Functional thinking: Functional design patterns, Part 2

Same problem, different paradigms

Neal Ford
Software Architect / Meme Wrangler
ThoughtWorks Inc.

03 April 2012

Design patterns represent just one way to solve problems, but if you primarily use object-oriented languages, you may come to *think* in design patterns. In this installment of *Functional thinking*, Neal Ford illustrates solutions to a common problem — incompatible interfaces — using traditional design patterns, metaprogramming, and functional composition. Each approach has pros and cons, but thinking about the design of the solutions helps you see problems in new ways.

View more content in this series

In the last installment, I began investigating the intersection of traditional Gang of Four (GoF) design patterns (see Resources) and more-functional approaches. I continue that journey in this installment, showing the solution to a common problem in three different paradigms: patterns, metaprogramming, and function composition.

If the primary paradigm your language supports is objects, it's easy to start thinking about solutions to every problem in those terms. However, most modern languages are *multiparadigm*, meaning they support object, metaobject, functional, and other paradigms. Learning to use different paradigms for suitable problems is part of the evolution toward being a better developer.

About this series

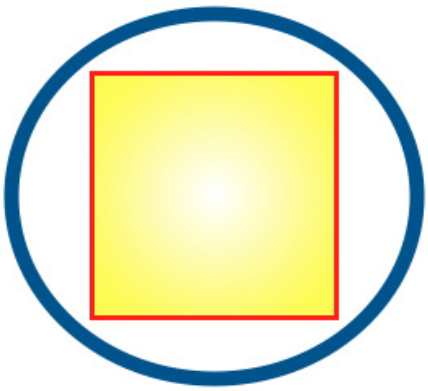
This series aims to reorient your perspective toward a functional mindset, helping you look at common problems in new ways and find ways to improve your day-to-day coding. It explores functional programming concepts, frameworks that allow functional programming within the Java™ language, functional programming 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 this installment, I'm going to attack the traditional problem solved by the *Adapter* design pattern: translating an interface to work with an incompatible one. First, the traditional approach, written in Java.

Adapter in Java

The Adapter pattern translates one interface for a class into a compatible interface. It is used when two classes could conceptually work together but cannot because of implementation details. For this example, I create a few simple classes modeling the problem of fitting a square peg into a round hole. A square peg will sometimes fit into a round hole, as illustrated in Figure 1, depending on the relative sizes of the peg and the hole:

Figure 1. A square peg in a round hole



To determine if a square will fit into a circle, I use the formula shown in Figure 2:

Figure 2. Formula for determining if a square will fit into a circle

$$\sqrt{\left(\frac{w}{2}\right)^2 \times 2}$$

The formula in Figure 2 takes the width of one of the square's sides divided by 2, squares it, multiplies the result by 2, and returns the square root. If this value is less than the radius of the circle, the peg will fit.

I could trivially solve the square peg/round hole problem with a simple utility class that handles conversions. But it exemplifies a larger problem. For example, what if I'm adapting a *Button* to fit on a type of *Panel* that it wasn't designed for yet can be made compatible with? The problem of square pegs and round holes is a convenience simplification of the general problem addressed by the Adapter design pattern: adapting two incompatible interfaces. To enable square pegs to work

with round holes, I need a handful of classes and interfaces to implement the Adapter pattern, as shown in Listing 1:

Listing 1. Square pegs and round holes in Java

```
public class SquarePeg {
    private int width;

    public SquarePeg(int width) {
        this.width = width;
    }

    public int getWidth() {
        return width;
    }
}

public interface Circularity {
    public double getRadius();
}

public class RoundPeg implements Circularity {
    private double radius;

    public double getRadius() {
        return radius;
    }

    public RoundPeg(int radius) {
        this.radius = radius;
    }
}

public class RoundHole {
    private double radius;

    public RoundHole(double radius) {
        this.radius = radius;
    }

    public boolean pegFits(Circularity peg) {
        return peg.getRadius() <= radius;
    }

}
```

To reduce the amount Java code, I've added an interface named `Circularity` to indicate that the implementer has a radius. This lets me write the `RoundHole` code in terms of round things, not just `RoundPeg`s. This is a common concession in the Adapter pattern to make type resolution easier.

To fit square pegs into round holes, I need an adapter that adds `Circularity` to `SquarePeg`s by exposing a `getRadius()` method, as shown in Listing 2:

Listing 2. Square peg adapter

```
public class SquarePegAdaptor implements Circularity {
    private SquarePeg peg;

    public SquarePegAdaptor(SquarePeg peg) {
        this.peg = peg;
    }

    public double getRadius() {
        return Math.sqrt(Math.pow((peg.getWidth()/2), 2) * 2);
    }
}
```

To test that my adapter does in fact let me fit suitably sized square pegs in round holes, I implement the test shown in Listing 3:

Listing 3. Testing adaptation

```
@Test
public void square_pegs_in_round_holes() {
    RoundHole hole = new RoundHole(4.0);
    Circularity peg;
    for (int i = 3; i <= 10; i++) {
        peg = new SquarePegAdaptor(new SquarePeg(i));
        if (i < 6)
            assertTrue(hole.pegFits(peg));
        else
            assertFalse(hole.pegFits(peg));
    }
}
```

In Listing 3, for each of the proposed widths, I wrap the `SquarePegAdaptor` around the creation of the `SquarePeg`, enabling `hole`'s `pegFits()` method to return an intelligent evaluation as to the fitness of the peg.

This code is straightforward, because this is a simple albeit verbose pattern to implement in Java. This paradigm is clearly the GoF design-pattern approach. However, the pattern approach isn't the only way.

Dynamic adapter in Groovy

Groovy (see Resources) supports several programming paradigms that Java does not, so I will use it for the remainder of the examples. First, I'll implement the "standard" Adapter pattern solution from Listing 2 as ported to Groovy, shown in Listing 4:

Listing 4. Pegs, holes, and adapters in Groovy

```
class SquarePeg {
    def width
}

class RoundPeg {
    def radius
}

class RoundHole {
    def radius
```

```
    def pegFits(peg) {
        peg.radius <= radius
    }
}

class SquarePegAdapter {
    def peg

    def getRadius() {
        Math.sqrt(((peg.width/2) ** 2)*2)
    }
}
```

The most noticeable difference between the Java version in Listing 2 and the Groovy version in Listing 4 is verbosity. Groovy was designed to remove some of Java's repetitiveness via dynamic typing and conveniences such as allowing the last line of the method to serve as the return value for the method automatically, as illustrated in the `getRadius()` method.

The test for the Groovy version of the adapter appears in Listing 5:

Listing 5. Testing traditional adapter in Groovy

```
@Test void pegs_and_holes() {
    def hole = new RoundHole(radius:4.0)
    (4..7).each { w ->
        def peg = new SquarePegAdapter(
                peg:new SquarePeg(width:w))
        if (w < 6 )
            assertTrue hole.pegFits(peg)
        else
            assertFalse hole.pegFits(peg)
    }
}
```

In Listing 5, I take advantage of another of Groovy's conveniences, calling the name/value constructor that Groovy generated automatically when I construct both `RoundHole`, `SquarePegAdaptor`, and `SquarePeg`.

Despite the syntactic sugar, this version is just like the Java version and follows the GoF design-pattern paradigm. It is common for Groovy developers coming from a Java background to port their old experience to a new syntax. However, Groovy has more-elegant ways to solve this problem, using metaprogramming.

Using metaprogramming for adaptation

One of the outstanding features of Groovy is its powerful support for metaprogramming. I'll use it to build the adapter directly into the class via `ExpandoMetaClass`.

`ExpandoMetaClass`

A common feature of dynamic languages is *open classes*: the ability to reopen existing classes (either yours or system classes like `String` or `Object`) to add, remove, or change methods. Open classes are frequently used in domain-specific languages (DSLs) and for building fluent interfaces. Groovy has two mechanisms for open classes: *categories* and `ExpandoMetaClass`. My example shows only the expando syntax.

`ExpandoMetaClass` enables you to add new methods to classes or individual object instances. In the case of adapting, I need to add "radiusness" to my `SquarePeg` before I can check to see if it will fit in the round hole, as illustrated in Listing 6:

Listing 6. Using `ExpandoMetaClass` to add radius to square pegs

```
static {
    SquarePeg.metaClass.getRadius = { ->
        Math.sqrt(((delegate.width/2) ** 2)*2)
    }
}

@Test void expando_adapter() {
    def hole = new RoundHole(radius:4.0)
    (4..7).each { w ->
        def peg = new SquarePeg(width:w)
        if (w < 6)
            assertTrue hole.pegFits(peg)
        else
            assertFalse hole.pegFits(peg)
    }
}
```

Every class in Groovy has a predefined `metaClass` property, exposing that class's `ExpandoMetaClass`. In Listing 6, I use that property to add a `getRadius()` method, using the familiar formula, to the `SquarePeg` class. Timing is important when you use `ExpandoMetaClass`; I must ensure that the method has been added before I try to invoke it in the unit test. Thus, I add the new method in the static initializer of the test class, which adds the method to `SquarePeg` as the test class is loaded. After the `getRadius()` method has been added to `SquarePeg`, I can pass it into the `hole.pegFits` method, and Groovy's dynamic typing takes care of the rest.

Using the `ExpandoMetaClass` is certainly more succinct than the longer pattern version. And it's practically invisible -- which is also one of its disadvantages. Adding methods wholesale to existing classes should be done with care, because you trade convenience for invisible behavior, which can be hard to debug. This is acceptable in some cases, as in DSLs and for pervasive changes to existing infrastructure on behalf of frameworks.

This example illustrates using the metaprogramming paradigm — modifying the existing class — to solve the adapter problem. However, this isn't the only way to use Groovy's dynamism to solve this problem.

Dynamic adapters

Groovy is optimized to integrate nicely with Java, including places where Java is relatively rigid. For example, dynamically generating classes is cumbersome in Java, but Groovy handles it easily. This suggests that I can generate an adapter class on the fly, as shown in Listing 7:

Listing 7. Using dynamic adapters

```
def roundPegOf(squarePeg) {
    [getRadius:{Math.sqrt(
                ((squarePeg.width/2) ** 2)*2)}] as RoundThing
}

@Test void functional_adaptor() {
    def hole = new RoundHole(radius:4.0)
    (4..7).each { w ->
        def peg = roundPegOf(new SquarePeg(width:w))
        if (w < 6)
            assertTrue hole.pegFits(peg)
        else
            assertFalse hole.pegFits(peg)
    }
}
```

Groovy's literal hash syntax uses square braces, which appear within the `roundPegOf()` method in Listing 7. To generate a class that implements an interface, Groovy lets you create a hash, with method names as the keys and implementation code blocks for the values. The `as` operator uses the hash to generate a class that implements the interface, using the hash's key names to generate instance methods. Thus, in Listing 7, the `roundPegOf()` method creates a single-entry hash with `getRadius` as the method name (Groovy's hash keys aren't required to use double quotes when they are strings) and my familiar conversion code as the implementation. The `as` operator converts this into a class that implements the `RoundThing` interface, acting as the adapter wrapped around the `SquarePeg` creation within the `functional_adaptor()` test.

The ability to generate a class on the fly removes much of the verbosity and formality of the traditional patterns approach. It's also more explicit than the metaprogramming approach: I'm not adding new methods to the class; I'm generating a just-in-time wrapper for adaptation purposes. This uses the design-pattern paradigm (adding an adapter class), but with minimal fuss and syntax.

Functional adapters

When all you own is a hammer, every problem looks like a nail. If your only paradigm is object orientation, you can lose the ability to see alternate possibilities. One of the hazards of spending too much time in languages without first-class functions is the over-application of patterns to solve problems. Many patterns (Observer, Visitor, and Command, to name a few examples) are at their heart mechanisms for applying portable code, implemented in languages that lack higher-order functions. I can discard much of the object trapping and just write a function to handle the conversion. And it turns out that this approach has some advantages.

Functions!

If you have first-class functions (functions that can appear anywhere any other language construct can appear, including outside classes), you can write a conversion function that handles adaptation for you, as illustrated by the Groovy code in Listing 8:

Listing 8. Using a simple conversion function

```
def pegFits(peg, hole) {
    Math.sqrt(((peg.width/2) ** 2)*2) <= hole.radius
}

@Test void functional_all_the_way() {
    def hole = new RoundHole(radius:4.0)
    (4..7).each { w ->
        def peg = new SquarePeg(width:w)
         if (w < 6)
            assertTrue pegFits(peg, hole)
        else
            assertFalse pegFits(peg, hole)
    }
}
```

In Listing 8, I create a function that accepts a `peg` and a `hole`, and use it to check the fitness of the peg. The approach works, but it removes the decision about fitness from the hole, where object orientation has deemed it belongs. In some cases, it might make sense to externalize that decision rather than adapt the class. This represents the functional paradigm: pure functions that accept parameters and return results.

Composition

Before leaving the functional approach, I'll show my favorite adapter, which merges the design-pattern and functional approaches. To illustrate an advantage of using lightweight dynamic generators delivered as first-class functions, consider the example in Listing 9:

Listing 9. Composing functions via lightweight dynamic adapters

```
class CubeThing {
    def x, y, z
}

def asSquare(peg) {
    [getWidth:{peg.x}] as SquarePeg
}
def asRound(peg) {
    [getRadius:{Math.sqrt(
                ((peg.width/2) ** 2)*2)}] as RoundThing
}

@Test void mixed_functional_composition() {
    def hole = new RoundHole(radius:4.0)
    (4..7).each { w ->
        def cube = new CubeThing(x:w)
         if (w < 6)
            assertTrue hole.pegFits(asRound(asSquare(cube)))
        else
            assertFalse hole.pegFits(asRound(asSquare(cube)))
    }
}
```

In Listing 9, I create little functions that return dynamic adapters that let me chain the adapters together in a convenient, readable way. *Composing* functions lets functions control and encapsulate what happens to their parameters without worrying about who might be using them as a parameter. This is very much a functional approach, using Groovy's ability to create dynamic wrapper classes as the implementation.

Contrast the lightweight dynamic adapter approach with the clunky version of adapter composition in the Java I/O libraries, shown in Listing 10:

Listing 10. Clunky adapter composition

```
ZipInputStream zis =
    new ZipInputStream(
        new BufferedInputStream(
            new FileInputStream(argv[0])));
```

The example in Listing 10 shows a common problem that adapters address: the ability to mix and match composed behavior. Lacking first-class functions, Java is forced to do composition via constructors. Using functions to wrap other functions and modifying their return is common in functional programming but less so in Java, because the language adds friction in the form of excessive syntax.

Conclusion

If you always stay trapped in the same paradigm, it becomes difficult to see the benefits to alternative approaches, because it doesn't fit your world view. Modern mixed-paradigm languages offer a palette of design choices, and understanding how each paradigm works (and interacts with the other paradigms) helps you choose better solutions. In this installment, I illustrated the common problem of adaptability and solved it via the traditional adapter design pattern in Java and Groovy. Next, solved the problem using Groovy metaprogramming and the `ExpandoMetaClass`, and then showed dynamic adapter classes. You also saw that having a lightweight syntax for adapter classes enables convenient function composition, which is cumbersome in Java.

In the next installment, I continue exploration at the intersection of design patterns and functional programming.

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.
- Adapter pattern: Adapter is a well-known Gang of Four design pattern.
- Groovy: Groovy is a dynamic language on the JVM with powerful metaprogramming and functional constructs.
- "*Practically Groovy*: Metaprogramming with closures, ExpandoMetaClass, and categories": Read more about Groovy's ability to add new methods to classes dynamically at run time.
- 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.