Functional thinking: Functional design patterns, Part 3

The Interpreter pattern and extending the language

Neal Ford
Software Architect / Meme Wrangler
ThoughtWorks Inc.

Skill Level: Intermediate

Date: 15 May 2012

The Gang of Four's Interpreter design pattern encourages extending a language by building a new language from it. Most functional languages let you extend the language in a variety of ways, such as operator overloading and pattern matching. Although Java™ doesn't permit any of these techniques, next-generation JVM languages do, with varying implementation details. In this article, Neal Ford investigates how Groovy, Scala, and Clojure realize the intent of the Interpreter design pattern by allowing functional extensions in ways that Java does not.

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.

This *Functional thinking* installment continues my investigation of alternate, functional solutions to Gang of Four (GoF) design patterns (see Resources). In this article, I investigate the least understood but most powerful of those patterns: *Interpreter*.

The definition of Interpreter is:

> Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

In other words, if the language you are using isn't appropriate for the problem, use it to build a language that is. Good examples of this approach appear in web frameworks like Grails and Ruby on Rails (see Resources), which extend their base languages (Groovy and Ruby, respectively) to make it easier to write web applications.

This pattern is the *least understood* because it's uncommon to build a new language, so the skills and idioms required are specialized. It is the most *powerful* of the design patterns because it encourages you to extend your programming language toward the problem you are solving. This is a common ethos in the Lisp (and therefore Clojure) worlds but less common in mainstream languages.

When using languages (such as Java) that disallow extensions to the language itself, developers tend to mold their thoughts into the syntax of the language; it's your only choice. However, when you become accustomed to working in languages that allow painless extension, you start bending the language toward the problem solution, not the other way around.

Java lacks straightforward language-extension mechanisms unless you resort to aspect-oriented programming. However, the next-generation JVM languages (Groovy, Scala, and Clojure) (see Resources) all allow extensions in a variety of ways. In doing so, they meet the intent of the Interpreter design pattern. First, I show how to implement operator overloading in all three languages, then show how Groovy and Scala let you extend existing classes.

Operator overloading

A common feature of functional languages is *operator overloading* — the ability to redefine operators (such as `+`, `-`, or `*`) to work with new types and exhibit new behaviors. Omission of operator overloading was a conscious decision during Java's formative period, but virtually every modern language now features it, including the natural successors to Java on the JVM.

Groovy

Groovy tries to update Java's syntax to the current century while preserving its natural semantics. Thus, Groovy allows operator overloading by automatically mapping operators to method names. For example, if you want to overload `Integer`'s `+` operator, you override the `Integer` class's `plus()` method. The entire list of mappings is available online (see Resources); Table 1 shows a partial list:

Table 1. Partial list of Groovy operator/method mappings

Operator	Method
`x + y`	`x.plus(y)`
`x * y`	`x.multiply(y)`
`x / y`	`x.div(y)`

`x ** y`	`x.power(y)`

As an example of operator overloading, I'll create a `ComplexNumber` class in both Groovy and Scala. *Complex numbers* are a mathematical concept with both a *real* and *imaginary* part, typically written as, for example, `3 + 4i`. Complex numbers are common in many scientific fields, including engineering, physics, electromagnetism, and chaos theory. Developers writing applications in those fields greatly benefit from the ability to create operators that mirror their problem domain. (For more information on complex numbers, see Resources.)

A Groovy `ComplexNumber` class appears in Listing 1:

Listing 1. `ComplexNumber` in Groovy

```
package complexnums

class ComplexNumber {
   def real, imaginary

  public ComplexNumber(real, imaginary) {
    this.real = real
    this.imaginary = imaginary
  }

  def plus(rhs) {
    new ComplexNumber(this.real + rhs.real, this.imaginary + rhs.imaginary)
  }

  def multiply(rhs) {
    new ComplexNumber(
        real * rhs.real - imaginary * rhs.imaginary,
        real * rhs.imaginary + imaginary * rhs.real)
  }

  String toString() {
    real.toString() + ((imaginary < 0 ? "" : "+") + imaginary + "i").toString()
  }
}
```

In Listing 1, I create a class that holds both real and imaginary parts, and I create the overloaded `plus()` and `multiply()` operators. Adding two complex numbers is straightforward: the `plus()` operator adds the two numbers' respective real and imaginary parts to each other to produce the result. Multiplying two complex numbers requires this formula:

```
(x + yi)(u + vi) = (xu - yv) + (xv + yu)i
```

The `multiply()` operator in Listing 1 replicates the formula. It multiplies the numbers' real parts, then subtracts the product of the imaginary parts, which is added to the product of the real and imaginary parts both multiplied by each other.

Listing 2 tests the complex-number operators:

Listing 2. Testing complex-number operators

```
package complexnums
```

```
import org.junit.Test
import static org.junit.Assert.assertTrue
import org.junit.Before

class ComplexNumberTest {
  def x, y

  @Before void setup() {
    x = new ComplexNumber(3, 2)
    y = new ComplexNumber(1, 4)
  }

  @Test void plus_test() {
    def z = x + y;
    assertTrue 3 + 1 == z.real
    assertTrue 2 + 4 == z.imaginary
  }

  @Test void multiply_test() {
    def z = x * y
    assertTrue(-5  == z.real)
    assertTrue 14 == z.imaginary
  }
}
```

In Listing 2, the `plus_test()` and `multiply_test()` methods' use of the overloaded operators — both of which are represented by the same symbols that the domain experts use — is indistinguishable from similar use of built-in types .

Scala (and Clojure)

Scala allows operator overloading by discarding the distinction between operators and methods: operators are merely methods with special names. Thus, to override the multiplication operator in Scala, you override the `*` method. I create complex numbers in Scala in Listing 3:

Listing 3. Complex numbers in Scala

```
class ComplexNumber(val real:Int, val imaginary:Int) {
    def +(operand:ComplexNumber):ComplexNumber = {
        new ComplexNumber(real + operand.real, imaginary + operand.imaginary)
    }

    def *(operand:ComplexNumber):ComplexNumber = {
        new ComplexNumber(real * operand.real - imaginary * operand.imaginary,
            real * operand.imaginary + imaginary * operand.real)
    }

    override def toString() = {
        real + (if (imaginary < 0) "" else "+") + imaginary + "i"
    }
}
```

The class in Listing 3 includes the familiar `real` and `imaginary` members, as well as the `+` and `*` operators/methods. As you can see in Listing 4, I can use `ComplexNumber`s naturally:

Listing 4. Using complex numbers in Scala

```
val c1 = new ComplexNumber(3, 2)
val c2 = new ComplexNumber(1, 4)
val c3 = c1 + c2
assert(c3.real == 4)
assert(c3.imaginary == 6)

val res = c1 + c2 * c3

printf("(%s) + (%s) * (%s) = %s\n", c1, c2, c3, res)
assert(res.real == -17)
assert(res.imaginary == 24)
```

By unifying operators and methods, Scala makes operator overloading trivial. Clojure uses the same mechanism to overload operators. For example, this Clojure code defines an overloaded `**` operator:

```
(defn ** [x y] (Math/pow x y))
```

Extending classes

Similarly to operator overloading, the next-generation JVM languages allow you to extend classes (including core Java classes) in ways that are impossible in the Java language itself. These facilities are often used to build domain-specific languages (DSLs). Although the GoF never considered DSLs — because they weren't common in the popular languages of the time — DSLs exemplify the original purpose of the Interpreter design pattern.

By adding units and other modifiers to core classes such as `Integer`, you can — as with adding operators — model real-world problems more closely. Both Groovy and Scala allow this, but with different mechanisms.

Groovy's `Expando` and category classes

Groovy includes two mechanisms for adding methods to existing classes: `ExpandoMetaClass` and *categories*. (I covered details of the `ExpandoMetaClass` in the last installment, in the context of the Adapter pattern.)

Let's say that your company, for bizarre legacy reasons, needs to express speeds in furlongs per fortnight rather than miles per hour (MPH), and developers find themselves performing this conversion often. Using Groovy's `ExpandoMetaClass`, you can add a `FF` property to `Integer` that handles the conversion, as shown in Listing 5:

Listing 5. Using `ExpandoMetaClass` to add a furlongs/fortnight unit to `Integer`

```
static {
  Integer.metaClass.getFF { ->
    delegate * 2688
  }
}

@Test void test_conversion_with_expando() {
  assertTrue 1.FF == 2688
}
```

The alternative to `ExpandoMetaClass` is to create a *category* wrapper class, a concept borrowed from Objective-C. In Listing 6, I add a (lowercase) `ff` property to `Integer`:

Listing 6. Adding units via a category class

```
class FFCategory {
  static Integer getFf(Integer self) {
    self * 2688
  }
}

@Test void test_conversion_with_category() {
  use(FFCategory) {
    assertTrue 1.ff == 2688
  }
}
```

A category class is a regular class with a collection of static methods. Each method accepts at least one parameter; the first parameter is the type this method augments. For example, in Listing 6, the `FFCategory` class has a `getFf()` method, which accepts an `Integer` parameter. When this category class is used with the `use` keyword, all appropriate types within the code block are augmented. In the unit test, I can reference the `ff` property (remember, Groovy automatically converts `get` methods to property references) within the code block, as shown at the bottom of Listing 6.

Having two mechanisms to choose from lets you control the scope of augmentations more exactly. For example, if the entire system uses MPH as the default unit of speed but also requires frequent conversion to furlongs per fortnight, a global change using the `ExpandoMetaClass` would be appropriate.

You may be skeptical of the usefulness of reopening core JVM classes, worrying about the broad-reaching implications. Category classes let you limit the scope of potentially dangerous enhancements. Here is an example from a real-world open source project that makes excellent use of this mechanism.

The *easyb* project (see Resources) lets you write tests that verify aspects of classes under test. Consider the snippet from an easyb test shown in Listing 7:

Listing 7. easyb testing a `queue` class

```
it "should dequeue items in same order enqueued", {
    [1..5].each {val ->
        queue.enqueue(val)
    }
    [1..5].each {val ->
        queue.dequeue().shouldBe(val)
    }
}
```

The `queue` class doesn't include a `shouldBe()` method, which I call during the verification phase of the test. The easyb framework has added the method for me; the `it()` method definition in easyb's source, shown in Listing 8, shows how:

Listing 8. easyb's `it()` method definition

```
def it(spec, closure) {
  stepStack.startStep(BehaviorStepType.IT, spec)
  closure.delegate = new EnsuringDelegate()
  try {
    if (beforeIt != null) {
      beforeIt()
    }
    listener.gotResult(new Result(Result.SUCCEEDED))
    use(categories) {
      closure()
    }
    if (afterIt != null) {
      afterIt()
    }
  } catch (Throwable ex) {
    listener.gotResult(new Result(ex))
  } finally {
    stepStack.stopStep()
  }
}

class BehaviorCategory {
  // ...

  static void shouldBe(Object self, value) {
    shouldBe(self, value, null)
  }

  //...
}
```

In Listing 8, the `it()` method accepts a spec (a string describing the test) and a closure block representing the body of the test. At the midway point of the method, the closure executes within the `BehaviorCategory` block, which appears at the bottom of the listing. The `BehaviorCategory` augments `Object`, allowing *any* instance in the Java universe to verify its value.

By allowing selective augmentation of `Object`, which resides at the top of the hierarchy, Groovy's open-class mechanism makes it possible to verify results easily for any instance but limits that change to the body of the `use` block.

Scala's implicit casts

Scala uses *implicit casts* to simulate the augmentation of existing classes. Implicit casts don't add methods to classes but allow the language to automatically convert an object to an appropriate type that does have the desired method. For example, I can't add an `isBlank()` method to the `String` class, but I can create an implicit conversion that automatically converts `String`s to a class that does have that method.

As an example, I want to add an `append()` method to `Array`, which lets me add `Person` instances easily to an appropriately typed array, as shown in Listing 9:

Listing 9. Adding a method to `Array` to add people

```
case class Person (firstName: String, lastName: String) {}

class PersonWrapper(a: Array[Person]) {
  def append(other: Person) = {
    a ++ Array(other)
  }
  def +(other: Person) = {
    a ++ Array(other)
  }
}

implicit def listWrapper(a: Array[Person]) = new PersonWrapper(a)
```

In Listing 9, I create a simple `Person` class with a couple of properties. To make `Array[Person]` (in Scala, generics use `[ ]` rather than `< >` as delimiters) `Person` aware, I create a `PersonWrapper` class, which includes the desired `append()` method. At the bottom of the listing, I create the implicit conversion that will automatically convert an `Array[Person]` to a `PersonWrapper` when I call the `append()` method on the array. Listing 10 tests the conversion:

Listing 10. Testing natural extensions to existing classes

```
val p1 = new Person("John", "Doe")
var people = Array[Person]()
people = people.append(p1)
```

In Listing 9, I also add a `+` method to the `PersonWrapper` class. Listing 11 shows how I use this nicely intuitive version of the operator:

Listing 11. Modifying the language to enhance readability

```
people = people + new Person("Fred", "Smith")
for (p <- people)
  printf("%s, %s\n", p.lastName, p.firstName)
```

Scala isn't actually adding a method to the original class, but it provides the appearance of doing so by automatically converting to a suitable type. The same diligence required for metaprogramming in languages like Groovy is required in Scala to avoid creating convoluted webs of interconnected classes using too many implicit casts. But when used correctly, implicit casts help you write very expressive code.

Conclusion

The original Interpreter design pattern from the GoF suggested creating a new language, but their base languages didn't support the graceful extension mechanisms we have at our disposal today. All the next-generation Java languages support extensibility at the language level, using a variety of techniques. In this installment, I demonstrated how operator overloading works in Groovy, Scala, and Clojure, and investigated class extension in Groovy and Scala.

In a future installment, I'll show how a combination of Scala-style pattern matching and generics enable you to replace a couple of traditional design patterns. Essential to that discussion is a concept that also plays a role in functional-style error handling, which is the next installment's topic.

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.
- Complex number: Complex numbers are a mathematical abstraction that play a role in many scientific fields.
- Scala: Scala is a modern, functional language on the JVM.
- Clojure: Clojure is a modern, functional Lisp that runs on the JVM.
- Groovy: Groovy is a modern dynamic language on the JVM with many functional aspects.
- Operator overloading in Groovy: This page shows the full list of operators supported in Groovy and the methods they map to.
- "*Practically Groovy*: Metaprogramming with closures, ExpandoMetaClass, and categories" (Scott Davis, developerWorks, June 2009): Learn more about metaprogramming in Groovy.
- easyb: easyb is an open source behavior-driven development tool developed in Groovy for use with both Groovy and Java projects.
- "Drive development with easyb" (Andrew Glover, developerWorks, November 2008): Find out how easyb helps developers and stakeholders collaborate.
- Grails: Grails is an open source web framework written in Java and Groovy.
- Ruby on Rails: Rails is an open source web framework written in Ruby, which runs on JRuby.
- 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

- 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

- Check out developerWorks blogs and 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.