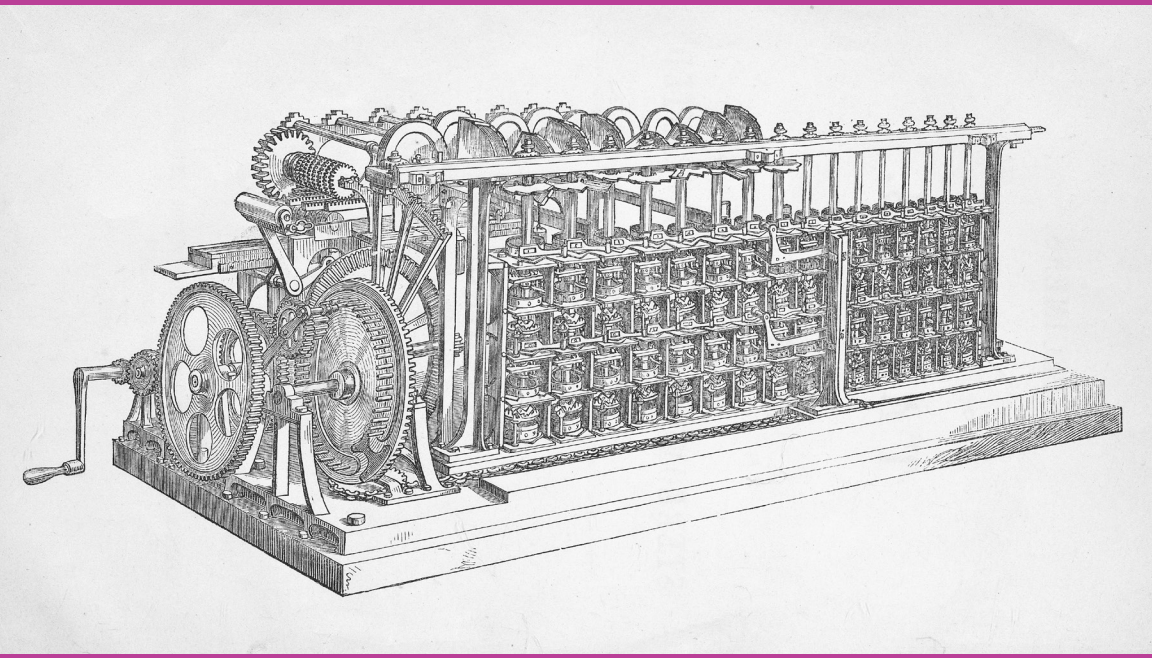
O'REILLY®

Object-Oriented vs. Functional Programming

Bridging the Divide Between Opposing Paradigms



Richard Warburton

Object-Oriented vs. Functional Programming

Bridging the Divide Between Opposing Paradigms

Richard Warburton

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Object-Oriented vs. Functional Programming
by Richard Warburton

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com* .

Editor: Brian Foster
Production Editor: Nicholas Adams
Copyeditor: Amanda Kersey
Proofreader: Nicholas Adams

Interior Designer: David Futato
Cover Designer: Randy Comer
Illustrator: Rebecca Demarest

November 2015: First Edition

Revision History for the First Edition
2015-10-30: First Release

978-1-491-93342-8

[LSI]

Table of Contents

Introduction

One of my favorite professional activities is speaking at software conferences. It's great fun because you get to meet developers who are passionate about their craft, and it gives you as a speaker the opportunity to share knowledge with them.

A talk that I've enjoyed giving recently is called "Twins: FP and OOP." I've given it at a number of conferences and user group sessions, and I've even had the pleasure of giving it as O'Reilly webcast. Developers enjoy the talk both because it has a large number of references to the film "Twins" and because it discusses one of the age-old relationships between functional and object-oriented programming.

There's only so much you can say in a conference talk though, so I was really excited when Brian Foster from O'Reilly contacted me to ask if I wanted to expand upon the topic in a report. You can also think of this as a short followup to my earlier O'Reilly published book *Java 8 Lambdas (O'Reilly)*.

You can watch the talk delivered at a conference online or delivered as an O'Reilly webcast.

What Object-Oriented and Functional Programmers Can Learn From Each Other

Before we get into the technical nitty-gritty of lambdas and design patterns, let's take a look at the technical communities. This will explain why comparing the relationship between functional and object-oriented is so important and relevant.

If you've ever read Hacker News, a programming subreddit, or any other online forum, you might have noticed there's often a touch of friction between functional programmers and developers practicing the object-oriented style. They often sound like they're talking in a different language to each other, sometimes even going so far as to throw the odd snarky insult around.

On the one hand, functional programmers can often look down on their OO counterparts. Functional programs can be very terse and elegant, packing a lot of behavior into very few lines of code. Functional programmers will make the case that in a multicore world, you need to avoid mutable state in order to scale out your programs, that programming is basically just math, and that now is the time for everyone to think in terms of functions.

Object-oriented programmers will retort that in actual business environments, very few programmers use functional languages. Object-oriented programming scales out well in terms of developers, and as an industry, we know how to do it. While programming can be viewed as a discipline of applied math, software engineering requires us to match technical solutions to business problems. The domain modelling and focus on representing real-world objects that OOP encourages in developers helps narrow that gap.

Of course, these stereotypes are overplaying the difference. Both groups of programmers are employed to solve similar business problems. Both groups are working in the same industry. Are they really so different?

I don't think so, and I think there's a lot that we can learn from each other.

What's in This Report

This report makes the case that a lot of the constructs of good object-oriented design also exist in functional programming. In order to make sure that we're all on the same page, Chapter 1 explains a little bit about functional programming and the basics of lambda expressions in Java 8.

In Chapter 2, we take a look at the SOLID principles, identified by Robert Martin, and see how they map to functional languages and paradigms. This demonstrates the similarity in terms of higher-level concepts.

In Chapter 3, we look at some behavioral design patterns. Design patterns are commonly used as a vocabulary of shared knowledge amongst object-oriented programmers. They're also often criticized by functional programmers. Here we'll look at how some of the most common object-oriented design patterns exist in the functional world.

Most of the examples in this guide are written in the Java programming language. That's not to say that Java is the only language that could have been used or that it's even a good one! It is perfectly adequate for this task though and understood by many people. This guide is also motivated by the release of Java 8 and its introduction of lambda expressions to the language. Having said all that, a lot of principles and concepts apply to many other programming languages as well, and I hope that whatever your programming language is, you take something away.

CHAPTER 1

Lambdas: Parameterizing Code by Behavior

Why Do I Need to Learn About Lambda Expressions?

Over the next two chapters, we're going to be talking in depth about the relationship between functional and object-oriented programming principles, but first let's cover some of the basics. We're going to talk about a couple of the key language features that are related to functional programming: lambda expressions and method references.

NOTE If you already have a background in functional programming, then you might want to skip this chapter and move along to the next one.

We're also going to talk about the change in thinking that they enable which is key to functional thinking: parameterizing code by behavior. It's this thinking in terms of functions and parameterizing by behavior rather than state which is key to differentiating functional programming from object-oriented programming. Theoretically this is something that we could have done in Java before with anonymous classes, but it was rarely done because they were so bulky and verbose.

We shall also be looking at the syntax of lambda expressions in the Java programming language. As I mentioned in the Introduction, a lot of these ideas go beyond Java; we are just using Java as a lingua-franca: a common language that many developers know well.

The Basics of Lambda Expressions

We will define a lambda expression as a concise way of describing an anonymous function. I appreciate that's quite a lot to take in at once, so we're going to explain what lambda expressions are by working through an example of some existing Java code. *Swing* is a platform-agnostic Java library for writing graphical user interfaces (GUIs). It has a fairly common idiom in which, in order to find out what your user did, you register an *event listener*. The event listener can then perform some action in response to the user input (see Example 1-1).

Example 1-1. Using an anonymous inner class to associate behavior with a button click

```
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent event) {
        System.out.println("button clicked");
    }
});
```

In this example, we're creating a new object that provides an implementation of the `ActionListener` class. This interface has a single method, `actionPerformed`, which is called by the `button` instance when a user actually clicks the on-screen button. The anonymous inner class provides the implementation of this method. In Example 1-1, all it does is print out a message to say that the button has been clicked.

NOTE

This is actually an example of *behavior parameterization*—we're giving the button an object that represents an action.

Anonymous inner classes were designed to make it easier for Java programmers to represent and pass around behaviors. Unfortunately, they don't make it easy enough. There are still four lines of

boilerplate code required in order to call the single line of important logic.

Boilerplate isn't the only issue, though: this code is fairly hard to read because it obscures the programmer's intent. We don't want to pass in an object; what we really want to do is pass in some behavior. In Java 8, we would write this code example as a lambda expression, as shown in Example 1-2.

Example 1-2. Using a lambda expression to associate behavior with a button click

```
button.addActionListener(event -> System.out.println("button clicked"));
```

Instead of passing in an object that implements an interface, we're passing in a block of code—a function without a name. `event` is the name of a parameter, the same parameter as in the anonymous inner class example. `->` separates the parameter from the body of the lambda expression, which is just some code that is run when a user clicks our button.

Another difference between this example and the anonymous inner class is how we declare the variable `event`. Previously, we needed to explicitly provide its type—`ActionEvent event`. In this example, we haven't provided the type at all, yet this example still compiles. What is happening under the hood is that `javac` is inferring the type of the variable `event` from its context—here, from the signature of `addActionListener`. What this means is that you don't need to explicitly write out the type when it's obvious. We'll cover this inference in more detail soon, but first let's take a look at the different ways we can write lambda expressions.

NOTE

Although lambda method parameters require less boilerplate code than was needed previously, they are still statically typed. For the sake of readability and familiarity, you have the option to include the type declarations, and sometimes the compiler just can't work it out!

Method References

A common idiom you may have noticed is the creation of a lambda expression that calls a method on its parameter. If we want a lambda

expression that gets the name of an artist, we would write the following:

```
artist -> artist.getName()
```

This is such a common idiom that there's actually an abbreviated syntax for this that lets you reuse an existing method, called a *method reference*. If we were to write the previous lambda expression using a method reference, it would look like this:

```
Artist::getName
```

The standard form is `Classname::methodName`. Remember that even though it's a method, you don't need to use brackets because you're not actually calling the method. You're providing the equivalent of a lambda expression that can be called in order to call the method. You can use method references in the same places as lambda expressions.

You can also call constructors using the same abbreviated syntax. If you were to use a lambda expression to create an `Artist`, you might write:

```
(name, nationality) -> new Artist(name, nationality)
```

We can also write this using method references:

```
Artist::new
```

This code is not only shorter but also a lot easier to read. `Artist::new` immediately tells you that you're creating a new `Artist` without your having to scan the whole line of code. Another thing to notice here is that method references automatically support multiple parameters, as long as you have the right functional interface.

It's also possible to create arrays using this method. Here is how you would create a `String` array:

```
String[]::new
```

When we were first exploring the Java 8 changes, a friend of mine said that method references "feel like cheating." What he meant was that, having looked at how we can use lambda expressions to pass

code around as if it were data, it felt like cheating to be able to reference a method directly.

In fact, method references are really making the concept of first-class functions explicit. This is the idea that we can pass behavior around and treat it like another value. For example, we can compose functions together.

Summary

Well, at one level we've learnt a little bit of new syntax that has been introduced in Java 8, which reduces boilerplate for callbacks and event handlers. But actually there's a bigger picture to these changes. We can now reduce the boilerplate around passing blocks of behavior: we're treating functions as first-class citizens. This makes parameterizing code by behavior a lot more attractive. This is key to functional programming, so key in fact that it has an associated name: higher-order functions.

Higher-order functions are just functions, methods, that return other functions or take functions as a parameter. In the next chapter we'll see that a lot of design principles in object-oriented programming can be simplified by the adoption of functional concepts like higher-order functions. Then we'll look at how many of the behavioral design patterns are actually doing a similar job to higher-order functions.

CHAPTER 2

SOLID Principles

Lambda-Enabled SOLID Principles

The SOLID principles are a set of basic principles for designing OO programs. The name itself is an acronym, with each of the five principles named after one of the letters: Single responsibility, Open/closed, Liskov substitution, Interface segregation, and Dependency inversion. The principles act as a set of guidelines to help you implement code that is easy to maintain and extend over time.

Each of the principles corresponds to a set of potential code smells that can exist in your code, and they offer a route out of the problems caused. Many books have been written on this topic, and I'm not going to cover the principles in comprehensive detail.

In the case of all these object-oriented principles, I've tried to find a conceptually related approach from the functional-programming realm. The goal here is to both show functional and object-oriented programming are related, and also what object-oriented programmers can learn from a functional style.

The Single-Responsibility Principle

Every class or method in your program should have only a single reason to change.

An inevitable fact of software development is that requirements change over time. Whether because a new feature needs to be added, your understanding of your problem domain or customer has

changed, or you need your application to be faster, over time software must evolve.

When the requirements of your software change, the responsibilities of the classes and methods that implement these requirements also change. If you have a class that has more than one responsibility, when a responsibility changes, the resulting code changes can affect the other responsibilities that the class possesses. This possibly introduces bugs and also impedes the ability of the code base to evolve.

Let's consider a simple example program that generates a `BalanceSheet`. The program needs to tabulate the `BalanceSheet` from a list of assets and render the `BalanceSheet` to a PDF report. If the implementer chose to put both the responsibilities of tabulation and rendering into one class, then that class would have two reasons for change. You might wish to change the rendering in order to generate an alternative output, such as HTML. You might also wish to change the level of detail in the `BalanceSheet` itself. This is a good motivation to decompose this problem at the high level into two classes: one to tabulate the `BalanceSheet` and one to render it.

The single-responsibility principle is stronger than that, though. A class should not just have a single responsibility: it should also encapsulate it. In other words, if I want to change the output format, then I should have to look at only the rendering class and not at the tabulation class.

This is part of the idea of a design exhibiting strong *cohesion*. A class is cohesive if its methods and fields should be treated together because they are closely related. If you tried to divide up a cohesive class, you would result in accidentally coupling the classes that you have just created.

Now that you're familiar with the single-responsibility principle, the question arises, what does this have to do with lambda expressions? Well lambda expressions make it a lot easier to implement the single-responsibility principle at the method level. Let's take a look at some code that counts the number of prime numbers up to a certain value (Example 2-1).

Example 2-1. Counting prime numbers with multiple responsibilities in a method

```
public long countPrimes(int upTo) {
    long tally = 0;
    for (int i = 1; i < upTo; i++) {
        boolean isPrime = true;
        for (int j = 2; j < i; j++) {
            if (i % j == 0) {
                isPrime = false;
            }
        }
        if (isPrime) {
            tally++;
        }
    }
    return tally;
}
```

It's pretty obvious that we're really doing two different responsibilities in Example 2-1: we're counting numbers with a certain property, and we're checking whether a number is a prime. As shown in Example 2-2, we can easily refactor this to split apart these two responsibilities.

Example 2-2. Counting prime numbers after refactoring out the isPrime check

```
public long countPrimes(int upTo) {
    long tally = 0;
    for (int i = 1; i < upTo; i++) {
        if (isPrime(i)) {
            tally++;
        }
    }
    return tally;
}

private boolean isPrime(int number) {
    for (int i = 2; i < number; i++) {
        if (number % i == 0) {
            return false;
        }
    }
    return true;
}
```

Unfortunately, we're still left in a situation where our code has two responsibilities. For the most part, our code here is dealing with

looping over numbers. If we follow the single-responsibility principle, then iteration should be encapsulated elsewhere. There's also a good practical reason to improve this code. If we want to count the number of primes for a very large `upTo` value, then we want to be able to perform this operation in parallel. That's right—the threading model is a responsibility of the code!

We can refactor our code to use the Java 8 streams library (see Example 2-3), which delegates the responsibility for controlling the loop to the library itself. Here we use the `range` method to count the numbers between `0` and `upTo`, `filter` them to check that they really are prime, and then `count` the result.

Example 2-3. Counting primes using the Java 8 streams API

```
public long countPrimes(int upTo) {
    return IntStream.range(1, upTo)
                    .filter(this::isPrime)
                    .count();
}

private boolean isPrime(int number) {
    return IntStream.range(2, number)
                    .allMatch(x -> (number % x) != 0);
}
```

So, we can use higher-order functions in order to help us easily implement the single-responsibility principle.

The Open/Closed Principle

> *Software entities should be open for extension,*
> *but closed for modification.*
>
> —Bertrand Meyer

The overarching goal of the open/closed principle is similar to that of the single-responsibility principle: to make your software less brittle to change. Again, the problem is that a single feature request or change to your software can ripple through the code base in a way that is likely to introduce new bugs. The open/closed principle is an effort to avoid that problem by ensuring that existing classes can be extended without their internal implementation being modified.

When you first hear about the open/closed principle, it sounds like a bit of a pipe dream. How can you extend the functionality of a class without having to change its implementation? The actual answer is that you rely on an abstraction and can plug in new functionality that fits into this abstraction. We can also use higher-order functions and immutability to achieve similar aims in a functional style.

Abstraction

Robert Martin's interpretation of the open/closed principle was that it was all about using polymorphism to easily depend upon an abstraction. Let's think through a concrete example. We're writing a software program that measures information about system performance and graphs the results of these measurements. For example, we might have a graph that plots how much time the computer spends in user space, kernel space, and performing I/O. I'll call the class that has the responsibility for displaying these metrics `MetricDataGraph`.

One way of designing the `MetricDataGraph` class would be to have each of the new metric points pushed into it from the agent that gathers the data. So, its public API would look something like Example 2-4.

Example 2-4. The MetricDataGraph public API

```
class MetricDataGraph {

    public void updateUserTime(int value);

    public void updateSystemTime(int value);

    public void updateIoTime(int value);

}
```

But this would mean that every time we wanted to add in a new set of time points to the plot, we would have to modify the `MetricDataGraph` class. We can resolve this issue by introducing an abstraction, which I'll call a `TimeSeries`, that represents a series of points in time. Now our `MetricDataGraph` API can be simplified to not depend upon the different types of metric that it needs to display, as shown in Example 2-5.

Example 2-5. Simplified MetricDataGraph API

```
class MetricDataGraph {

    public void addTimeSeries(TimeSeries values);

}
```

Each set of metric data can then implement the `TimeSeries` interface and be plugged in. For example, we might have concrete classes called `UserTimeSeries`, `SystemTimeSeries`, and `IoTimeSeries`. If we wanted to add, say, the amount of CPU time that gets stolen from a machine if it's virtualized, then we would add a new implementation of `TimeSeries` called `StealTimeSeries`. `MetricDataGraph` has been extended but hasn't been modified.

Higher-Order Functions

Higher-order functions also exhibit the same property of being open for extension, despite being closed for modification. A good example of this is the `ThreadLocal` class. The `ThreadLocal` class provides a variable that is special in the sense that each thread has a single copy for it to interact with. Its static `withInitial` method is a higher-order function that takes a lambda expression that represents a factory for producing an initial value.

This implements the open/closed principle because we can get new behavior out of `ThreadLocal` without modifying it. We pass in a different factory method to `withInitial` and get an instance of `ThreadLocal` with different behavior. For example, we can use `ThreadLocal` to produce a `DateFormatter` that is thread-safe with the code in Example 2-6.

Example 2-6. A ThreadLocal date formatter

```
// One implementation
ThreadLocal<DateFormat> localFormatter
    = ThreadLocal.withInitial(SimpleDateFormat::new);

// Usage
DateFormat formatter = localFormatter.get();
```

We can also generate completely different behavior by passing in a different lambda expression. For example, in Example 2-7 we're creating a unique identifier for each Java thread that is sequential.

Example 2-7. A ThreadLocal identifier

```
// Or...
AtomicInteger threadId = new AtomicInteger();
ThreadLocal<Integer> localId
    = ThreadLocal.withInitial(() -> threadId.getAndIncrement());

// Usage
int idForThisThread = localId.get();
```

Immutability

Another interpretation of the open/closed principle that doesn't follow in the object-oriented vein is the idea that immutable objects implement the open/closed principle. An immutable object is one that can't be modified after it is created.

The term "immutability" can have two potential interpretations: *observable immutability* or *implementation immutability*. Observable immutability means that from the perspective of any other object, a class is immutable; implementation immutability means that the object never mutates. Implementation immutability implies observable immutability, but the inverse isn't necessarily true.

A good example of a class that proclaims its immutability but actually is only observably immutable is `java.lang.String`, as it caches the hash code that it computes the first time its `hashCode` method is called. This is entirely safe from the perspective of other classes because there's no way for them to observe the difference between it being computed in the constructor every time or cached.

I mention immutable objects in the context of this report because they are a fairly familiar concept within functional programming. They naturally fit into the style of programming that I'm talking about.

Immutable objects implement the open/closed principle in the sense that because their internal state can't be modified, it's safe to add new methods to them. The new methods can't alter the internal state of the object, so they are closed for modification, but they are adding behavior, so they are open to extension. Of course, you still need to

be careful in order to avoid modifying state elsewhere in your program.

Immutable objects are also of particular interest because they are inherently thread-safe. There is no internal state to mutate, so they can be shared between different threads.

If we reflect on these different approaches, it's pretty clear that we've diverged quite a bit from the traditional open/closed principle. In fact, when Bertrand Meyer first introduced the principle, he defined it so that the class itself couldn't ever be altered after being completed. Within a modern Agile developer environment, it's pretty clear that the idea of a class being complete is fairly outmoded. Business requirements and usage of the application may dictate that a class be used for something that it wasn't intended to be used for. That's not a reason to ignore the open/closed principle though, just a good example of how these principles should be taken as guidelines and heuristics rather than followed religiously or to the extreme. We shouldn't judge the original definition too harshly, however, since it used in a different era and for software with specific and defined requirements.

A final point that I think is worth reflecting on is that in the context of Java 8, interpreting the open/closed principle as advocating an abstraction that we can plug multiple classes into or advocating higher-order functions amounts to the same approach. Because our abstraction needs to be represented by an interface upon which methods are called, this approach to the open/closed principle is really just a usage of polymorphism.

In Java 8, any lambda expression that gets passed into a higher-order function is represented by a functional interface. The higher-order function calls its single method, which leads to different behavior depending upon which lambda expression gets passed in. Again, under the hood, we're using polymorphism in order to implement the open/closed principle.

The Liskov Substitution Principle

Let q(x) be a property provable about objects x of type T. Then q(y) should be true for objects y of type S where S is a subtype of T.

The Liskov substitution principle is often stated in these very formal terms, but is actually a very simple concept. Informally we can think

of this as meaning that child classes should maintain the behavior they inherit from their parents. We can split out that property into four distinct areas:

- Preconditions cannot be strengthened in a subtype. Where the parent worked, the child should.
- Postconditions cannot be weakened in a subtype. Where the parent caused an effect, then the child should.
- Invariants of the supertype must be preserved in a subtype. Where parent always stuck left or maintained something, then the child should as well.
- Rule from history: don't allow state changes that your parent didn't. For example, a mutable point can't subclass an immutable point.

Functional programming tends to take a different perspective to LSP. In functional programming inheritance of behavior isn't a key trait. If you avoid inheritance hierachies then you avoid the problems that are associated with them, which is the antipattern that the Liskov substitution principle is designed to solve. This is actually becoming increasing accepted within the object-oriented community as well through the composite reuse principle: compose, don't inherit.

The Interface-Segregation Principle

The dependency of one class to another one should depend on the smallest possible interface

In order to properly understand the interface-segregation principle, let's consider a worked example in which we have people who work in a factory during the day and go home in the evening. We might define our worker interface as follows:

Example 2-8. Parsing the headings out of a file

```
interface Worker {
    public void goHome();
    public void work();
}
```

Initially our `AssemblyLine` requires two types of `Worker`: an `AssemblyWorker` and a `Manager`. Both of these go home in the eve-

ning but have different implementations of their work method depending upon what they do.

As time passes, however, and the factory modernizes, they start to introduce robots. Our robots also do work in the factory, but they don't go home at the end of the day. We can see now that our worker interface isn't meeting the ISP, since the `goHome()` method isn't really part of the minimal interface.

Now the interesting point about this example is that it all relates to subtyping. Most statically typed object-oriented languages, such as Java and C++, have what's known as nominal subtyping. This means that for a class called `Foo` to extend a class called `Bar`, you need to see `Foo extends Bar` in your code. The relationship is explicit and based upon the name of the class. This applies equally to interfaces as well as classes. In our worked example, we have code like Example 2-9 in order to let our compiler know what the relationship is between classes.

Example 2-9. Parsing the headings out of a file

```
class AssemblyWorker implements Worker
class Manager implements Worker
class Robot implements Worker
```

When the compiler comes to check whether a parameter argument is type checked, it can identify the parent type, `Worker`, and check based upon these explicit named relationships. This is shown in Example 2-10.

Example 2-10. Parsing the headings out of a file

```
public void addWorker(Worker worker) {
    workers.add(worker);
}

public static AssemblyLine newLine() {
    AssemblyLine line = new AssemblyLine();
    line.addWorker(new Manager());
    line.addWorker(new AssemblyWorker());
    line.addWorker(new Robot());
    return line;
}
```

The alternative approach is called structural subtyping, and here the relationship is implicit between types based on the shape/structure of the type. So if you call a method called `getFoo()` on a variable, then that variable just needs a `getFoo` method; it doesn't need to implement an interface or extend another class. You see this a lot in functional programming languages and also in systems like the C++ template framework. The duck typing in languages like Ruby and Python is a dynamically typed variant of structural subtyping.

If we think about this hypothetical example in a language which uses structural subtyping, then our example might be re-written like Example 2-11. The key is that the parameter `worker` has no explicit type, and the `StructuralWorker` implementation doesn't need to say explicitly that it implements or extends anything.

Example 2-11. Parsing the headings out of a file

```
class StructuralWorker {
    def work(step:ProductionStep) {
        println("I'm working on: " + step.getName)
    }
}

def addWorker(worker) {
  workers += worker
}

def newLine() = {
    val line = new AssemblyLine
    line.addWorker(new Manager())
    line.addWorker(new StructuralWorker())
    line.addWorker(new Robot())
    line
}
```

Structural subtyping removes the need for the interface-segregation principle, since it removes the explicit nature of these interfaces. A minimal interface is automatically inferred by the compiler from the use of these parameters.

The Dependency-Inversion Principle

Abstractions should not depend on details; details should depend on abstractions.

On of the ways in which we can make rigid and fragile programs that are resistant to change is by coupling high-level business logic and low-level code designed to glue modules together. This is because these are two different concerns that may change over time.

The goal of the dependency-inversion principle is to allow programmers to write high-level business logic that is independent of low-level glue code. This allows us to reuse the high-level code in a way that is abstract of the details upon which it depends. This modularity and reuse goes both ways: we can substitute in different details in order to reuse the high-level code, and we can reuse the implementation details by layering alternative business logic on top.

Let's look at a concrete example of how the dependency-inversion principle is traditionally used by thinking through the high-level decomposition involved in implementing an application that builds up an address book automatically. Our application takes in a sequence of electronic business cards as input and accumulates our address book in some storage mechanism.

It's fairly obvious that we can separate this code into three basic modules:

- The business card reader that understands an electronic business card format
- The address book storage that stores data into a text file
- The accumulation module that takes useful information from the business cards and puts it into the address book

We can visualize the relationship between these modules as shown in Figure 2-1.

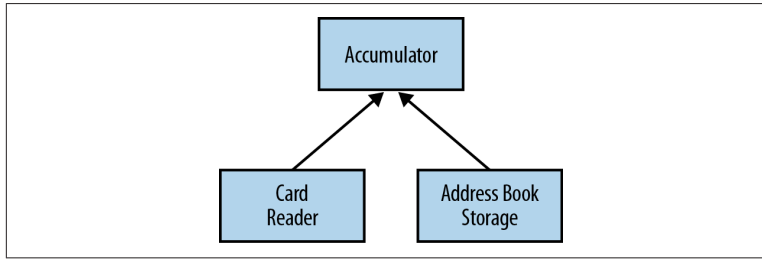


Figure 2-1. Dependencies

In this system, while reuse of the accumulation model is more complex, the business card reader and the address book storage do not

depend on any other components. We can therefore easily reuse them in another system. We can also change them; for example, we might want to use a different reader, such as reading from people's Twitter profiles; or we might want to store our address book in something other than a text file, such as a database.

In order to give ourselves the flexibility to change these components within our system, we need to ensure that the implementation of our accumulation module doesn't depend upon the specific details of either the business card reader or the address book storage. So, we introduce an abstraction for reading information and an abstraction for writing information. The implementation of our accumulation module depends upon these abstractions. We can pass in the specific details of these implementations at runtime. This is the dependency-inversion principle at work.

In the context of lambda expressions, many of the higher-order functions that we've encountered enable a dependency inversion. A function such as `map` allows us to reuse code for the general concept of transforming a stream of values between different specific transformations. The `map` function doesn't depend upon the details of any of these specific transformations, but upon an abstraction. In this case, the abstraction is the functional interface `Function`.

A more complex example of dependency inversion is resource management. Obviously, there are lots of resources that can be managed, such as database connections, thread pools, files, and network connections. I'll use files as an example because they are a relatively simple resource, but the principle can easily be applied to more complex resources within your application.

Let's look at some code that extracts headings from a hypothetical markup language where each heading is designated by being suffixed with a colon (`:`). Our method is going to extract the headings from a file by reading the file, looking at each of the lines in turn, filtering out the headings, and then closing the file. We shall also wrap any `Exception` related to the file I/O into a friendly domain exception called a `HeadingLookupException`. The code looks like Example 2-12.

Example 2-12. Parsing the headings out of a file

```
public List<String> findHeadings(Reader input) {
    try (BufferedReader reader = new BufferedReader(input)) {
        return reader.lines()
                     .filter(line -> line.endsWith(":"))
                     .map(line -> line.substring(0, line.length() - 1))
                     .collect(toList());
    } catch (IOException e) {
        throw new HeadingLookupException(e);
    }
}
```

Unfortunately, our heading-finding code is coupled with the resource-management and file-handling code. What we really want to do is write some code that finds the headings and delegates the details of a file to another method. We can use a `Stream<String>` as the abstraction we want to depend upon rather than a file. A `Stream` is much safer and less open to abuse. We also want to be able to a pass in a function that creates our domain exception if there's a problem with the file. This approach, shown in Example 2-13, allows us to segregate the domain-level error handling from the resource-management-level error handling.

Example 2-13. The domain logic with file handling split out

```
public List<String> findHeadings(Reader input) {
  return withLinesOf(
    input,
    lines -> lines.filter(line -> line.endsWith(":"))
                  .map(line -> line.substring(0, line.length()-1))
                  .collect(toList()),
    HeadingLookupException::new);
}
```

I expect that you're now wondering what that `withLinesOf` method looks like! It's shown in Example 2-14.

Example 2-14. The definition of withLinesOf

```
private <T> T withLinesOf(
   Reader input,
   Function<Stream<String>, T> handler,
   Function<IOException, RuntimeException> error) {

   try (BufferedReader reader = new BufferedReader(input)) {
```

```
        return handler.apply(reader.lines());
    } catch (IOException e) {
        throw error.apply(e);
     }
}
```

`withLinesOf` takes in a reader that handles the underlying file I/O. This is wrapped up in `BufferedReader`, which lets us read the file line by line. The `handler` function represents the body of whatever code we want to use with this function. It takes the `Stream` of the file's lines as its argument. We also take another handler called `error` that gets called when there's an exception in the I/O code. This constructs whatever domain exception we want. This exception then gets thrown in the event of a problem.

To summarize, higher-order functions provide an inversion of control, which is a form of dependency-inversion. We can easily use them with lambda expressions. The other observation to note with the dependency-inversion principle is that the abstraction that we depend upon doesn't have to be an `interface`. Here we've relied upon the existing `Stream` as an abstraction over raw reader and file handling. This approach also fits into the way that resource management is performed in functional languages—usually a higher-order function manages the resource and takes a callback function that is applied to an open resource, which is closed afterward. In fact, if lambda expressions had been available at the time, it's arguable that the `try-with-resources` feature of Java 7 could have been implemented with a single library function.

Summary

We've now reached the end of this section on SOLID, but I think it's worth going over a brief recap of the relationships that we've exposed. We talked about how the single-responsibility principle means that classes should only have a single reason to change. We've talked about how we can use functional-programming ideas to achieve that end, for example, by decoupling the threading model from application logic. The open/closed principle is normally interpreted as a call to use polymorphism to allow classes to be written in a more flexible way. We've talked about how immutability and higher-order functions are both functional programming techniques which exhibit this same open/closed dynamic.

The Liskov substitution principle imposes a set of constraints around subclassing that defines what it means to implement a correct subclass. In functional programming, we de-emphasize inheritance in our programming style. No inheritance, no problem! The interface-segregation principle encourages us to minimize the dependency on large interfaces that have multiple responsibilities. By moving to functional languages that encourage structural subtyping, we remove the need to declare these interfaces.

Finally we talked about how higher-order functions were really a form of dependency inversion. In all cases, the SOLID principles offer us a way of writing effective object-oriented programs, but we can also think in functional terms and see what the equivalent approach would be in that style.

CHAPTER 3

Design Patterns

Functional Design Patterns

One of the other bastions of design we're all familiar with is the idea of *design patterns*. Patterns document reusable templates that solve common problems in software architecture. If you spot a problem and you're familiar with an appropriate pattern, then you can take the pattern and apply it to your situation. In a sense, patterns codify what people consider to be a best-practice approach to a given problem.

In this section, we're instead going to look at how existing design patterns have become better, simpler, or in some cases, implementable in a different way. In all cases, the application of lambda expressions and a more functional approach are the driving factor behind the pattern changing.

The Command Pattern

A *command object* is an object that encapsulates all the information required to call another method later. The *command pattern* is a way of using this object in order to write generic code that sequences and executes methods based on runtime decisions. There are four classes that take part in the command pattern, as shown in Figure 3-1:

Receiver
: Performs the actual work

Command
: Encapsulates all the information required to call the receiver

Invoker
: Controls the sequencing and execution of one or more commands

Client
: Creates concrete command instances

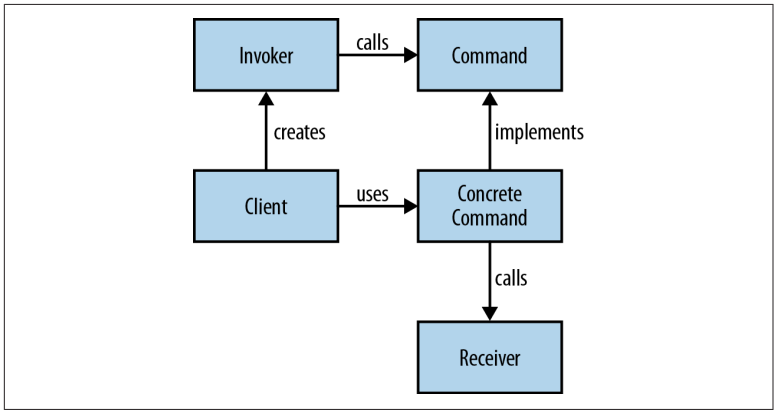


Figure 3-1. The command pattern

Let's look at a concrete example of the command pattern and see how it improves with lambda expressions. Suppose we have a GUI `Editor` component that has actions upon it that we'll be calling, such as `open` or `save`, like in Example 3-1. We want to implement macro functionality—that is, a series of operations that can be recorded and then run later as a single operation. This is our receiver.

Example 3-1. Common functions a text editor may have

```
interface Editor {

    void save();

    void open();

    void close();
```

```
}
```

In this example, each of the operations, such as `open` and `save`, are commands. We need a generic command interface to fit these different operations into. I'll call this interface `Action`, as it represents performing a single action within our domain. This is the interface that all our command objects implement (Example 3-2).

Example 3-2. All our actions implement the Action interface

```
interface Action {

    void perform();

}
```

We can now implement our `Action` interface for each of the operations. All these classes need to do is call a single method on our `Editor` and wrap this call into our `Action` interface. I'll name the classes after the operations that they wrap, with the appropriate class naming convention—so, the `save` method corresponds to a class called `Save`. Example 3-3 and Example 3-4 are our command objects.

Example 3-3. Our save action delegates to the underlying method call on Editor

```
class Save implements Action {

    private final Editor editor;

    public Save(Editor editor) {
        this.editor = editor;
    }

    @Override
    public void perform() {
        editor.save();
    }
}
```

Example 3-4. Our open action also delegates to the underlying method call on Editor

```
class Open implements Action {

    private final Editor editor;

    public Open(Editor editor) {
        this.editor = editor;
    }

    @Override
    public void perform() {
        editor.open();
    }
}
```

Now we can implement our `Macro` class. This class can `record` actions and run them as a group. We use a `List` to store the sequence of actions and then call `forEach` in order to execute each `Action` in turn. Example 3-5 is our invoker.

Example 3-5. A macro consists of a sequence of actions that can be invoked in turn

```
class Macro {

    private final List<Action> actions;

    public Macro() {
        actions = new ArrayList<>();
    }

    public void record(Action action) {
        actions.add(action);
    }

    public void run() {
        actions.forEach(Action::perform);
    }

}
```

When we come to build up a macro programmatically, we add an instance of each command that has been recorded to the `Macro` object. We can then just run the macro, and it will call each of the commands in turn. As a lazy programmer, I love the ability to define common workflows as macros. Did I say "lazy?" I meant focused on

improving my productivity. The `Macro` object is our client code and is shown in Example 3-6.

Example 3-6. Building up a macro with the command pattern

```
Macro macro = new Macro();
macro.record(new Open(editor));
macro.record(new Save(editor));
macro.record(new Close(editor));
macro.run();
```

How do lambda expressions help? Actually, all our command classes, such as `Save` and `Open`, are really just lambda expressions wanting to get out of their shells. They are blocks of behavior that we're creating classes in order to pass around. This whole pattern becomes a lot simpler with lambda expressions because we can entirely dispense with these classes. Example 3-7 shows how to use our `Macro` class without these command classes and with lambda expressions instead.

Example 3-7. Using lambda expressions to build up a macro

```
Macro macro = new Macro();
macro.record(() -> editor.open());
macro.record(() -> editor.save());
macro.record(() -> editor.close());
macro.run();
```

In fact, we can do this even better by recognizing that each of these lambda expressions is performing a single method call. So, we can actually use method references in order to wire the editor's commands to the macro object (see Example 3-8).

Example 3-8. Using method references to build up a macro

```
Macro macro = new Macro();
macro.record(editor::open);
macro.record(editor::save);
macro.record(editor::close);
macro.run();
```

The command pattern is really just a poor man's lambda expression to begin with. By using actual lambda expressions or method refer-

ences, we can clean up the code, reducing the amount of boilerplate required and making the intent of the code more obvious.

Macros are just one example of how we can use the command pattern. It's frequently used in implementing component-based GUI systems, undo functions, thread pools, transactions, and wizards.

NOTE

There is already a functional interface with the same structure as our interface `Action` in core Java—`Runnable`. We could have chosen to use that in our macro class, but in this case, it seemed more appropriate to consider an `Action` to be part of the vocabulary of our domain and create our own interface.

Strategy Pattern

The strategy pattern is a way of changing the algorithmic behavior of software based upon a runtime decision. How you implement the strategy pattern depends upon your circumstances, but in all cases, the main idea is to be able to define a common problem that is solved by different algorithms and then encapsulate all the algorithms behind the same programming interface.

An example algorithm we might want to encapsulate is compressing files. We'll give our users the choice of compressing our files using either the `zip` algorithm or the `gzip` algorithm and implement a generic `Compressor` class that can compress using either algorithm.

First we need to define the API for our strategy (see Figure 3-2), which I'll call `CompressionStrategy`. Each of our compression algorithms will implement this interface. They have the `compress` method, which takes and returns an `OutputStream`. The returned `OutputStream` is a compressed version of the input (see Example 3-9).

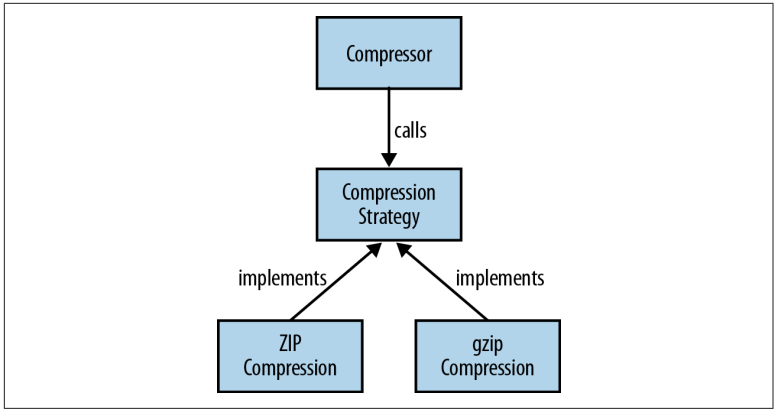


Figure 3-2. The Strategy Pattern

Example 3-9. Defining a strategy interface for compressing data

```
interface CompressionStrategy {

    OutputStream compress(OutputStream data) throws IOException;

}
```

We have two concrete implementations of this interface, one for `gzip` and one for `zip`, which use the built-in Java classes to write `gzip` (Example 3-10) and `zip` (Example 3-11) files.

Example 3-10. Using the gzip algorithm to compress data

```
class GzipCompressionStrategy implements CompressionStrategy {

    @Override
    public OutputStream compress(OutputStream data) throws IOException {
        return new GZIPOutputStream(data);
    }

}
```

Example 3-11. Using the zip algorithm to compress data

```
class ZipCompressionStrategy implements CompressionStrategy {

    @Override
    public OutputStream compress(OutputStream data) throws IOException {
        return new ZipOutputStream(data);
```

```
    }
}
```

Now we can implement our `Compressor` class, which is the *context* in which we use our strategy. This has a `compress` method on it that takes input and output files and writes a compressed version of the input file to the output file. It takes the `CompressionStrategy` as a constructor parameter that its calling code can use to make a run-time choice as to which compression strategy to use—for example, getting user input that would make the decision (see Example 3-12).

Example 3-12. Our compressor is provided with a compression strategy at construction time

```
class Compressor {

    private final CompressionStrategy strategy;

    public Compressor(CompressionStrategy strategy) {
        this.strategy = strategy;
    }

    public void compress(Path inFile, File outFile) throws IOException {
        try (OutputStream outStream = new FileOutputStream(outFile)) {
            Files.copy(inFile, strategy.compress(outStream));
        }
    }
}
```

If we have a traditional implementation of the strategy pattern, then we can write client code that creates a new `Compressor` with whichever strategy we want (Example 3-13).

Example 3-13. Instantiating the Compressor using concrete strategy classes

```
Compressor gzipCompressor =
  new Compressor(new GzipCompressionStrategy());
gzipCompressor.compress(inFile, outFile);

Compressor zipCompressor = new Compressor(new ZipCompressionStrategy());
zipCompressor.compress(inFile, outFile);
```

As with the command pattern discussed earlier, using either lambda expressions or method references allows us to remove a whole layer

of boilerplate code from this pattern. In this case, we can remove each of the concrete strategy implementations and refer to a method that implements the algorithm. Here the algorithms are represented by the constructors of the relevant `OutputStream` implementation. We can totally dispense with the `GzipCompressionStrategy` and `ZipCompressionStrategy` classes when taking this approach. Example 3-14 is what the code would look like if we used method references.

Example 3-14. Instantiating the Compressor using method references

```
Compressor gzipCompressor = new Compressor(GZIPOutputStream::new);
gzipCompressor.compress(inFile, outFile);

Compressor zipCompressor = new Compressor(ZipOutputStream::new);
zipCompressor.compress(inFile, outFile);
```

Yet again thinking in a more functional way—modelling in terms of functions rather than classes and objects—has allowed us to reduce the boilerplate and simplify an existing design pattern. This is the great win about being able to combine the functional and object-oriented world view: you get to pick the right approach for the right situation.

Summary

In this section, we have evaluated a series of design patterns and talked about how they could all be used differently with lambda expressions. In some respect, a lot of these patterns are really object-oriented embodiments of functional ideas. Take the command pattern. It's called a pattern and has some different interacting components, but the essence of the pattern is the passing around and invoking of behavior. The command pattern is all about first-class functions.

The same thing with the Strategy pattern. Its really all about putting together some behavior and passing it around; again it's a design pattern that's mimicking first-class functions. Programming languages that have a first-class representation of functions often don't talk about the strategy or command patterns, but this is what developers are doing. This is an important theme in this report. Often times, both functional and object-oriented programming languages

end up with similar patterns of code, but with different names associated with them.

CHAPTER 4
Conclusions

Object-Oriented vs. Functional Languages

In this report, we've covered a lot of ways in which ideas from functional programming relate to existing object-oriented design principles. These idioms aren't as different as a lot of people make them out to be. Definitely functional programming emphasizes the power of reuse and composition of behavior through higher-order functions. And there's no doubt that immutable data structures improve the safety of our code. But these features can be supported in an object-oriented context as well, the common theme being the benefits that be achieved are universal to both approaches. We're always seeking to write code that is safer and offers more opportunity for composing together behavior in a flexible manner.

Functional programming is about a thought process. It's not necessarily the case that you need a new language in order to program in a functional style. Some language features often help though. The introduction of lambda expressions in Java 8 makes it a language more suited to functional programming. While object-oriented programming traditionally has been about encapsulating data and behavior together, it is now adding more support for behavior on its own thanks to ideas from functional programming.

Other languages such as Scala or Haskell take functional ideas further. Scala offers a mix of both functional and object-oriented programming facilities, whilst Haskell focuses purely on functional programming. It's well worth exploring these languages and seeing what

set of language features you find useful in your problem domain. However, there's no need to necessarily move to Scala or Haskell thinking that they're the only way to program in a functional style. However, they certainly offer some features that Java lacks, and it's sometimes worth using different programming languages.

I appreciate that this maybe is a controversial opinion to those who have spent their careers advocating functional programming, but software development isn't about idealism or religious evangelism: it's about producing reliable software that works for our clients and business. A functional style of programming can help us achieve this end, but it is not the end in and of itself.

Programming Language Evolution

One of the interesting trends over time in programming languages is the gradual shift between languages that are more object-oriented and more functional. If we jump in our Delorean and go back in time to the 1980s, a lot of interesting changes were going on. Older procedural programming lanugages were being phased out and there was a growth in the popularity of both object-oriented and functional programming languages.

Interestingly enough, a lot of the early advocates of both functional and object-oriented languages combined features of the other. If you ask any object-oriented purist what her ideal programming language is, she'll tell you it's Smalltalk. Smalltalk 80 had lambda expressions, and Smalltalk's collections library was inherently functional in nature, and equivalent operations to map, reduce, and filter existed (albeit under different names).

A lot of purist functional programmers from the period would tell you that Common LISP is the ideal functional language. Interestingly enough, it had a system for object orientation called CLOS (Common LISP Object System). So back in the 1980s, there was reasonable recognition that neither paradigm was the only true way to enlightenment.

During the 1990s, programming changed. Object-oriented programming became established as a dominant programming approach for business users. Languages such as Java and C++ grew in popularity. In the late 1990s and early 2000s, Java became a

hugely popular language. In 2001, the JavaOne conference had 28,000 attendees. That's the size of a rock concert!

At the time of writing, the trend has changed again. Popular programming languages are moving away from being specifically object-oriented or functional. You will no doubt get the old holdout such as Haskell or Clojure, but by and large, languages are going hybrid. Both C++ and Java 8 have added lambda expressions and started to retrofit useful elements of functional programming to their existing object-oriented capabilities. Not only that but the underlying ideas which have been adopted in terms of generics in Java and templating in C++ originated in functional programming languages.

Newer languages are multiparadigm from the get-go. F# is a great example of a language which has a functional style but also maintains parity with C# in terms of its object-oriented features. Languages such as Ruby, Python, and Groovy can be written in both a functional and object-oriented style, all having functional features in their collections API. There are a number of newer languages on the JVM that have developed over the last decade or so, and they predominantly have a mixture of functional and object-oriented features. Scala, Ceylon, and Kotlin all come to mind in this regard.

The future is hybrid: pick the best features and ideas from both functional and object-oriented approaches in order to solve the problem at hand.

About the Author

Richard Warburton is an empirical technologist, solver of deep-dive technical problems, and works independently as a software engineer and trainer. Recently he wrote *Java 8 Lambdas* (O'Reilly) and helps developers learn via Iteratr Learning and Pluralsight. He's worked as a developer in diverse areas, including statistical analytics, static analysis, compilers, and network protocols. He is a leader in the London Java community. Richard is also a known conference speaker, having talked at Devoxx, JavaOne, QCon SF, JFokus, Devoxx UK, Geecon, Oredev, JAX London, JEEConf, and Codemotion. He obtained a PhD in computer science from The University of Warwick.