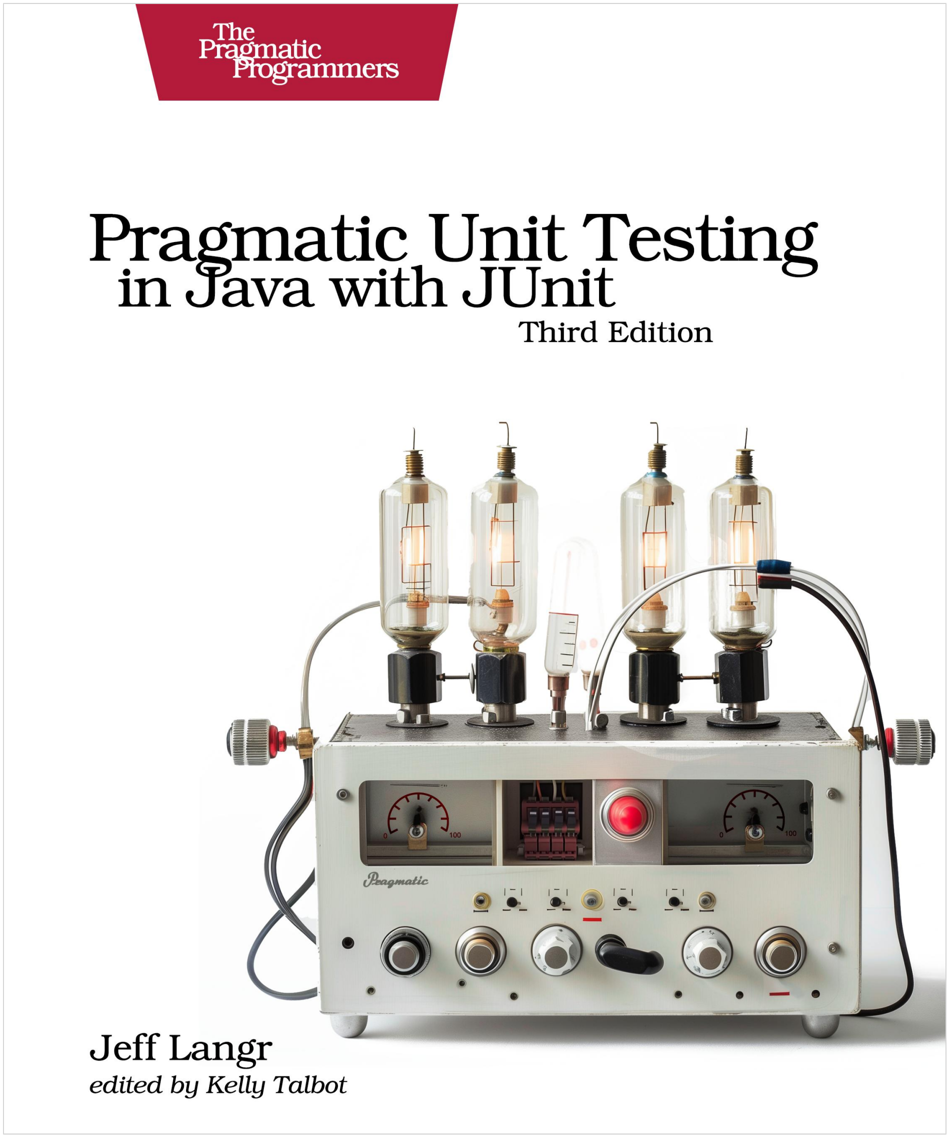

The Pragmatic Programmers

Pragmatic Unit Testing in Java with JUnit

Third Edition

Jeff Langr

edited by Kelly Talbot

Preface

Andy Hunt and Dave Thomas wrote and published the first edition of this book in 2003, about five years after JUnit was introduced. Yet if you told a 2003 Java developer to write tests to verify their own code, they'd likely have said, "Not a chance. I'm a programmer, not a tester."

Many developers since have found value in unit testing. Rather than spend copious time manually testing their solutions, and re-manually-testing them with each change, these programmers write small tests (in Java) that verify small behaviors in the system. They run these tests through a tool (usually JUnit), and immediately discover if their logic is faulty.

A decade later, unit testing had become an expected developer skill and a common interview topic. On request of Andy and Dave, I overhauled *Pragmatic Unit Testing in Java with JUnit* with additional topics and up-to-date code. I delivered the second edition in 2015.

By 2015, unit testing had become a mandate from many managers and team leads who sought increased quality in their systems. Unfortunately, many of these mandates came with well-meaning but misguided metric goals, particularly around code coverage. Today, many teams are expected to meet code coverage goals (see Improving Unit Testing Skills Using Code Coverage, on page ?) of 80%, for example.

However, shops mandating coverage goals usually see poor return on their unit testing investment. Many developers game the metric, because they don't know how to fully leverage unit testing. They write just enough "tests" to meet the metric target. These tests provide very little value: They only minimally gate defects, they don't help document system behaviors, and they afford little safety for improving the design via refactoring.

For too many developers, the taste for unit testing has soured as a result.

Another decade on, unit testing has survived and even thrived in some shops, embraced by people who've learned to practice it properly. Its most dedicated practitioners continually write tests to guide the development of code in a practice known as test-driven development (TDD). They achieve near-comprehensive code coverage in the form of fast unit tests. These tests gate most logic defects, document *all* unit behaviors, and provide the utmost confidence to continually clean the code.

No other testing form can help you effectively manage and document the thousands of unit behaviors embodied in your system. Well-written unit tests provide feedback within seconds after each tiny change you make.

This third edition of *Pragmatic Unit Testing* represents the staying power of a technique that remains relevant decades after its first formal use. This book has been overhauled—simplified, rewritten and reorganized to make learning more accessible. Virtually all code examples have been reworked to use Java 21. Many code examples have been reworked to simplify or expand on key concepts, and numerous examples have been added. It extends coverage on JUnit itself, and includes a new discussion on testing code generated by an LLM.

Unit testing remains relevant in the age of AI. Until AI tooling can verify the code it writes (it cannot in 2024), you must vet that code with unit tests. Amusingly, AI can help you write those unit tests. Learn how in a new final chapter, Chapter 13, Keeping AI Honest With Unit Tests, on page ?.

Why Unit Testing

You are *unit testing* when you (a programmer) write test code to verify *units* of code. A unit is a small bit of code that exhibits some useful behavior in your system. A unit doesn't usually represent complete end-to-end behavior; it usually supports some small facet of functionality.

The unit tests you write will manipulate your units—small bits of code—directly. As such, you'll code them in Java. You'll run these tests through JUnit, a tool that marks your tests as passing or failing.

Here are a few *whens* and *whys* for writing / running unit tests:

- You finished coding a feature and want to ensure it works as expected.
- You want to document a change so that you and others later understand the choices you coded into the system.
- You changed code and want to know if you broke anything.
- You want to understand the current behavior of the system.

Most important, good unit tests increase your confidence to ship your production system. You still need *integration* and/or *acceptance* tests, which verify end-to-end behavior. This book focuses only on unit tests.

After reading this book, you'll be able to produce lots of unit tests in no time. Take care: It's easy to create lots of costly-to-maintain tests that provide little value. Heed the advice in this book to ensure your investment in unit testing continues to pay off.

Who This Book Is For

This book is an information-loaded introductory book for programmers (comfortable with Java programming) new to unit testing. You'll learn just about everything you need to dive into testing your production systems.

What You Need

To follow along and code the examples shown in this book, you'll need the following three pieces of software:

- Java. The examples in this book use Java 21. You can find numerous sources for downloading Oracle and alternative implementations.
- An IDE. The examples in this book were built using IntelliJ IDEA (which is available as both a free "community" edition as well as a licensed product), but you can use Eclipse, NetBeans, VSCode, vim, Emacs, or pretty much any editor.
- JUnit. JUnit is usually a dependency that your build tool (Gradle or Maven, mostly) will retrieve.

JUnit has been the *de facto* Java unit testing standard for 25 years, though other tools exist. This book's examples use JUnit 5 (also known as JUnit Jupiter), which supports Java version 8 or higher.

JUnit is shipped with most major IDEs, including IntelliJ IDEA, Eclipse, and NetBeans. For VSCode users, JUnit support is included with its Extension Pack for Java plugin.

If your team uses another unit-testing tool, the vast majority of this book still applies to your world: Most of this book focuses on unit testing concepts and best practices, not a specific tool. Unit testing tools are fairly simple tools. As such, you'll still be able to readily understand the tests within this book, and also to adapt them to tests that use your tool.

Refer to the individual product sites for details on how to download, install, and configure the development tools.

How This Book Is Organized

This book is divided into four main parts:

- Unit-Testing Foundations. Learn JUnit basics by writing tests for a small example, then for a number of common code situations. Learn about testing code with challenging dependencies using test doubles. Dig into code coverage, testing multithreaded code, and integration testing topics.
- Mastering JUnit with "E"s. *E*xamine outcomes by using JUnit's many assertion forms, *E*stablish organization in your tests using lifecycle methods, nested classes, and parameterized tests, and *E*xecute appropriate subsets of tests with tags and other mechanisms.
- Increasing ROI: Unit Testing and Design. Focus on the relevance of design to unit testing and vice versa. Learn to refactor in the small, in the large, and in your tests.
- Bigger Topics Around Unit Testing. Advance your unit testing practice by learning the discipline of TDD. Discover useful considerations for adopting unit testing within your project. Wrap up your journey by understanding how unit testing remains relevant in the era of AI.

Your best path to success: Code the examples as you read!

Code and Online Resources

You'll find gobs of Java code throughout the book, most of which is included in the source distribution, downloadable from the official book page.[1]

Code snippets that can be found as part of the distribution appear with the path and filename immediately above the chunk of code. For example:

utj3-units/02/src/test/java/units/SomeStringUtils.java
```
@Test
void uppercasesSingleLetter() {
   assertEquals("A", capitalize("a"));
}
```

That snippet of code appears in the SomeStringUtils.java file in the source distribution, in the directory utj3-units/02/src/test/java/units. If you're reading this as an ebook, click the filename header to navigate to the code.

You can also find the code at GitHub[2]. The repository name appears as the first part of a listing's file name. The above code snippet resides in the utj3-

1. https://pragprog.com/titles/utj3/pragmatic-unit-testing-in-java-with-junit-third-edition/
2. https://github.com/jlangr

units repository. The branches in GitHub are named v1, v2, v3, and so on, so that snippet would come from branch v2.

Most chapters refer to only one repository. A few chapters reference more than one repository, and some repositories are used across multiple chapters.

Each repository includes a build.gradle file as well as a working Gradle wrapper. From the root of each repository, you can execute the command ./gradlew build to compile the project and execute its tests.

Many of the code listings are annotated with arrows pointing to one or more lines, to emphasize what you should focus on—added/changed lines or otherwise interesting bits of code. If an arrow points to a method signature, it means to either focus on the signature itself or on the entire method (you'll be able to figure out which from the context).

The opposite of emphasizing is de-emphasizing. I do this using ellipses. In the following snippet, other things in the code may be interesting or relevant, but the body of someMethod most certainly is not relevant:

```
void someMethod() {
   // ...
}
```

The code snippets you'll see are automatically extracted from source code, meaning that the repository code should match what's in the book. However, your IDE's settings may create formatting and other minor differences (e.g. how import statements are represented).

To reduce a bit of clutter, most of the code listings omit package statements.

Visit the book's official Pragmatic Bookshelf page for more resources.[3]

Test-Driven Development (TDD)

I practice TDD, but this isn't a book on it. Chapter 11, Advancing with Test-Driven Development (TDD), on page ? will show you how it works and how it can help. With TDD, you code unit tests first, i.e. before each bit of production code you need to write. In this book, you'll write tests for code that already exists.

Across a couple decades of practicing TDD, I gathered many insights that have made my unit testing efforts easier and more valuable. You'll learn these insights throughout *Pragmatic Unit Testing in Java*. They'll apply whether you write tests before or after you write code.

3. https://pragprog.com/titles/utj3/pragmatic-unit-testing-in-java-with-junit-third-edition/

Coding Style

This book's code exhibits my preferred programming style and formatting for Java, tempered also with a need to keep code listings brief. Many of you *will* be offended by my elimination of "safety braces" around single-line conditional blocks. For example:

```
if (condition) doSomething();
```

- It gets rid of a line in the code listing.
- In most cases, the if body fits onto a single line, as demonstrated.
- I find it reads better as a singular concept.
- If you ever need a second statement in the conditional body, you'll likely remember to add the safety braces.
- If you forget… That's why you write unit tests.

If you're unswayed, feel free to add braces. Try without, though—you might like it.

Jeff Langr
jeff@langrsoft.com
July 2024