Extracted from:

Functional Programming Patterns in Scala and Clojure

Write Lean Programs for the JVM

The Pragmatic Bookshelf

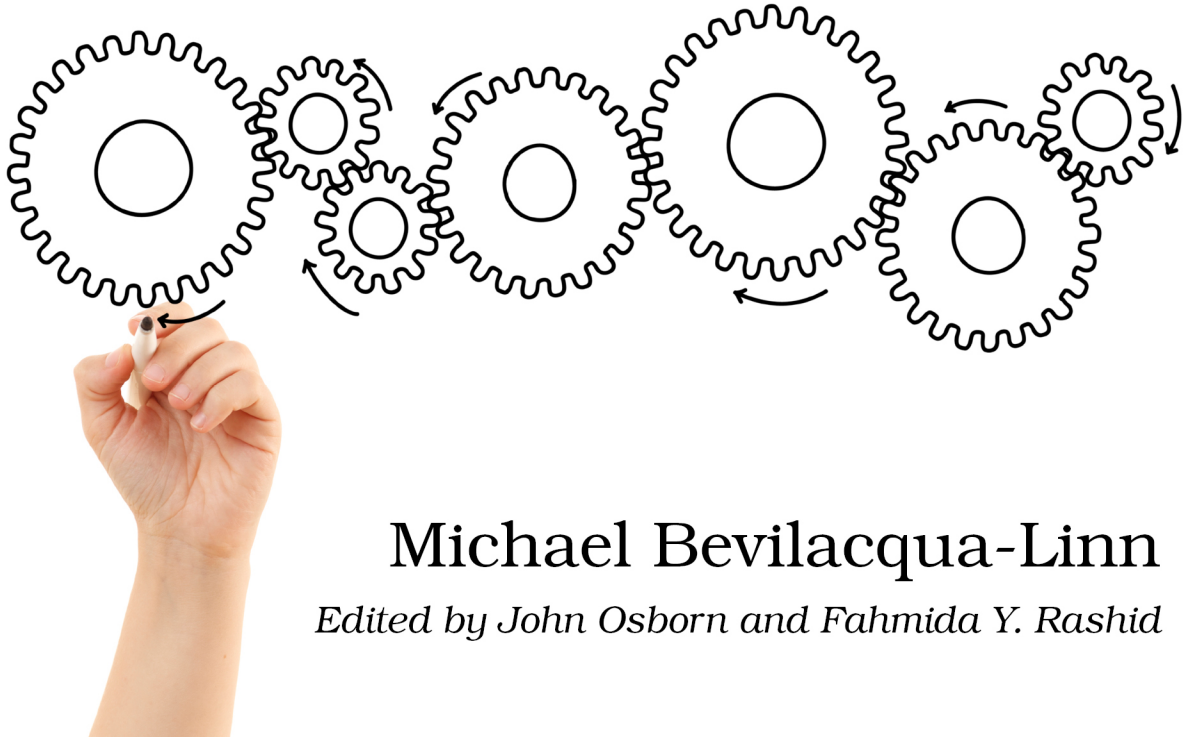
Dallas, Texas • Raleigh, North Carolina

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

Pattern 3

Replacing Command

Intent

To turn a method invocation into an object and execute it in a central location so that we can keep track of invocations so they can be undone, logged, and so forth

Overview

Command encapsulates an action along with the information needed to perform it. Though it seems simple, the pattern has quite a few moving parts. In addition to the `Command` interface and its implementations, there's a client, which is responsible for creating the `Command`; an invoker, which is responsible for running it; and a receiver, on which the `Command` performs its action.

The invoker is worth talking about a little because it's often misunderstood. It helps to decouple the invocation of a method from the client asking for it to be invoked and gives us a central location in which all method invocations take place. This, combined with the fact that the invocation is represented by an object, lets us do handy things like log the method invocation so it can be undone or perhaps serialized to disk.

Figure 3, *Command Outline*, on page 6 sketches out how Command fits together.

A simple example is a logging `Command`. Here, the client is any class that needs to do logging and the receiver is a `Logger` instance. The invoker is the class that the client calls instead of calling the `Logger` directly.

Also Known As

Action

Functional Replacement

Command has a few moving parts, as does our functional replacement. The Command class itself is a Functional Interface that generally carries state, so we'll replace it with the closures we introduced in Pattern 2, *Replacing State-Carrying Functional Interface*, on page ?.

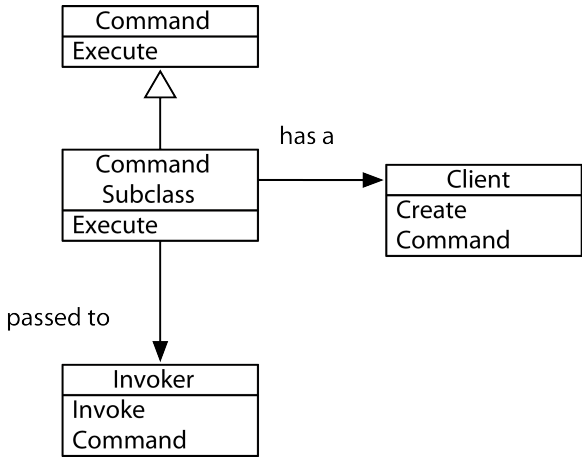


Figure 3—Command Outline. The outline of the Command pattern

Next we'll replace the invoker with a simple function responsible for executing commands, which I'll call the execution function. Just like the invoker, the execution function gives us a central place to control execution of our commands, so we can store or otherwise manipulate them as needed.

Finally, we'll create a *Function Builder* that's responsible for creating our commands so we can create them easily and consistently.

Sample Code: Cash Register

Let's look at how we'd implement a simple cash register with Command. Our cash register is very basic: it only handles whole dollars, and it contains a total amount of cash. Cash can only be added to the register.

We'll keep a log of transactions so that we can replay them. We'll take a look at how we'd do this with the traditional Command pattern first before moving on to the functional replacements in Scala and Clojure.

Classic Java

A Java implementation starts with defining a standard `Command` interface. This is an example of Pattern 1, *Replacing Functional Interface*, on page ?. We implement that interface with a `Purchase` class.

The receiver for our pattern is a `CashRegister` class. A `Purchase` will contain a reference to the `CashRegister` it should be executed against. To round out the pattern, we'll need an invoker, `PurchaseInvoker`, to actually execute our purchases.

A diagram of this implementation is below, and the full source can be found in this book's code samples.

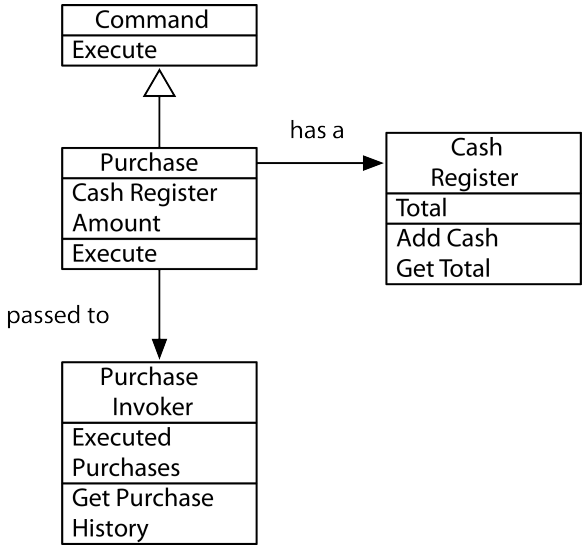


Figure 4—Cash Register Command. The structure of a cash register as a Command pattern in Java

Now that we've sketched out a Java implementation of the Command pattern, let's see how we can simplify it using functional programming.

In Scala

The cleanest replacement for Command in Scala takes advantage of Scala's hybrid nature. We'll retain a CashRegister class, just as in Java; however, instead of creating a Command interface and implementation, we'll simply use higher-order functions. Instead of creating a separate class to act as an invoker, we'll just create an execution function. Let's take a look at the code, starting with the CashRegister itself:

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/command/register/Register.scala

```
class CashRegister(var total: Int) {
  def addCash(toAdd: Int) {
    total += toAdd
  }
}
```

Next we'll create the function makePurchase to create our purchase functions. It takes amount and register as arguments to add to it, and it returns a function that does the deed, as the following code shows:

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/command/register/Register.scala
```
def makePurchase(register: CashRegister, amount: Int) = {
  () => {
    println("Purchase in amount: " + amount)
    register.addCash(amount)
  }
}
```

Finally, let's look at our execution function, executePurchase. It just adds the purchase function it was passed to a Vector to keep track of purchases we've made before executing it. Here's the code:

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/command/register/Register.scala
```
var purchases: Vector[() => Unit] = Vector()
def executePurchase(purchase: () => Unit) = {
  purchases = purchases :+ purchase
  purchase()
}
```

What's That Var Doing in Here?

The code on page 8 has a mutable reference front and center.

```
var purchases: Vector[() => Unit] = Vector()
```

This might seem a bit odd in a book on functional programming. Shouldn't everything be immutable? It turns out that it's difficult, though not impossible, to model everything in a purely functional way. Keeping track of changing stack is especially tricky.

Never fear though; all we're doing here is moving around a reference to a bit of immutable data. This gives us most of the benefits of immutability. For instance, we can safely create as many references to our original Vector without worrying about accidentally modifying the original, as the following code shows:

```
scala> var v1 = Vector("foo", "bar")
v1: scala.collection.immutable.Vector[String] = Vector(foo, bar)

scala> val v1Copy = v1
v1Copy: scala.collection.immutable.Vector[String] = Vector(foo, bar)

scala> v1 = v1 :+ "baz"
v1: scala.collection.immutable.Vector[String] = Vector(foo, bar, baz)

scala> v1Copy
res0: scala.collection.immutable.Vector[String] = Vector(foo, bar)
```

It's possible to program in a purely functional way using the excellent Scalaz library,[a] but this book focuses on a more pragmatic form of functional programming.

a. https://code.google.com/p/scalaz/

Here's our solution in action:

```
scala> val register = new CashRegister(0)
register: CashRegister = CashRegister@53f7eb48

scala> val purchaseOne = makePurchase(register, 100)
purchaseOne: () => Unit = <function0>

scala> val purchaseTwo = makePurchase(register, 50)
purchaseTwo: () => Unit = <function0>

scala> executePurchase(purchaseOne)
Purchase in amount: 100

scala> executePurchase(purchaseTwo)
Purchase in amount: 50
```

As you can see, the register now has the correct total:

```
scala> register.total
res2: Int = 150
```

If we reset the register to 0, we can replay the purchases using the ones we've stored in the purchases vector:

```
scala> register.total = 0
register.total: Int = 0

scala> for(purchase <- purchases){ purchase.apply() }
Purchase in amount: 100
Purchase in amount: 50

scala> register.total
res4: Int = 150
```

Compared to the Java version, the Scala version is quite a bit more straightforward. No need for a Command, Purchase, or separate invoker class when you've got higher-order functions.

In Clojure

The overall structure of the Clojure solution is similar to the Scala one. We'll use higher-order functions for our commands, and we'll use an execution function to execute them. The biggest difference between the Scala and Clojure solutions is the cash register itself. Since Clojure doesn't have object-oriented features, we can't create a CashRegister class.

Instead, we'll simply use a Clojure atom to keep track of the cash in the register. To do so, we'll create a make-cash-register function that returns a fresh atom to

represent a new register and an add-cash function that takes a register and an amount. We'll also create a reset function to reset our register to zero.

Here's the code for the Clojure cash register:

ClojureExamples/src/mbfpp/oo/command/cash_register.clj

```
(defn make-cash-register []
  (let [register (atom 0)]
    (set-validator! register (fn [new-total] (>= new-total 0)))
    register))

(defn add-cash [register to-add]
  (swap! register + to-add))

(defn reset [register]
  (swap! register (fn [oldval] 0)))
```

We can create an empty register:

```
=> (def register (make-cash-register))
#'mblinn.oo.command.ex1.version-one/register
```

And we'll add some cash:

```
=> (add-cash register 100)
100
```

Now that we've got our cash register, let's take a look at how we'll create commands. Remember, in Java this would require us to implement a Command interface. In Clojure we just use a function to represent purchases.

To create them, we'll use the make-purchase function, which takes a register and an amount and returns a function that adds amount to register. Here's the code:

ClojureExamples/src/mbfpp/oo/command/cash_register.clj

```
(defn make-purchase [register amount]
  (fn []
    (println (str "Purchase in amount: " amount))
    (add-cash register amount)))
```

Here we use it to create a couple of purchases:

```
=> (def register (make-cash-register))
#'mblinn.oo.command.ex1.version-one/register
=> @register
0
=> (def purchase-1 (make-purchase register 100))
#'mblinn.oo.command.ex1.version-one/purchase-1
=> (def purchase-2 (make-purchase register 50))
#'mblinn.oo.command.ex1.version-one/purchase-2
```

And here we run them:

```
=> (purchase-1)
Purchase in amount: 100
100
=> (purchase-2)
Purchase in amount: 50
150
```

To finish off the example, we'll need our execution function, execute-purchase, which stores the purchase commands before executing them. We'll use an atom, purchases, wrapped around a vector for that purpose. Here's the code we need:

ClojureExamples/src/mbfpp/oo/command/cash_register.clj

```
(def purchases (atom []))
(defn execute-purchase [purchase]
  (swap! purchases conj purchase)
  (purchase))
```

Now we can use execute-purchase to execute the purchases we defined above so that this time we'll get them in our purchase history. We'll reset register first:

```
=> (execute-purchase purchase-1)
Purchase in amount: 100
100
=> (execute-purchase purchase-2)
Purchase in amount: 50
150
```

Now if we reset the register again, we can run through our purchase history to rerun the purchases:

```
=> (reset register)
0
=> (doseq [purchase @purchases] (purchase))
Purchase in amount: 100
Purchase in amount: 50
nil
=> @register
150
```

That's our Clojure solution! One tidbit I find interesting about it is how we modeled our cash register without using objects by simply representing it as a bit of data and functions that operate on it. This is, of course, common in the functional world and it often leads to simpler code and smaller systems.

This might seem limiting to the experienced object-oriented programmer at first; for instance, what if you need polymorphism or hierarchies of types? Never fear, Clojure provides the programmer with all of the good stuff from the object-oriented world, just in a different, more decoupled form. For

instance, Clojure has a way to create ad hoc hierarchies, and its multimethods and protocols give us polymorphism. We'll look at some of these tools in more detail in Pattern 10, *Replacing Visitor*, on page ?.

Discussion

I've found that Command, though it's used everywhere, is one of the most misunderstood patterns of *Design Patterns: Elements of Reusable Object-Oriented Software [GHJV95]*. People often conflate the `Command` interface with the Command pattern. The `Command` interface is only a small part of the overall pattern and is itself an example of Pattern 1, *Replacing Functional Interface*, on page ?. This isn't to say that the way it's commonly used is wrong, but it is often different than the way the Gang of Four describes it, which can lead to some confusion when talking about the pattern.

The examples in this section implemented a replacement for the full pattern in all its invoker/receiver/client glory, but it's easy enough to strip out unneeded parts. For example, if we didn't need our command to be able to work with multiple registers, we wouldn't have to pass a register into `makePurchase`.

For Further Reading

Design Patterns: Elements of Reusable Object-Oriented Software [GHJV95]—*Command*

Related Patterns

Pattern 1, *Replacing Functional Interface*, on page ?