# Functional thinking: Coupling and composition, Part 1

## Exploring the implications of natively coupled abstractions

Neal Ford
Software Architect / Meme Wrangler
ThoughtWorks Inc.

30 August 2011

Working every day in a particular abstraction (such as object orientation) makes it hard to see when that abstraction is leading you to a solution that isn't the best alternative. This article is the first of two that explores some implications of object-oriented thinking for code reuse, comparing them to more-functional alternatives such as composition.

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.

*Object-oriented programming makes code understandable by encapsulating moving parts. Functional programming makes code understandable by* minimizing *moving parts.*
— Michael Feathers, author of *Working with Legacy Code*, via Twitter

Working in a particular abstraction every day causes it to seep gradually into your brain, influencing the way you solve problems. One of the goals of this series is to illustrate a functional way of looking at typical problems. For this and the next installment, I tackle code reuse via refactoring and the attendant abstraction impact.

One of the goals of object orientation is to make encapsulating and working with state easier. Thus, its abstractions tend toward using state to solve common problems, implying the use of multiple classes and interactions — what the quote above by Michael Feathers calls "moving parts." Functional programming tries to minimize moving parts by *composing* parts together rather than *coupling* structures together. This is a subtle concept that's hard to see for developers whose primary experience is with object-oriented languages.

## Code reuse via structure

The imperative (and especially) object-oriented programming style uses structure and messaging as building blocks. To reuse object-oriented code, you extract the target code into another class, then use inheritance to access it.

### Inadvertent code duplication

To illustrate code reuse and its implications, I return to a version of the number classifier that previous installments use to illustrate code structure and style. The classifier determines if a positive integer is *abundant*, *perfect*, or *deficient*. If the sum of the number's factors is greater than two times the number, it is abundant; if the sum equals two times the number, it is perfect; otherwise (if the sum is less than two times the number), it is deficient.

You could also write code that uses a positive integer's factors to determine if it is a prime number (defined as an integer greater than 1 whose only factors are 1 and the number itself). Because both of these problems rely on a number's factors, they are good candidates for refactoring (no pun intended) and thus for illustrating styles of code reuse.

Listing 1 shows the number classifier written in an imperative style:

### Listing 1. Imperative number classifier

```
import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;

import static java.lang.Math.sqrt;

public class ClassifierAlpha {
    private int number;

    public ClassifierAlpha(int number) {
        this.number = number;
    }

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

    public Set<Integer> factors() {
        HashSet<Integer> factors = new HashSet<Integer>();
        for (int i = 1; i <= sqrt(number); i++)
            if (isFactor(i)) {
                factors.add(i);
                factors.add(number / i);

            }
        return factors;
    }

    static public int sum(Set<Integer> factors) {
        Iterator it = factors.iterator();
        int sum = 0;
        while (it.hasNext())
            sum += (Integer) it.next();
        return sum;
    }

    public boolean isPerfect() {
        return sum(factors()) - number == number;
    }

    public boolean isAbundant() {
        return sum(factors()) - number > number;
    }

    public boolean isDeficient() {
        return sum(factors()) - number < number;
    }

}
```

I discuss the derivation of this code in the first installment, so I won't repeat it now. Its purpose here is to illustrate code reuse. That leads me to the code in Listing 2, which tests for prime numbers:

## Listing 2. Prime-number testing, written imperatively

```
import java.util.HashSet;
import java.util.Set;

import static java.lang.Math.sqrt;

public class PrimeAlpha {
    private int number;

    public PrimeAlpha(int number) {
        this.number = number;
    }

    public boolean isPrime() {
        Set<Integer> primeSet = new HashSet<Integer>() {{
            add(1); add(number);}};
        return number > 1 &&
                factors().equals(primeSet);
    }

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

    public Set<Integer> factors() {
        HashSet<Integer> factors = new HashSet<Integer>();
        for (int i = 1; i <= sqrt(number); i++)
            if (isFactor(i)) {
                factors.add(i);
                factors.add(number / i);
            }
        return factors;
    }
}
```

A few items of note appear in Listing 2. The first is the slightly odd initialization code in the `isPrime()` method. This is an example of an *instance initializer*. (For more on instance initialization — a Java technique that's incidental to functional programming — see "Evolutionary architecture and emergent design: Leveraging reusable code, Part 2.")

The other items of interest in Listing 2 are the `isFactor()` and `factors()` methods. Notice that they are identical to their counterparts in the `ClassifierAlpha` class (in Listing 1). This is the natural outcome of implementing two solutions independently and realizing that you have virtually the same functionality.

## Refactoring to eliminate duplication

The solution to this type of duplication is to refactor the code into a single `Factors` class, which appears in Listing 3:

### Listing 3. Common refactored factoring code

```
import java.util.Set;
import static java.lang.Math.sqrt;
import java.util.HashSet;

public class FactorsBeta {
    protected int number;

    public FactorsBeta(int number) {
        this.number = number;
    }

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

    public Set<Integer> getFactors() {
        HashSet<Integer> factors = new HashSet<Integer>();
        for (int i = 1; i <= sqrt(number); i++)
            if (isFactor(i)) {
                factors.add(i);
                factors.add(number / i);
            }
        return factors;
    }
}
```

The code in Listing 3 is the result of using *Extract Superclass* refactoring. Notice that because both of the extracted methods use the `number` member variable, it is dragged into the superclass. While performing this refactoring, the IDE asked me how I would like to handle access (accessor pair, protected scope, and so on). I chose protected scope, which adds `number` to the class and creates a constructor to set its value.

Once I've isolated and removed the duplicated code, both the number classifier and the prime-number tester are much simpler. Listing 4 shows the refactored number classifier:

### Listing 4. Refactored, simplified number classifier

```
import java.util.Iterator;
import java.util.Set;

public class ClassifierBeta extends FactorsBeta {

    public ClassifierBeta(int number) {
        super(number);
    }

    public int sum() {
        Iterator it = getFactors().iterator();
        int sum = 0;
        while (it.hasNext())
            sum += (Integer) it.next();
        return sum;
    }

    public boolean isPerfect() {
        return sum() - number == number;
    }

    public boolean isAbundant() {
        return sum() - number > number;
    }

    public boolean isDeficient() {
        return sum() - number < number;
    }

}
```

Listing 5 shows the refactored prime-number tester:

### Listing 5. Refactored, simplified prime-number tester

```
import java.util.HashSet;
import java.util.Set;

public class PrimeBeta extends FactorsBeta {
    public PrimeBeta(int number) {
        super(number);
    }

    public boolean isPrime() {
        Set<Integer> primeSet = new HashSet<Integer>() {{
            add(1); add(number);}};
        return getFactors().equals(primeSet);
    }
}
```

Regardless of which access option you choose for the `number` member when refactoring, you must deal with a network of classes when you think about this problem. Often this is a good thing because it allows you to isolate parts of the problem, but it also has downstream consequences when you make changes to the parent class.

This is an example of code reuse via *coupling*: tying two elements (in this case, classes) via the shared state of the `number` field and the `getFactors()` method from the superclass. In other words, this works by using the built-in coupling rules in the language. Object orientation defines coupled interaction styles (how you access member variables via inheritance, for example), so you have predefined rules about how things are coupled — which is good, because you can reason about behavior in a consistent way. Don't misunderstand me — I'm not suggesting that using inheritance is a bad idea. Rather, I'm suggesting that it is overused in object-oriented languages in lieu of alternative abstractions that have better characteristics.

## Code reuse via composition

In the second installment of this series, I presented a functional version of the number classifier in Java, shown in Listing 6:

### Listing 6. More-functional version of number classifier

```
public class FClassifier {

    static public boolean isFactor(int number, int potential_factor) {
        return number % potential_factor == 0;
    }
    static public Set<Integer> factors(int number) {
        HashSet<Integer> factors = new HashSet<Integer>();
        for (int i = 1; i <= sqrt(number); i++)
            if (isFactor(number, i)) {
                factors.add(i);
                factors.add(number / i);
            }
        return factors;
    }

    public static int sumOfFactors(int number) {
        Iterator<Integer> it = factors(number).iterator();
        int sum = 0;
        while (it.hasNext())
            sum += it.next();
        return sum;
    }

    public static boolean isPerfect(int number) {
        return sumOfFactors(number) - number == number;
    }

    public static boolean isAbundant(int number) {
        return sumOfFactors(number) - number > number;
    }

    public static boolean isDeficient(int number) {
        return sumOfFactors(number) - number < number;
    }
}
```

I also have a functional version (using pure functions and no shared state) of the prime-number tester, whose `isPrime()` method appears in Listing 7. The rest of its code is identical to the same-named methods in Listing 6.

## Listing 7. Functional version of prime-number tester

```
public static boolean isPrime(int number) {
    Set<Integer> factors = factors(number);
    return number > 1 &&
            factors.size() == 2 &&
            factors.contains(1) &&
            factors.contains(number);
}
```

Just as I did with the imperative versions, I extract the duplicated code into its own `Factors` class, changing the name of the `factors` method to `of` for readability, as shown in Listing 8:

## Listing 8. The functional refactored `Factors` class

```
import java.util.HashSet;
import java.util.Set;
import static java.lang.Math.sqrt;

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

    static public Set<Integer> of(int number) {
        HashSet<Integer> factors = new HashSet<Integer>();
        for (int i = 1; i <= sqrt(number); i++)
            if (isFactor(number, i)) {
                factors.add(i);
                factors.add(number / i);
            }
        return factors;
    }
}
```

Because all the state in the functional version is passed as parameters, no shared state comes along with the extraction. Once I have extracted this class, I can refactor both the functional classifier and prime-number tester to use it. Listing 9 shows the refactored number classifier:

## Listing 9. Refactored number classifier

```
public class FClassifier {

    public static int sumOfFactors(int number) {
        Iterator<Integer> it = Factors.of(number).iterator();
        int sum = 0;
        while (it.hasNext())
            sum += it.next();
        return sum;
    }

    public static boolean isPerfect(int number) {
        return sumOfFactors(number) - number == number;
    }

    public static boolean isAbundant(int number) {
        return sumOfFactors(number) - number > number;
    }

    public static boolean isDeficient(int number) {
        return sumOfFactors(number) - number < number;
    }
}
```

Listing 10 shows the refactored prime-number tester:

### Listing 10. Refactored prime-number tester

```
import java.util.Set;

public class FPrime {

    public static boolean isPrime(int number) {
        Set<Integer> factors = Factors.of(number);
        return number > 1 &&
                factors.size() == 2 &&
                factors.contains(1) &&
                factors.contains(number);
    }
}
```

Note that I didn't use any special libraries or languages to make the second version more functional. Rather, I did it by using *composition* rather than coupling for code reuse. Both Listing 9 and Listing 10 use the `Factors` class, but its use is entirely contained within individual methods.

The distinction between coupling and composition is subtle but important. In a simple example like this, you can see the skeleton of the code structure showing through. However, when you end up refactoring a large code base, coupling pops up everywhere because that's one of the reuse mechanisms in object-oriented languages. The difficulty of understanding exuberantly coupled structures has harmed reuse in object-oriented languages, limiting effective reuse to well-defined technical domains like object-relational mapping and widget libraries. The same level of reuse has eluded us when we write less obviously structured Java code (such as the code you write in business applications).

You could have made the imperative version better by noticing what the IDE is offering during the refactoring, politely declining, and using composition instead.

## Conclusion

Thinking as a more functional programmer means thinking differently about all aspects of coding. Code reuse is an obvious development goal, and imperative abstractions tend to solve that problem differently from the way that functional programmers solve it. This installment contrasted the two styles of code reuse: coupling via inheritance and composition via parameters. The next installment will continue to explore this important divide.

## 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.
- Stuart Halloway on Clojure: Learn more about Clojure — a modern, functional Lisp that runs on the JVM — from this developerWorks podcast.
- *The busy Java developer's guide to Scala*: Scala is another modern, functional language for the JVM. Read about it in this developerWorks series by Ted Neward.
- 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. 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.