Functional thinking: Coupling and composition, Part 2

Object-oriented vs. functional building blocks

Neal Ford
Software Architect / Meme Wrangler
ThoughtWorks Inc.

04 October 2011

Programmers accustomed to using the building blocks of object orientation (inheritance, polymorphism, and so on) can become blind both to its shortcomings and to alternatives. Functional programming uses different building blocks to achieve reuse, based on more general-purpose concepts like list transformations and portable code. This installment of *Functional thinking* compares coupling via inheritance with composition as reuse mechanisms, and pinpoints one of the key differences between imperative and functional programming.

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.

In the last installment, I illustrated different flavors of code reuse. In the object-oriented version, I extracted duplicated methods, moving them to a superclass along with a `protected` field. In the functional version, I extracted the pure functions (ones that have no side-effects) into their own class, calling them by supplying parameter values. I changed the reuse mechanism from *protected field via inheritance* to *method parameters*. The features (such as inheritance) that comprise object-oriented languages have clear benefits, but they can also have inadvertent side-effects. As some readers accurately commented, many experienced OOP developers have learned *not* to share state via inheritance for that very reason. But if your deeply ingrained paradigm is object orientation, alternatives are sometimes hard to see.

In this installment, I'll contrast coupling via language mechanisms with composition plus portable code as a way to extract reusable code — which also serves to uncover a key philosophical difference about code reuse. First, I'll revisit a classic problem: how to write a proper `equals()` method in the presence of inheritance.

The `equals()` method revisited

Joshua Bloch's book *Effective Java* includes a section on how to craft proper `equals()` and `hashCode()` methods (see Resources). The complication arises from the interaction between equality semantics and inheritance. The `equals()` method in Java must adhere to the characteristics specified by the Javadoc for `Object.equals()`:

- It is *reflexive*: for any non-null reference value x, x.equals(x) should return true.
- It is *symmetric*: for any non-null reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true.
- It is *transitive*: for any non-null reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.
- It is *consistent*: for any non-null reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified.
- For any non-null reference value x, x.equals(null) should return false.

In his example, Bloch creates two classes — a `Point` and a `ColorPoint` — and tries to create an `equals()` method that works correctly for both. Trying to ignore the extra field in the inherited class breaks symmetry, and trying to account for it breaks transitivity. Josh Bloch offers a dire prognosis for this problem:

> *There is simply no way to extend an instantiable class and add an aspect while preserving the equals contract.*

Implementing equality is much simpler when you needn't concern yourself with inherited mutable fields. Adding coupling mechanisms like inheritance creates subtle nuances and traps. (It turns out there's a way to solve this problem that retains inheritance, but at cost of adding an additional dependent method. See the Inheritance and `canEqual()` sidebar.)

Inheritance and `canEqual()`

In *Programming Scala*, the authors provide a mechanism that allows equality to work even in the presence of inheritance (see Resources). The root of the problem Bloch discusses is that parent classes don't "know" enough about subclasses to determine if they should participate in an equality comparison. To address this, you add a `canEqual()` method to the base class and override it for child classes you want equality comparisons for. This allows the current class (via `canEqual()`) to decide if it's reasonable and sensible to equate two types.

This mechanism solves the problem, but at the expense of adding yet another coupling point between the parent and child classes via the `canEqual()` method.

Recall the Michael Feathers quote that leads off the previous two installments of this series:

> Object-oriented programming makes code understandable by *encapsulating* moving parts. Functional programming makes code understandable by *minimizing* moving parts.

The difficulty in implementing `equals()` illustrates Feathers' *moving parts* metaphor. Inheritance is a coupling mechanism: it binds two entities together with well-defined rules about visibility, method dispatch, and so on. In languages like Java, polymorphism is also tied to inheritance. Those

coupling points are what make Java an object-oriented language. But allowing moving parts has consequences, especially at the language level. Helicopters are notoriously hard to fly because a control exists for each of the pilot's four limbs. Moving one control affects the other controls, so the pilot must become adept at dealing with the side-effects that each control has on the others. Language parts are like helicopter controls: you can't readily add (or change) them with affecting all the other parts.

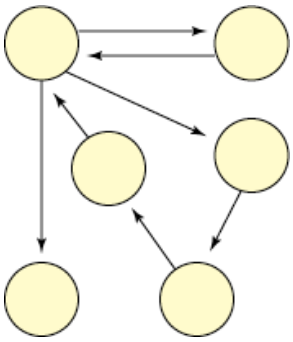
Inheritance is such a natural part of object-oriented languages that most developers lose sight of the fact that, at its heart, it is a coupling mechanism. When odd things break or don't work, you just learn the (sometimes arcane) rules to mitigate the problem and move on. However, those implicit coupling rules affect the way you think about fundamental aspects of your code, such as how to achieve reuse, extensibility, and equality.

Effective Java probably wouldn't have been so successful if Bloch had left the equality question dangling. Instead, he used it as an opportunity to reintroduce good advice from earlier in the book: *prefer composition to inheritance*. Bloch's solution to the `equals()` problem uses *composition* rather than coupling. It eschews inheritance entirely, making `ColorPoint` *own* a reference to an instance of `Point` rather than become a type of point.

Composition and inheritance

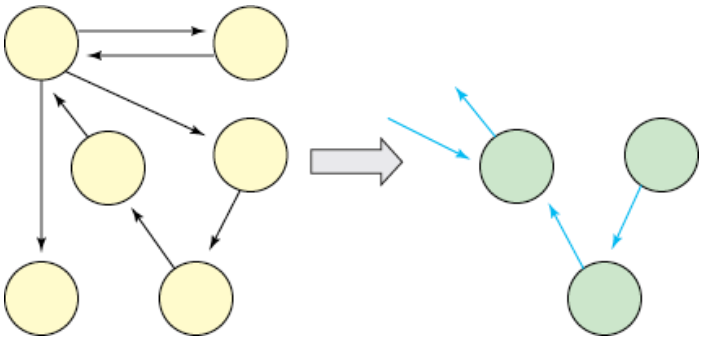
Composition — in the form of passed parameters plus first-class functions — appears frequently in functional programming libraries as a reuse mechanism. Functional languages achieve reuse at a coarser-grained level than object-oriented languages, extracting common machinery with parameterized behavior. Object-oriented systems consist of objects that communicate by sending messages to (or, more specifically, executing methods on) other objects. Figure 1 represents an object-oriented system:

Figure 1. Object-oriented system



When you discover a useful collection of classes and their corresponding messages, you extract that graph of classes for reuse, as shown in Figure 2:

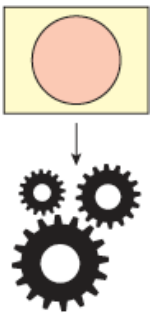
Figure 2. Extracting useful pieces of the graph



Not surprisingly, one of most popular books in the software-engineering world is *Design Patterns: Elements of Reusable Object-Oriented Software* (see Resources), a catalog of exactly the type of extraction shown in Figure 2. Reuse via patterns is so pervasive that numerous other books also catalog (and provide distinct names for) such extractions. The design-patterns movement has been a tremendous boon to the software development world because it supplies nomenclature and exemplars. But, fundamentally, reuse via design patterns is fine-grained: one solution (the Flyweight Pattern, for example) is orthogonal to another (the Memento Pattern). Each of the problems solved by design patterns is highly specific, which makes patterns useful because you can often find a pattern that matches your current problem — but narrowly useful because it is so specific to the problem.

Functional programmers also want reusable code, but they use different building blocks. Rather than try to create well-known relationships (coupling) between structures, functional programming tries to extract coarse-grained reuse mechanisms — based in part on *category theory*, a branch of mathematics that defines relationships (morphism) between types of objects (see Resources). Most applications do things with lists of elements, so a functional approach is to build reuse mechanisms around the idea of lists plus contextualized, portable code. Functional languages rely on *first-class functions* (functions that can appear anywhere any other language construct can appear) as parameters and return values. Figure 3 illustrates this concept:

Figure 3. Reuse via coarse-grained mechanisms plus portable code



In Figure 3, the gear box represents abstractions that deal generically with some fundamental data structure, and the yellow box represents portable code, encapsulating data inside it.

Common building blocks

In the second installment of this series, I constructed a number-classifier example using the Functional Java library (see Resources). That example uses three different building blocks, but without explanation. I'll investigate those building blocks now.

Folds

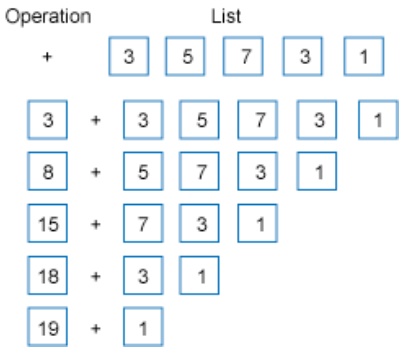
One of the methods in the number classifier performs a sum across all the gathered factors That method appears in Listing 1:

Listing 1. The `sum()` method from the functional number classifier

```
public int sum(List<Integer> factors) {
    return factors.foldLeft(fj.function.Integers.add, 0);
}
```

At first it's not obvious how the one-line body in Listing 1 performs a sum operation. This example is a specific type in the general family of list transformations called *catamorphisms* — transformations from one form to another (see Resources). In this case, the *fold* operation refers to a transformation that combines each element of the list with the next one, accumulating a single result for the entire list. A *fold left* collapses the list leftward, starting with a seed value and combining each element of the list in turn to yield a final result. Figure 4 illustrates a fold operation:

Figure 4. Fold operation



Because addition is commutative, it doesn't matter if you do a `foldLeft()` or `foldRight()`. But some operations (including subtraction and division) care about order, so the symmetrical `foldRight()` method exists to handle those cases.

Listing 1 uses the Functional Java supplied `add` enumeration; it includes the most common mathematical operations for you. But what about cases in which you need more-refined criteria? Consider the example in Listing 2:

Listing 2. `foldLeft()` with user-supplied criteria

```
static public int addOnlyOddNumbersIn(List<Integer> numbers) {
    return numbers.foldLeft(new F2<Integer, Integer, Integer>() {
        public Integer f(Integer i1, Integer i2) {
            return (!(i2 % 2 == 0)) ? i1 + i2 : i1;
        }
    }, 0);
}
```

Because Java doesn't yet have first-class functions in the form of lambda blocks (see Resources), Functional Java is forced to improvise with generics. The built-in `F2` class has the correct structure for a fold operation: it creates a method that accepts two integer parameters (these are the two values being folded upon one another) and the return type. The example in Listing 2 sums odd numbers by returning the sum of both numbers only if the second number is odd, otherwise returning only the first number.

Filtering

Another common operation on lists is *filtering*: creating a smaller list by filtering items in a list based on some user-defined criteria. Filtering is illustrated in Figure 5:

Figure 5. Filtering a list

When filtering, you produce another list (or collection) potentially smaller thanTThis example the original, depending on the filtering criteria. In the number-classifier example, I use filtering to determine the factors of a number, as shown in Listing 3:

Listing 3. Using filtering to determine factors

```
public boolean isFactor(int number, int potential_factor) {
    return number % potential_factor == 0;
}

public List<Integer> factorsOf(final int number) {
    return range(1, number + 1)
            .filter(new F<Integer, Boolean>() {
                public Boolean f(final Integer i) {
                    return isFactor(number, i);
                }
            });
}
```

The code in Listing 3 creates a range of numbers (as a `List`) from 1 up to the target number, then applies the `filter()` method, using the `isFactor()` method (defined at the top of the listing) to eliminate numbers that aren't factors of the target number.

The same functionality shown in Listing 3 can be achieved much more concisely in a language that has closures. A Groovy version appears in Listing 4:

Listing 4. Groovy version of a filtering operation

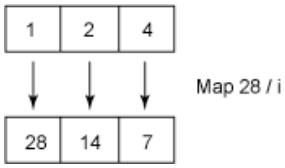
```
def isFactor(number, potential) {
  number % potential == 0;
}

def factorsOf(number) {
  (1..number).findAll { i -> isFactor(number, i) }
}
```

The Groovy version of `filter()` is `findAll()`, which accepts a code block specifying your filter criteria. The last line of the method is the method's return value, which is the list of factors in this case.

Mapping

The *map* operation transforms a collection into a new collection by applying a function to each of the elements, as illustrated in Figure 6:

Figure 6. Mapping a function onto a collection



In the number-classifier example, I use mapping in the optimized version of the `factorsOf()` method, shown in Listing 5:

Listing 5. Optimized factors-finder method using Functional Java's `map()`

```
public List<Integer> factorsOfOptimized(final int number) {
    final List<Integer> factors = range(1, (int) round(sqrt(number) + 1))
            .filter(new F<Integer, Boolean>() {
                public Boolean f(final Integer i) {
                    return isFactor(number, i);
                }
            });
    return factors.append(factors.map(new F<Integer, Integer>() {
        public Integer f(final Integer i) {
            return number / i;
        }
    }))
   .nub();
}
```

The code in Listing 5 first gathers the list of factors up to the square root of the target number, saving it in the `factors` variable. I then append a new collection onto `factors` — generated by the `map()` function on the `factors` list — applying the code to generate the symmetrical (the matching factor above the square root) list. The last `nub()` method ensures that no duplicates exist in the list.

As usual, the Groovy version is much more straightforward, as shown in Listing 6, because flexible types and code blocks are first-class citizens:

Listing 6. Groovy optimized factors

```
def factorsOfOptimized(number) {
  def factors = (1..(Math.sqrt(number))).findAll { i -> isFactor(number, i) }
  factors + factors.collect({ i -> number / i})
}
```

The method names differ, but the code in Listing 6 performs the same task as the code in Listing 5: get a range of numbers from 1 to the square root, filter it down to factors, then append a list to it by mapping each of the list values with the function that yields the symmetrical factor.

Functional perfection revisited

With the availability of higher-order functions, the entire problem of determining if a number is perfect or not collapses to a couple of lines of code in Groovy, as shown in Listing 7:

Listing 7. Groovy perfect-number finder

```
def factorsOf(number) {
  (1..number).findAll { i -> isFactor(number, i) }
}

def isPerfect(number) {
    factorsOf(number).inject(0, {i, j -> i + j}) == 2 * number
}
```

This is of course a contrived example relating to number classification, so it's hard to generalize to different types of code. However, I've noticed a significant change in coding style on projects using languages that support these abstractions (whether they are functional languages or not). I first noticed this on Ruby on Rails projects. Ruby has these same list-manipulation methods that use closure blocks, and I was struck by how often `collect()`, `map()`, and `inject()` appear. Once you become accustomed to having these tools in your toolbox, you'll find yourself turning to them again and again.

Conclusion

One of the challenges of learning a new paradigm like functional programming is learning the new building blocks and "seeing" them peek out of problems as a potential solution. In functional programming, you have far fewer abstractions, but each one is generic (with specificity added via first-class functions). Because functional programming relies heavily on passed parameters and composition, you have fewer rules to learn about the interactions among moving parts, making your job easier.

Functional programming achieves code reuse by abstracting out generic pieces of machinery, customizable via higher-order functions. This article highlighted some of the difficulties introduced by inherent coupling mechanisms in object-oriented languages, which led to a discussion of the common way graphs of classes are harvested to yield reusable code. This is the domain of design patterns. I then showed how coarse-grained mechanisms, based on category theory, that allow you to leverage the code written (and debugged) by the language designer to solve problems. In each case, the solution is succinct and declarative. This illustrates code reuse by composing parameters and functionality to create generic behavior.

In the next installment, I'll delve further into the functional features of a couple of dynamic languages on the JVM: Groovy and JRuby.

Resources

Learn

- *Effective Java* (Joshua Bloch, Addison-Wesley, 2001): Bloch's book is a seminal work on how to use the Java language correctly.
- *Programming in Scala*, 1st ed. (Martin Odersky, Lex Spoon, and Bill Venners): This book is available online. The excellent second edition is available at booksellers everywhere.
- *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.
- Category theory: Category theory is a branch of mathematics the covers in an abstract way the properties of particular mathematical concepts.
- Catamorphism: A catamorphism denotes the unique mapping from one algebra to another algebra.
- "Language designer's notebook: First, do no harm" (Brian Goetz, developerWorks, July 2011): Read about the design considerations behind *lambda expressions*, a new language feature in the works for Java SE 8. Lambda expressions are function literals — expressions that embody a deferred computation that can be treated as a value and invoked later.
- 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

- Functional Java: Functional Java is a framework that adds many functional language constructs to Java.
- Download IBM product evaluation versions or explore the online trials in the IBM SOA Sandbox and get your hands on application development tools and middleware products from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.

Discuss

- Get involved in the developerWorks community.

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.