関数型の考え方: 関数型のデザイン・パターン、第 2 回

同じ問題を異なるパラダイムで解決する

Neal Ford
Software Architect / Meme Wrangler
ThoughtWorks Inc.

2012年 5月 10日

デザイン・パターンは問題を解決するための 1 つの方法に過ぎませんが、主にオブジェクト指向の言語を使用しているとしたら、最終的にはデザイン・パターンの考え方に行き着くことになるでしょう。連載「関数型の考え方」の今回の記事では、Neal Ford が互換性のないインターフェースという共通の問題を、従来のデザイン・パターン、メタプログラミング、そして関数合成の手法を使用して解決する方法を説明します。どの手法にも利点と欠点がありますが、さまざまなソリューションの設計を考えることで、問題を新しい角度から見られるようになります。

このシリーズの他の記事を見る

この連載の前回の記事で、従来の Gang of Four (GoF) のデザイン・パターン (「参考文献」を参照) 手法と、より関数型に近い手法の共通部分について調査を開始しました。この記事ではその調査の続きとして、共通の問題を 3 つの異なるパラダイムで解決する方法を紹介します。そのパラダイムとは、パターン、メタプログラミング、そして関数合成です。

オブジェクト指向を主要なパラダイムとしてサポートしている言語を使用していると、あらゆる問題のソリューションをオブジェクトの観点から考えがちです。けれども、最近のほとんどの言語はマルチパラダイムであり、オブジェクト、メタオブジェクト、関数型、およびその他のパラダイムをサポートしています。問題に応じてさまざまなパラダイムを使い分ける方法を学ぶことが、より優れた開発者になるための進化の道のりの 1 つとなります。

この連載について

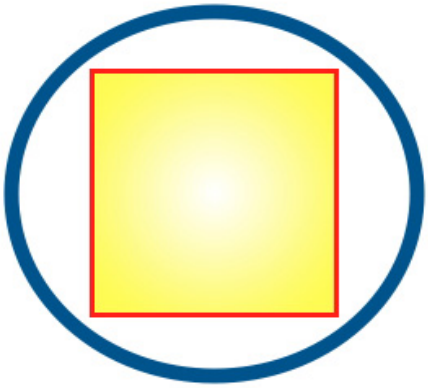
この連載の目的は、読者の皆さんの考え方を関数型の発想へと方向転換し、よくある問題を新たな考え方で検討することによって、日常的なコーディングの改善方法を見つけるお手伝いをすることです。そのために、関数型プログラミングに特徴的な概念、関数型プログラミングを Java 言語で行えるようにするフレームワーク、JVM 上で動作する関数型プログラミング言語、そして今後、言語設計を学習する上での方向性などについて詳しく探ります。この連載の対象読者は、Java および Java の抽象化がどのように機能するかは知っていても、関数型言語を使用した経験がほとんど、あるいはまったくない開発者です。

今回の記事では、Adapter (アダプター) デザイン・パターンが解決する従来の問題に取り組みます。その問題とは、あるインターフェースを別のインターフェースに対応するように変換することです。まずは、Java による従来の手法を検討します。

Java で作成するアダプター

Adapter パターンは、あるクラスのインターフェースを別のインターフェースに変換するというものです。このパターンは、2 つのクラスが概念的には連動するものの、実装の詳細により、連動することができない場合に使用されます。このような例として、四角い杭を丸い穴にはめ込むという問題をモデル化する、いくつかの単純なクラスを作成します。四角い杭と丸い穴の相対サイズによっては、杭が穴の中に収まることもあります (図 1 を参照)。

図 1. 丸い穴の中に収まった四角い杭



正方形が円の中に収まるかどうかを判別するには、図 2 に示す式を使用します。

図 2. 正方形が円の中に収まるかどうかを判別するための式

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

図 2 の式では、正方形の一辺の長さ (w) を 2 で割って半分にしたものを 2 乗し、その結果に 2 を掛けて最後にそのルートを計算しています。この値が円の半径より小さければ、杭は穴の中に収まることになります。

四角い杭と丸い穴の問題は、変換を処理する単純なユーティリティー・クラスを使えば簡単に解けますが、この問題は、さらに複雑な問題の範例となります。例えば、あるタイプのパネルに合

わせることはできるけれども、そのように設計されているわけではないボタンを、そのパネルに合わせようとしている場合などです。四角い杭と丸い穴の問題は、Adapter デザイン・パターンで対処する一般的な問題、つまり相容れない 2 つのインターフェースを適合させるという問題を都合良く単純化したものです。四角い杭を丸い穴に適合させるためには、ほんの少しのクラスとインターフェースによって Adapter パターンを実装する必要があります (リスト 1 を参照)。

リスト 1. Java での四角い杭と丸い穴

```
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;
    }

}
```

Java コードの量を減らすために、`Circularity` という名前のインターフェースを追加して、このインターフェースを実装するクラスが半径を扱えるようにしました。これにより、`RoundPeg` だけに限らず、丸いものという観点から `RoundHole` コードを作成することができます。これは、Adapter パターンでタイプの解決を容易にするために一般に許容されていることです。

四角い杭を丸い穴に適合させるには、`getRadius()` メソッドを公開することによって `Circularity` を `SquarePeg` に追加するアダプターが必要です (リスト 2 を参照)。

リスト 2. 四角い杭のアダプター

```
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);
    }
}
```

このアダプターによって適切なサイズの四角い杭を丸い穴に実際に適合させられることをテストするために、リスト 3 に記載するテストを実装します。

リスト 3. アダプターをテストする

```
@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));
    }
}
```

リスト 3 では、提案される杭の幅のそれぞれについて、`SquarePeg` を作成して `SquarePegAdaptor` でラップし、`hole` の `pegFits()` メソッドが杭の適合性についてインテリジェントな評価を返せるようにしています。

このコードは、Java で実装する冗長ながらも単純なパターンであるため、簡単に理解できる内容となっています。このパラダイムは明らかに、GoF のデザイン・パターン手法です。しかし、このデザイン・パターン手法だけが唯一の方法ではありません。

Groovy で作成する動的アダプター

Groovy (「参考文献」を参照) は、Java ではサポートされていないいくつかのプログラミング・パラダイムをサポートしているので、この記事の残りの例には Groovy を使用します。まず、リスト 2 に記載した「標準的」な Adapter パターンのソリューションを Groovy に移植した場合のソリューションを実装します。リスト 4 を見てください。

リスト 4. Groovy での杭、穴、およびアダプター

```
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)
    }
}
```

リスト 2 の Java バージョンとリスト 4 の Groovy バージョンとの最も顕著な違いは、冗長さです。Groovy は、Java でのコードの繰り返しを動的型付けと便利な機能によって減らすように設計されています。Groovy では、例えば `getRadius()` メソッドに示されているように、自動的にメソッドの最後の行をメソッドの戻り値として機能させることができます。

リスト 5 に、Groovy バージョンのアダプターのテストを記載します。

リスト 5. Groovy で作成された従来のアダプターをテストする

```
@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)
    }
}
```

リスト 5 では、Groovy が持つもう 1 つの便利な機能を利用し、`RoundHole`、`SquarePegAdaptor`、および `SquarePeg` の作成時に、Groovy が自動的に生成した名前と値のコンストラクターを呼び出します。

構文糖が加えられているとは言え、この Groovy バージョンは Java バージョンと同様であり、GoF のデザイン・パターンのパラダイムに従っています。Java の経歴を持つ Groovy 開発者が、かつての経験を新しい構文に移植するのは通常のことです。けれども、Groovy にはこの問題をより簡潔に解決する方法があります。それは、メタプログラミングを使用した手法です。

メタプログラミングを使用した適合方法

Groovy の卓越した機能の 1 つは、その強力なメタプログラミングのサポートです。これから、`ExpandoMetaClass` を使ったメタプログラミングによって、直接アダプターをクラスに組み込みます。

`ExpandoMetaClass`

動的言語に共通の機能は、オープン・クラスです。オープン・クラスでは、メソッドを追加、削除、または変更するために既存のクラス (独自のクラスであるか、`String` や `Object` などのシステム・クラスであるかに関わらず) を再オープンすることができます。オープン・クラスは、DSL (Domain-Specific Language) で多用されているとともに、「流れるようなインターフェース」を作

成するためにも頻繁に使用されています。Groovy にはオープン・クラスのメカニズムとして、カテゴリーと `ExpandoMetaClass` の 2 つがありますが、この記事の例では後者の構文だけを使用します。

`ExpandoMetaClass` を使用することで、クラスまたは個々のオブジェクト・インスタンスに新しいメソッドを追加することができます。この適合の例で、四角い杭が丸い穴の中に収まるかどうかをチェックするには、まず、`SquarePeg` に「半径の要素」を追加する必要があります (リスト 6 を参照)。

リスト 6. `ExpandoMetaClass` を使用して四角い杭に半径を追加する

```
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)
    }
}
```

Groovy では、あらゆるクラスに、そのクラスの `ExpandoMetaClass` を公開する `metaClass` プロパティーが事前に定義されます。リスト 6 ではそのプロパティーを使用して、お馴染みの公式を使った `getRadius()` メソッドを `SquarePeg` クラスに追加しています。`ExpandoMetaClass` を使用するときには、そのタイミングが重要です。このメソッドは、ユニット・テストで呼び出す前に追加されていなければなりません。そのため、この新規メソッドをテスト・クラスの静的イニシャライザー内に追加し、テスト・クラスがロードされるときにこのメソッドが `SquarePeg` に追加されるようにしています。`getRadius()` メソッドが追加された上で、`SquarePeg` を `hole.pegFits()` メソッドに渡せば、後の処理は Groovy の動的型付けに任せることができます。

`ExpandoMetaClass` を使用したほうが、より長いパターンの Groovy バージョンよりも簡潔なコードになることは確かです。このコードは実際に、コードに表れない振る舞いをします。けれどもそれは、欠点の 1 つでもあります。既存のクラスへのメソッドの大々的な追加は、慎重に行わなければなりません。それは、便利さと引き換えに、コードに表れない目に見えない振る舞いによって、デバッグが困難になるという代償を支払うことになるためです。その代償が許容されるのは、DSL での場合や、フレームワークを使用する代わりに既存のインフラストラクチャーを広範にわたって変更する場合などです。

この例は、メタプログラミング・パラダイムによって (既存のクラスを変更して) アダプター問題を解決できることを明らかにしていますが、Groovy の動的特性を使用してこのアダプター問題を解決する方法は、これだけではありません。

動的アダプター

Groovy は、Java では比較的融通の効かない部分も含め、Java とすんなり統合するように最適化されています。例えば、動的にクラスを生成するのは Java では面倒な作業ですが、Groovy では難なくこの作業をこなせます。これはすなわち、アダプター・クラスをオンザフライで生成できることを意味します (リスト 7 を参照)。

リスト 7. 動的アダプターを使用する

```
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 のリテラル・ハッシュ構文は、リスト 7 の `roundPegOf()` メソッドの中で使われているように、角括弧を使用します。インターフェースを実装するクラスを生成する場合、Groovy ではメソッド名をキーとし、値を実装コード・ブロックとしたハッシュを作成できるようになっています。`as` 演算子はこのハッシュを使用してクラスを生成します。そのクラスが実装するインターフェースは、ハッシュのキー名を使用してインスタンス・メソッドを生成します。したがってリスト 7 の `roundPegOf()` メソッドは、`getRadius` をメソッド名として使用し (Groovy のハッシュ・キーがストリングの場合には、二重引用符を使用する必要はありません)、お馴染みの変換コードを実装として使用して、単一エントリーのハッシュを作成します。`as` 演算子はこれを、`RoundThing` インターフェースを実装するクラスに変換します。このクラスは、`functional_adaptor()` テスト内での `SquarePeg` の作成をラップするアダプターとして機能します。

このようにオンザフライでクラスを生成できると、従来のパターン手法に伴う冗長性や一連の手続きが大幅に取り除かれます。また、この方法のほうがメタプログラミング手法よりも明示的です。クラスに新しいメソッドを追加するのではなく、適合を目的としたジャスト・イン・タイムのラッパーを生成しています。この方法はデザイン・パターンのパラダイム (アダプター・クラスの追加) を使用するものの、煩雑さと構文は最小限に抑えられます。

関数型アダプター

ハンマーしか持っていないときには、すべての問題が釘のように見えます。つまり、使用できるパラダイムがオブジェクト指向だけだとすると、別の可能性は目に入ってこないものです。第一級関数を使用しない言語を長い間使用していると、問題を解決するためのパターンを適用し過ぎるという危険があります。多くのパターン (例をいくつか挙げるだけも、Observer パターン、Visitor パターン、Command パターンなどがあります) は基本的に、高階関数のない言語で実装された、移植可能なコードを適用するためのメカニズムです。これらのパターンで使用するオ

ブジェクトに関する操作の大部分は破棄することができ、代わりに単に変換を処理するだけの関数を作成することができます。すると、この手法にはいくつかの利点があることがわかってきます。

関数

第一級関数 (クラスの外部を含め、他の言語構成体を使用できるところであれば、どこででも使用できる関数) があれば、適合を自動的に処理する変換関数を作成することができます。リスト 8 の Groovy コードはその一例です。

リスト 8. 単純な変換関数を使用する

```
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)
    }
}
```

リスト 8 では、`peg` と `hole` を引数として受け取る関数を作成し、その関数を使って杭の適合性をチェックしています。この手法は機能しますが、オブジェクト指向では必要であると考えられていた、穴との適合性に関する判断の部分がなくなっています。場合によっては、クラスを適合させるよりも、穴との適合性に関する判断を外部化する方が理に適うこともあります。これは、関数型パラダイムを意味します。つまり、パラメーターを受け取って結果を返す、純粋な関数です。

合成

関数型の手法についての話題を終える前に、私が気に入っているアダプターを紹介します。これは、デザイン・パターンと関数型の手法を 1 つに統合したアダプターです。第一級関数として実現された軽量の動的ジェネレーターを使用するメリットを説明するために、リスト 9 の例について検討します。

リスト 9. 軽量の動的アダプターによって関数を合成する

```
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)))
    }
}
```

リスト 9 では簡単にわかりやすい方法でアダプターを連結できるように、動的アダプターを返すごく少数の関数を作成しています。関数を合成することで、関数はそれぞれのパラメーターに対する処理内容を制御してカプセル化することができます。関数ごとに、誰がそれをパラメーターとして使用するのかを心配する必要はありません。これは、Groovy の動的ラッパー・クラスを実装として作成する機能を使用した、まさに関数型の手法と言えます。

この軽量の動的アダプター手法を、リスト 10 に示す Java I/O ライブラリーでの無骨なアダプター合成手法と比較してみてください。

リスト 10. 無骨なアダプター合成

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

リスト 10 の例には、アダプターが対処する一般的な問題が示されています。アダプターはこのように、合成された振る舞いを上手く組み合わせることができます。第一級関数のない Java では、コンストラクターを使用して合成するしか方法がありません。他の関数をラップする関数を使用して、それぞれの戻り値を変更するのは、関数型プログラミングでは一般的な方法ですが、Java ではそれほどよく使用されません。なぜなら、Java の場合には、過剰な構文という形でさらにコードが追加されるためです。

まとめ

常に同じパラダイムにとらわれていると、それに代わる手法のメリットが目に入りにくくなってきます。それは、そのパラダイムの世界での考え方にその手法が適合しないからです。パラダイムを組み合わせた最近の言語は、さまざまな設計上の選択肢を提供します。それぞれのパラダイムがどのように機能するか (そして他のパラダイムとどのように相互作用するか) を理解することが、より良いソリューションの選択に役立ちます。今回の記事では、適合性という一般的な問題を説明し、まず、Java と Groovy で従来の Adapter デザイン・パターンを使用してこの問題を解決しました。次に、Groovy のメタプログラミングおよび `ExpandoMetaClass` を使用して問題を解決した後、動的なアダプター・クラスを紹介しました。また、アダプター・クラスに軽量の構文を使用することで、Java では厄介な関数の合成を容易に行えるようになることを説明しました。

次回の記事でも引き続き、デザイン・パターンと関数型プログラミングの共通部分を探ります。

著者について

Neal Ford

Neal Ford は世界的な IT コンサルティング企業である ThoughtWorks のソフトウェア・アーキテクトであり、Meme Wrangler でもあります。また彼は、アプリケーション、教育資料、雑誌記事、コースウェア、ビデオや DVD によるプレゼンテーションなどの設計と開発も行っています。さまざまな技術に関する本の著者、編集者でもあり、最新の著書は『プロダクティブ・プログラマ — プログラマのための生産性向上術』です。彼は大規模なエンタープライズ・アプリケーションの設計や構築を専門にしています。また彼は世界各地で開催される開発者会議での講演者としても国際的に有名です。彼の Web サイトをご覧ください。