Functional Programming Patterns in Scala and Clojure

Write Lean Programs for the JVM

The Pragmatic Bookshelf

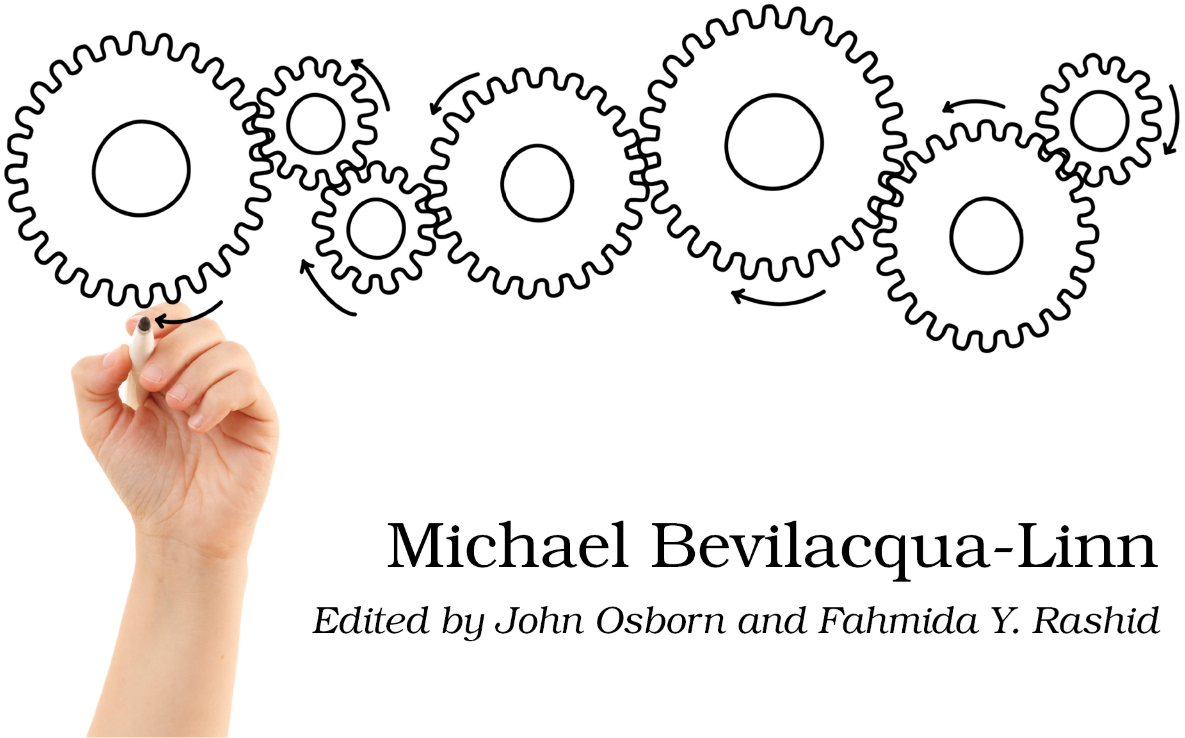
Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Functional Programming Patterns

in Scala and Clojure

Write Lean Programs for the JVM



Michael Bevilacqua-Linn

Edited by John Osborn and Fahmida Y. Rashid

Functional Programming Patterns in Scala and Clojure

Write Lean Programs for the JVM

Michael Bevilacqua-Linn

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

Acknowledgments

I'd like to thank my parents, without whom I would not exist.

Thanks also go to my wonderful girlfriend, who put up with many a night and weekend listening to me mutter about code samples, inconsistent tenses, and run-on sentences.

This book would have suffered greatly without a great group of technical reviewers. My thanks to Rod Hilton, Michajlo "Mishu" Matijkiw, Venkat Subramaniam, Justin James, Dave Cleaver, Ted Neward, Neal Ford, Richard Minerich, Dustin Campbell, Dave Copeland, Josh Carter, Fred Daoud, and Chris Smith.

Finally, I'd like to thank Dave Thomas and Andy Hunt. Their book, *The Pragmatic Programmer*, is one of the first books I read when I started my career. It made a tremendous impact, and I've still got my original dog-eared, fingerprint-covered, bruised and battered copy. In the Pragmatic Bookshelf, they've created a publisher that's truly dedicated to producing high-quality technical books and supporting the authors who write them.

Preface

This book is about patterns and functional programming in Scala and Clojure. It shows how to replace, or greatly simplify, many of the common patterns we use in object-oriented programming, and it introduces some patterns commonly used in the functional world.

Used together, these patterns let programmers solve problems faster and in a more concise, declarative style than with object-oriented programming alone. If you're using Java and want to see how functional programming can help you work more efficiently, or if you've started using Scala and Clojure and can't quite wrap your head around functional problem-solving, this is the book for you.

Before we dig in, I'd like to start off with a story. This story is true, though some names have been changed to protect the not-so-innocent.

A Tale of Functional Programming

by: Michael Bevilacqua-Linn, software firefighter

The site isn't down, but an awful lot of alarms are going off. We trace the problems to changes someone made to a third-party API we use. The changes are causing major data problems on our side; namely, we don't know what the changes are and we can't find anyone who can tell us. It also turns out the system that talks to the API uses legacy code, and the only guy who knows how to work on it happens to be away on vacation. This a big system: 500,000-lines-of-Java-and-OSGI big.

Support calls are flooding in, lots of them. Expensive support calls from frustrated customers. We need to fix the problem quickly. I start up a Clojure REPL and use it to poke around the problem API.

My boss pokes his head into my office. "How's it going?" he asks. "Working on it," I say. Ten minutes later, my grandboss pokes his head into my office. "How's it going?" he asks. "Working on it," I say. Another ten minutes pass by when my great-grandboss pokes his head into my office. "How's it going?" he asks. "Working on it," I say. I get a half hour of silence before the CTO pokes his head into my office. "Working on it," I say before he opens his mouth.

An hour passes, and I figure out what's changed. I whip up a way to keep the data clean until the legacy developer gets back and can put together a proper fix. I hand my little program off

> to the operations team, which gets it up and running in a JVM, somewhere safe. The support calls stop coming in, and everyone relaxes a bit.
>
> A week or so later at an all-hands meeting, the great-grandboss thanks me for the Java program I wrote that saved the day. I smile and say, "That wasn't Java."

The REPL, Clojure's interactive programming environment, helped a lot in this story. However, lots of languages that aren't particularly functional have similar interactive programming environments, so that's not all there is to it.

Two of the patterns that we'll see in this book, Pattern 21, *Domain-Specific Language*, on page ?, and Pattern 15, *Chain of Operations*, on page ?, contributed greatly to this story's happy ending.

Earlier on, I had written a small instance of domain-specific language for working with these particular APIs that helped me explore them very quickly even though they're very large and it was difficult to figure out where the problem might lie. In addition, the powerful data transformation facilities that functional programming relies on, such as the examples we'll see in Pattern 15, *Chain of Operations*, on page ?, helped me quickly write code to clean up the mess.

How This Book Is Organized

We'll start with an introduction to patterns and how they relate to functional programming. Then we'll take a look at an extended example, a small web framework called TinyWeb. We'll first show TinyWeb written using classic object-oriented patterns in Java. We'll then rewrite it, piece by piece, to a hybrid style that is object oriented and functional, using Scala. We'll then write in a functional style using Clojure.

The TinyWeb extended example serves a few purposes. It will let us see how several of the patterns we cover in this book fit together in a comprehensive manner. We also use it to introduce the basics of Scala and Clojure. Finally, since we'll transform TinyWeb from Java to Scala and Clojure bit by bit, it gives us a chance to explore how to easily integrate Java code with Scala and Clojure.

The remainder of the book is organized into two sections. The first, Chapter 3, *Replacing Object-Oriented Patterns*, on page ?, describes functional replacements for object-oriented patterns. These take weighty object-oriented patterns and replace them with concise functional solutions.

Peter Norvig, author of the classic Lisp text *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp [Nor92]*, current director of research at Google, and all-around very smart guy, pointed out in *Design*

Patterns in Dynamic Languages that expressive languages like Lisp could turn classic object-oriented patterns invisible.[1]

Unfortunately, not many people in the mainstream software development world seem to have read Norvig, but when we can replace a complicated pattern with something simpler, it makes sense that we should. It makes our code more concise, easier to understand, and easier to maintain.

The second section, Chapter 4, *Functional Patterns*, on page ?, describes patterns that are native to the functional world. These patterns run the gamut from tiny—patterns consisting of a line or two of code—to very large—ones that deal with entire programs.

Sometimes these patterns have first-class language support, which means that someone else has done the hard work of implementing them for us. Even when they don't, we can often use an extremely powerful pattern, Pattern 21, *Domain-Specific Language*, on page ?, to add it. This means that functional patterns are more lightweight than object-oriented patterns. You still need to understand the pattern before you can use it, but the implementation becomes as simple as a few lines of code.

Pattern Template

The patterns are laid out using the following format, with some exceptions. For example, a pattern that doesn't have any other common name would not have the Also Known As subsection, and the Functional Replacement subsections only apply to the patterns in Chapter 3, *Replacing Object-Oriented Patterns*, on page ?.

Intent

The Intent subsection provides a quick explanation of the intent of this pattern and the problem it solves.

Overview

Here is where you'll find a deeper motivation for the pattern and an explanation of how it works.

Also Known As

This subsection lists other common names for the pattern.

1. http://norvig.com/design-patterns/

Functional Replacement

Here you'll find how to replace this pattern with functional programming techniques—sometimes object-oriented patterns can be replaced with basic functional language features and sometimes with simpler patterns.

Example Code

This subsection contains samples of the pattern—for object-oriented patterns, we first show a sketch of the object-oriented solution using either class diagrams or a sketch of the Java code before showing how to replace them in Clojure and Scala. Functional patterns will be shown in Clojure and Scala only.

Discussion

This area provides a summary and discussion of interesting points about the pattern.

For Further Reading

Look here for a list of references for further information on the pattern.

Related Patterns

This provides a list of other patterns in this book that are related to the current one.

Why Scala and Clojure

Many of the patterns in this book can be applied using other languages with functional features, but we will focus on Clojure and Scala for our examples. We focus on these two languages for quite a few reasons, but first and foremost because they're both practical languages suitable for coding in production environments.

Both Scala and Clojure run on a Java virtual machine (JVM), so they interoperate well with existing Java libraries and have no issues being dropped into the JVM infrastructure. This makes them ideal to run alongside existing Java codebases. Finally, while both Scala and Clojure have functional features, they're quite different from each other. Learning to use both of them exposes us to a very broad range of functional programming paradigms.

Scala is a hybrid object-oriented/functional language. It's statically typed and combines a very sophisticated type system with local type inference, which allows us to often omit explicit type annotations in our code.

Clojure is a modern take on Lisp. It has Lisp's powerful macro system and dynamic typing, but Clojure has added some new features not seen in older Lisps. Most important is its unique way of dealing with state change by using reference types, a software transactional memory system, and efficient immutable data structures.

While Clojure is not an object-oriented language, it does give us some good features that are common in object-oriented languages, just not in the way we may be familiar with. For instance, we can still get polymorphism through Clojure's multimethods and protocols, and we can get hierarchies through Clojure's ad hoc hierarchies.

As we introduce the patterns, we'll explore both of these languages and their features, so this book serves as a good introduction to both Scala and Clojure. For further detail on either language, my favorite books are *Programming Clojure [Hal09]* and *The Joy of Clojure [FH11]* for Clojure, and *Programming Scala: Tackle Multi-Core Complexity on the Java Virtual Machine [Sub09]* and *Scala In Depth [Sue12]* for Scala.

How to Read This Book

The best place to start is with Chapter 1, *Patterns and Functional Programming,* on page ?, which goes over the basics of functional programming and its relation to patterns. Next, Chapter 2, *TinyWeb: Patterns Working Together,* on page ?, introduces basic concepts in Scala and Clojure and shows how several of the patterns in this book fit together.

From there you can jump around, pattern by pattern, as needed. The patterns covered earlier in Chapter 3, *Replacing Object-Oriented Patterns,* on page ?, and Chapter 4, *Functional Patterns,* on page ?, tend to be more basic than later ones, so they're worth reading first if you have no previous functional experience.

A quick summary of each pattern can be found in Section 1.2, *Pattern Glossary,* on page ?, for easy browsing. Once you're through the introduction, you can use it to look up a pattern that solves the particular problem you need to solve.

However, if you are completely new to functional programming, you should start with Pattern 1, *Replacing Functional Interface,* on page ?, Pattern 2, *Replacing State-Carrying Functional Interface,* on page ?, and Pattern 12, *Tail Recursion,* on page ?.

Online Resources

As you work through the book, you can download all the included code files from http://pragprog.com/titles/mbfpp/source_code. On the book's home page at http://pragprog.com/book/mbfpp, you can find links to the book forum and to report errata. Also, for ebook buyers, clicking on the box above the code extracts downloads the code for that extract for you.