Functional thinking: Functional design patterns, Part 1

How patterns manifest in the functional world

Neal Ford
Software Architect / Meme Wrangler
ThoughtWorks Inc.

Skill Level: Intermediate

Date: 06 Mar 2012

Contrary to popular belief, design patterns exist in functional programming — but they sometimes differ from their object-oriented counterparts in appearance and behavior. In this installment of *Functional thinking*, Neal Ford looks at ways in which patterns manifest in the functional paradigm, illustrating how the solutions differ.

View more content in this series

About this series

This series aims to reorient your perspective toward a functional mindset, helping you look at common problems in new ways and find ways to improve your day-to-day coding. It explores functional programming concepts, frameworks that allow functional programming within the Java™ language, functional programming languages that run on the JVM, and some future-leaning directions of language design. The series is geared toward developers who know Java and how its abstractions work but have little or no experience using a functional language.

Some contingents in the functional world claim that the concept of the design pattern is flawed and isn't needed in functional programming. A case can be made for that view under a narrow definition of *pattern* — but that's an argument more about semantics than use. The *concept* of a design pattern — a named, cataloged solution to a common problem — is alive and well. However, patterns sometimes take different guises under different paradigms. Because the building blocks and approaches to problems are different in the functional world, some of the traditional Gang of Four patterns (see Resources) disappear, while others preserve the problem but solve it radically differently. This installment and the next investigate some traditional design patterns and rethink them in a functional way.

In the functional-programming world, traditional design patterns generally manifest in one of three ways:

- The pattern is absorbed by the language.
- The pattern solution still exists in the functional paradigm, but the implementation details differ.
- The solution is implemented using capabilities other languages or paradigms lack. (For example, many solutions that use metaprogramming are clean and elegant — and they're not possible in Java.)

I'll investigate these three cases in turn, starting in this installment with some familiar patterns, most of which are wholly or partially subsumed by modern languages.

Factories and currying

Currying is a feature of many functional languages. Named after mathematician Haskell Curry (for whom the Haskell programming language is also named), currying transforms a multiargument function so that it can be called as a chain of single-argument functions. Closely related is *partial application*, a technique for assigning a fixed value to one or more of the arguments to a function, thereby producing another function of smaller *arity*. (Arity is the number of parameters to the function.) I discussed both techniques in "Thinking functionally, Part 3."

In the context of design patterns, currying acts as a factory for functions. A common feature in functional programming languages is first-class (or higher-order) functions, which allow functions to act as any other data structure. Thanks to this feature, I can easily create functions that return other functions based on some criterion, which is the essence of a factory. For example, if you have a general function that adds two numbers, you can use currying as a factory to create a function that always adds one to its parameter — an incrementer, as shown in Listing 1, implemented in Groovy:

Listing 1. Currying as a function factory

```
def adder = { x, y -> return x + y }
def incrementer = adder.curry(1)

println "increment 7: ${incrementer(7)}" // prints "increment 7: 8"
```

In Listing 1, I curry the first parameter as `1`, returning a function that accepts a single parameter. In essence, I have created a function factory.

When your language supports this kind of behavior natively, it tends to be used as a building block for other things, both large and small. For example, consider the Scala example shown in Listing 2:

Listing 2. Scala's "casual" use of currying

```
object CurryTest extends Application {

  def filter(xs: List[Int], p: Int => Boolean): List[Int] =
    if (xs.isEmpty) xs
    else if (p(xs.head)) xs.head :: filter(xs.tail, p)
    else filter(xs.tail, p)

  def dividesBy(n: Int)(x: Int) = ((x % n) == 0)

  val nums = List(1, 2, 3, 4, 5, 6, 7, 8)
  println(filter(nums, dividesBy(2)))
  println(filter(nums, dividesBy(3)))
}
```

The code in Listing 2 is one of the examples of both recursion and currying from the Scala documentation (see Resources). The `filter()` method recursively filters a list of integers via the parameter `p`. `p` is a *predicate function* — a common term in the functional world for a Boolean function. The `filter()` method checks to see if the list is empty and, if it is, simply returns; otherwise, it checks the first element in the list (`xs.head`) via the predicate to see if it should be included in the filtered list. If it passes the predicate, the return is a new list with the head at the front and the filtered tail as the remainder. If the first element fails the predicate test, the return becomes solely the filtered remainder of the list.

What's interesting in Listing 2 from a patterns standpoint is the "casual" use of currying in the `dividesBy()` method. Notice that `dividesBy()` accepts two parameters and returns `true` or `false` based on whether the second parameter divides evenly into the first. However, when this method is called as part of the invocation of the `filter()` method, it is called with only one parameter — the result of which is a curried function that is then used as the predicate within the `filter()` method.

This example illustrates the first two of the ways that patterns manifest in functional programming, as I listed them at the start of this article. First, currying is built into the language or runtime, so the concept of a function factory is ingrained and doesn't require extra structure. Second, it illustrates my point about different implementations. Using currying as in Listing 2 would probably never occur to a typical Java programmer; we've never really had portable code and certainly never thought about constructing specific functions from more general ones. In fact, chances are that most imperative developers wouldn't think of using a design pattern here, because creating a specific `dividesBy()` method from a general one seems like a small problem, whereas design patterns — relying mostly on structure to solve problems and therefore requiring a lot of overhead to implement — seem like solutions to large problems. Using currying as it was intended doesn't justify the formality of a special name other than the one it already has.

First-class functions and design patterns

Having first-class functions greatly simplifies many commonly used design patterns. (The Command design pattern even disappears, because you no longer need an object wrapper for portable functionality.)

Template Method

First-class functions make the Template Method design pattern (see Resources) simpler to implement, because they remove potentially unnecessary structure. Template Method defines the skeleton of an algorithm in a method, deferring some steps to subclasses and forcing them define those steps without changing the algorithm's structure. A typical implementation of the Template Method appears in Listing 3, in Groovy:

Listing 3. "Standard" Template Method implementation

```
abstract class Customer {
  def plan

  def Customer() {
    plan = []
  }

  def abstract checkCredit()
  def abstract checkInventory()
  def abstract ship()

  def process() {
    checkCredit()
    checkInventory()
    ship()
  }
}
```

In Listing 3, the `process()` method relies on the `checkCredit()`, `checkInventory()`, and `ship()` methods, whose definitions must be supplied by subclasses because they are abstract methods.

Because first-class functions can act as any other data structure, I can redefine the example in Listing 3 using code blocks, as shown in Listing 4:

Listing 4. Template Method with first-class functions

```
class CustomerBlocks {
  def plan, checkCredit, checkInventory, ship

  def CustomerBlocks() {
    plan = []
  }

  def process() {
    checkCredit()
    checkInventory()
    ship()
  }
}

class UsCustomerBlocks extends CustomerBlocks{
  def UsCustomerBlocks() {
```

```
    checkCredit = { plan.add "checking US customer credit" }
    checkInventory = { plan.add "checking US warehouses" }
    ship = { plan.add "Shipping to US address" }
  }
}
```

In Listing 4, the steps in the algorithm are merely properties of the class, assignable like any other property. This is an example in which the language feature mostly absorbs the implementation details. It's still useful to talk about this pattern as a solution (deferring steps to subsequent handlers) to a problem, but the implementation is simpler.

The two solutions aren't equivalent. In the "traditional" Template Method example in Listing 3, the abstract class requires subclasses to implement the dependent methods. Of course, the subclass might just create an empty method body, but the abstract-method definition forms a kind of documentation, reminding subclassers to take it into account. On the other hand, the rigidity of method declarations may not be suitable in situations in which more flexibility is required. For example, I could create a version of my `Customer` class that accepts any list of methods for processing.

Deep support for features like code blocks makes languages developer-friendly. Consider the case where you want to allow subclassers to skip some of the steps. Groovy has a special protected access operator (`?.`) that ensures the object isn't null before invoking a method on it. Consider the `process()` definition in Listing 5:

Listing 5. Adding protection to code-block invocation

```
def process() {
  checkCredit?.call()
  checkInventory?.call()
  ship?.call()
}
```

In Listing 5, whoever implements the subclass can choose which of the child methods to assign code to, leaving the others safely blank.

Strategy

Another popular design pattern simplified by first-class functions is the Strategy pattern. Strategy defines a family of algorithms, encapsulating each one and making them interchangeable. It lets the algorithm vary independently from clients that use it. First-class functions make it simple to build and manipulate strategies.

A traditional implementation of the Strategy design pattern, for calculating the products of numbers, appears in Listing 6:

Listing 6. Using the Strategy design pattern for products of two numbers

```
interface Calc {
  def product(n, m)
}

class CalcMult implements Calc {
  def product(n, m) { n * m }
}

class CalcAdds implements Calc {

  def product(n, m) {
    def result = 0
    n.times {
      result += m
    }
    result
  }
}
```

In Listing 6, I define an interface for the product of two numbers. I implement the interface with two different concrete classes (strategies): one using multiplication and the other using addition. To test these strategies, I create a test case, shown in Listing 7:

Listing 7. Testing product strategies

```
class StrategyTest {
  def listOfStrategies = [new CalcMult(), new CalcAdds()]

  @Test
  public void product_verifier() {
    listOfStrategies.each { s ->
      assertEquals(10, s.product(5, 2))
    }
  }
}
```

As expected in Listing 7, both strategies return the same value. Using code blocks as first-class functions, I can reduce much of the ceremony from the previous example. Consider the case of exponentiation strategies, shown in Listing 8:

Listing 8. Testing exponentiation with less ceremony

```
@Test
public void exp_verifier() {
  def listOfExp = [
      {i, j -> Math.pow(i, j)},
      {i, j ->
        def result = i
        (j-1).times { result *= i }
        result
      }]

  listOfExp.each { e ->
    assertEquals(32, e(2, 5))
    assertEquals(100, e(10, 2))
    assertEquals(1000, e(10, 3))
  }
}
```

In Listing 8, I define two strategies for exponentiation directly inline, using Groovy code blocks. As in the Template Method example, I trade formality for convenience. The traditional approach forces name and structure around each strategy, which is sometimes desirable. However, note that I have the option to add more stringent safeguards to the code in Listing 8, whereas I can't easily bypass the restrictions imposed by the more traditional approach — which is more of a dynamic-versus-static argument than a functional-programming-versus-design-patterns one.

The patterns affected by the presence of first-class functions are mostly examples of patterns being absorbed by the language. Next, I'll show one that keeps the semantics but changes implementation.

Flyweight and memoization

The Flyweight pattern is an optimization technique that uses sharing to support a large number of fine-grained object references. You keep a pool of objects available, creating references into the pool for particular views. Flyweight uses the idea of a *canonical object* — a single representative object that represents all other objects of that type. For example, if you have a particular consumer product, a canonical version of the product represents all products of that type. In an application, instead of creating a list of products for each user, you create one list of canonical products, and each user has a reference into that list for their product.

Consider the classes in Listing 9, which model computer types:

Listing 9. Simple classes modeling types of computers

```
class Computer {
  def type
  def cpu
  def memory
  def hardDrive
  def cd
}

class Desktop extends Computer {
  def driveBays
  def fanWattage
  def videoCard
}

class Laptop extends Computer {
  def usbPorts
  def dockingBay
}

class AssignedComputer {
  def computerType
  def userId

  public AssignedComputer(computerType, userId) {
    this.computerType = computerType
    this.userId = userId
  }
}
```

In these classes, let's say that it is inefficient to create a new `Computer` instance for each user, assuming that all the computers have the same specifications. An `AssignedComputer` associates a computer with a user.

A common way to make this code more efficient combines the Factory and Flyweight patterns. Consider the singleton factory for generating canonical computer types, shown in Listing 10:

Listing 10. Singleton factory for flyweight computer instances

```
class ComputerFactory {
  def types = [:]
  static def instance;

  private ComputerFactory() {
    def laptop = new Laptop()
    def tower = new Desktop()
    types.put("MacBookPro6_2", laptop)
    types.put("SunTower",  tower)
  }

  static def getInstance() {
    if (instance == null)
      instance = new ComputerFactory()
    instance
  }

  def ofType(computer) {
    types[computer]
  }
}
```

The `ComputerFactory` class builds a cache of possible computer types, then delivers the proper instance via its `ofType()` method. This is a traditional singleton factory as you would write it in Java.

However, Singleton is a design pattern too (see Resources), and it serves as another good example of a pattern absorbed by a runtime. Consider the simplified `ComputerFactory`, using the `@Singleton` annotation provided by Groovy, shown in Listing 11:

Listing 11. Simplified singleton factory

```
@Singleton class ComputerFactory {
  def types = [:]

  private ComputerFactory() {
    def laptop = new Laptop()
    def tower = new Desktop()
    types.put("MacBookPro6_2", laptop)
    types.put("SunTower",  tower)
  }

  def ofType(computer) {
    types[computer]
  }
}
```

To test that the factory returns canonical instances, I write a unit test, shown in Listing 12:

Listing 12. Testing canonical types

```
@Test
public void flyweight_computers() {
  def bob = new AssignedComputer(ComputerFactory.instance.ofType("MacBookPro6_2"), "Bob")
  def steve = new AssignedComputer(ComputerFactory.instance.ofType("MacBookPro6_2"),
  "Steve") assertTrue(bob.computerType == steve.computerType)
}
```

Saving common information across instances is a good idea, and it's an idea that I want to preserve as I cross into functional programming. However, the implementation details are quite different. This is an example of preserving the *semantics* of a pattern while changing (preferably, simplifying) the implementation.

In the last installment, I covered *memoization*, a feature built into a programming language that enables automatic caching of recurring function-return values. In other words, a memoized function allows the runtime to cache the values for you. Recent versions of Groovy support memoization (see Resources). Consider the functions defined in Listing 13:

Listing 13. Memoization of flyweights

```
def computerOf = {type ->
  def of = [MacBookPro6_2: new Laptop(), SunTower: new Desktop()]
  return of[type]
}

def computerOfType = computerOf.memoize()
```

In Listing 13, the canonical types are defined within the `computerOf` function. To create a memoized instance of the function, I simply call the `memoize()` method, defined by the Groovy runtime.

Listing 14 shows a unit test comparing the invocation of the two approaches:

Listing 14. Comparing approaches

```
@Test
public void flyweight_computers() {
  def bob = new AssignedComputer(ComputerFactory.instance.ofType("MacBookPro6_2"), "Bob")
  def steve = new AssignedComputer(ComputerFactory.instance.ofType("MacBookPro6_2"),
  "Steve") assertTrue bob.computerType == steve.computerType

  def sally = new AssignedComputer(computerOfType("MacBookPro6_2"), "Sally")
  def betty = new AssignedComputer(computerOfType("MacBookPro6_2"), "Betty")
  assertTrue sally.computerType == betty.computerType
}
```

The end result is the same, but notice the huge difference in implementation details. For the "traditional" design pattern, I created a new class to act as a factory,

implementing two patterns. For the functional version, I implemented a single method, then returned a memoized version. Offloading details like caching to the runtime means fewer opportunities for hand-written implementations to fail. In this case, I preserved the semantics of the Flyweight pattern but with a very simple implementation.

Conclusion

In this installment, I've introduced the three ways that the semantics of design patterns manifest in functional programming. First, they can be absorbed by the language or runtime. I showed examples of this using the Factory, Strategy, Singleton, and Template Method patterns. Second, patterns can preserve their semantics but have completely different implementations; I showed an example of the Flyweight pattern using classes versus using memoization. Third, functional languages and runtimes can have wholly different features, allowing them to solve problems in completely different ways.

In the next installment, I'll continue this investigation of the intersection of design patterns and functional programming and show examples of the third approach.

Resources

Learn

- *The Productive Programmer* (Neal Ford, O'Reilly Media, 2008): Neal Ford's most recent book discusses tools and practices that help you improve your coding efficiency.
- *Design Patterns: Elements of Reusable Object-Oriented Software* (Erich Gamma et al., Addison-Wesley, 1994): The Gang of Four's classic work on design patterns.
- Design Patterns in Dynamic Languages: Peter Norvig's presentation makes a case that powerful languages (such as functional languages) have less need for design patterns.
- Groovy: Groovy is a multiparadigm JVM language with syntax very close to Java. Its many advanced features include many functional-programming augmentations.
- Scala: Scala is a modern functional language on the JVM.
- *The busy Java developer's guide to Scala*: Dig more deeply into Scala in this developerWorks series by Ted Neward.
- Template method pattern: Template Method is a well-known Gang of Four design pattern.
- Singleton pattern: Singleton is another well-known Gang of Four design pattern.
- Closure memoization: Check out the release notes for the memoization feature added to Groovy 1.8.
- Browse the technology bookstore for books on these and other technical topics.
- developerWorks Java technology zone: Find hundreds of articles about every aspect of Java programming.

Get products and technologies

- Evaluate IBM products in the way that suits you best: Download a product trial, try a product online, use a product in a cloud environment, or spend a few hours in the SOA Sandbox learning how to implement Service Oriented Architecture efficiently.

Discuss

- Get involved in the developerWorks community. Connect with other developerWorks users while exploring the developer-driven blogs, forums, groups, and wikis.

About the author

Neal Ford

Neal Ford is a software architect and Meme Wrangler at **Thought**Works, a global IT consultancy. He also designs and develops applications, instructional materials, magazine articles, courseware, and video/DVD presentations, and he is the author or editor of books spanning a variety of technologies, including the most recent *The Productive Programmer*. He focuses on designing and building large-scale enterprise applications. He is also an internationally acclaimed speaker at developer conferences worldwide. Check out his Web site.