関数型の考え方: 連結と合成、第 1 回

ネイティブに連結された抽象化に含まれる意味を探る

Neal Ford
Software Architect / Meme Wrangler
ThoughtWorks Inc.

2011年 10月 07日

毎日、特定の抽象化 (オブジェクト指向など) を扱っていると、その抽象化によって導かれているソリューションが最善の策ではないことに気付きにくいものです。この記事から 2 回にわたり、オブジェクト指向の考え方がコードの再利用に与える影響を詳しく探り、その影響と合成などの関数型の手段がコードの再利用に与える影響とを比較します。

このシリーズの他の記事を見る

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。

「オブジェクト指向プログラミングでは、可変の構成要素をカプセル化することによってコードを理解しやすくする一方、関数型プログラミングでは、可変の構成要素を最小限にすることによってコードを理解しやすくします。」
『Working with Legacy Code』の著者、Michael Feathers 氏による Twitter への投稿

毎日、特定の抽象化を扱っていると、そのやり方が徐々に頭に染み付いてきて、問題の解決法に影響を持つようになってきます。この連載で目指す 1 つの目標は、典型的な問題に対する関数型の考え方を明らかにすることです。今回と次回の記事では、リファクタリングによるコードの再利用と、それに伴う抽象化の影響について取り上げます。

オブジェクト指向の目標の 1 つは、状態をカプセル化して扱いやすくすることです。そのために、オブジェクト指向の抽象化では共通の問題を解決する手段として、状態を使用する傾向があります。これが意味するのは、複数のクラスと相互作用 (つまり、上記の引用で Michael Feathers 氏が称している「可変の構成要素」) を使用するということです。関数型プログラミングでは、構造を「連結」するのではなく、可変の構成要素を「合成」することによって、その数を最小限に

しようと試みます。これは捕らえにくい概念であり、主にオブジェクト指向の言語を扱ってきた開発者にとって、簡単に理解できることではありません。

構造によるコードの再利用

命令型で (特に) オブジェクト指向のプログラミング・スタイルでは、構造とメッセージングを基本的な構成単位として使用します。オブジェクト指向のコードを再利用するには、ターゲットのコードを別のクラスに抽出し、そのコードを利用するために継承を用います。

意図せぬコードの重複

コードの再利用とそこに含まれる意味を説明するために、ここでは以前の記事でコードの構造とプログラミング・スタイルを説明するために使った数値分類子を再び例として引用します。数値分類子は、整数が過剰数、完全数、または不足数のいずれであるかを判断します。数値の約数の和が、その数値を 2 倍した値より大きい場合、数値は過剰数であり、数値を 2 倍した値と等しければ、完全数です。それ以外の場合 (約数の和が数値の 2 倍よりも小さい場合) は不足数ということになります。

あるいは、正の整数の約数を使用して、その整数が素数 (1 より大きく、約数が 1 とその整数自身だけである整数) であるかどうかを判断するコードを作成することもできます。この 2 つの問題はどちらも数値の約数を利用するため、リファクタリングの対象としても、コード再利用のプログラミング・スタイルを説明する上でも絶好の候補です。

リスト 1 に、命令型で作成した数値分類子を記載します。

リスト 1. 命令型の数値分類子

```
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;
    }

}
```

連載最初の記事で、このコードの基になったコードについては説明したので、ここでその説明を繰り返すことはしません。今回はコードの再利用を説明することが目的なので、素数をテストするためのリスト 2 のコードを作成します。

リスト 2. 命令型で作成した素数のテスト

```
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;
    }
}
```

リスト 2 には、いくつか注目すべき項目があります。まず、`isPrime()` メソッドに含まれる多少奇異に見える初期化コードに注目してください。これは、インスタンス・イニシャライザーの一例です (インスタンス・イニシャライザー (関数型プログラミングに付随する Java 手法) について

の詳細は、「Evolutionary architecture and emergent design: Leveraging reusable code, Part 2」を参照してください)。

リスト 2 では、`isFactor()` メソッドと `factors()` メソッドにも注目してください。この 2 つは、`ClassifierAlpha` クラス (リスト 1 を参照) に含まれている、それぞれに対応するメソッドとまったく変わりありません。これは、2 つのソリューションをそれぞれ単独に実装したところ、ごく自然な結果として実質的に同じ機能になったものです。

重複をなくすためのリファクタリング

この種の重複を解決する方法は、コードを 1 つの `Factors` クラスにリファクタリングすることです (リスト 3 を参照)。

リスト 3. 一般的なリファクタリングを実行後の約数へ分解するコード

```
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;
    }
}
```

リスト 3 のコードは、「スーパークラスの抽出 (Extract Superclass)」というリファクタリングを実行した結果です。抽出した 2 つのメソッドは両方とも `number` メンバー変数を使用していることから、スーパークラスに移動されています。このリファクタリングの実行中に、IDE からアクセスの処理方法 (アクセサー・ペア、protected スコープなど) を選択するように求められたので、私は protected スコープを選択しました。その結果、`number` がクラスに追加され、その値を設定するためのコンストラクターが作成されています。

重複するコードを切り離して削除した結果、数値分類子と素数テスターは両方とも遥かにシンプルになっています。リスト 4 に、リファクタリング後の数値分類子を示します。

リスト 4. リファクタリングによってシンプルになった数値分類子

```
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;
    }

}
```

リファクタリング後の素数テスターは、リスト 5 に示すとおりです。

リスト 5. リファクタリングによってシンプルになった素数テスター

```
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);
    }
}
```

リファクタリングの際に、`number` メンバーに対するアクセス方法としてどれを選択するかに関わらず、この問題について検討するときにはクラスのネットワークに対処しなければなりません。大抵は、クラスのネットワークに対処すると、問題をそれぞれの部分に切り分けられて有効ですが、それによって、親クラスを変更する段階になって影響が出てきます。

これは「連結」によるコード再利用の一例で、`number` フィールドとスーパークラスの `getFactors()` メソッドが共有する状態によって 2 つの要素 (この例の場合はクラス) を結び付けています。つまり、この方法は、言語に組み込まれている連結ルールを使用することによって機能します。オブジェクト指向では、連結された相互作用のスタイル (例えば、継承によってメンバー変数にアクセスする方法) を定義しているため、連結の方法については事前に定義されたルールがあります。このことは、振る舞いについて一貫した方法で判断できるため、望ましいことです。誤解を避けるために言っておきますが、私は継承を使用するのが悪い考えであると言っているのではありません。ただ、オブジェクト指向の言語では継承があまりにも使用され過ぎていると言っているのです。それよりも優れた特性を持つ、別の抽象化があります。

合成によるコードの再利用

リスト 6 に記載するのは、この連載の 2 本目の記事で紹介した Java による関数型バージョンの数値分類子です。

リスト 6. より関数型に近い数値分類子

```
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;
    }
}
```

素数テスターの関数型バージョンも用意しました (純粋な関数を使用し、共有する状態はありません)。リスト 7 に、このバージョンでの `isPrime()` メソッドを記載します。コードの残りのメソッドについては、リスト 6 に記載した同じ名前のメソッドとまったく変わりありません。

リスト 7. 関数型バージョンの素数テスター

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

命令型バージョンで行ったように、重複するコードを専用の `Factors` クラスに抽出し、`factors` メソッドの名前を `of` に変更して読みやすくします (リスト 8 を参照)。

リスト 8. 関数型にリファクタリングした`Factors` クラス

```
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;
    }
}
```

関数型バージョンでは、すべての状態はパラメーターとして渡されます。したがって、この抽出には、共有の状態は伴っていません。このクラスを抽出した後は、このクラスを使用するように関数型の分類子と素数テスターの両方をリファクタリングすることができます。リスト 9 に、リファクタリング後の数値分類子を記載します。

リスト 9. リファクタリング後の数値分類子

```
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;
    }
}
```

リスト 10 は、リファクタリング後の素数テスターです。

リスト 10. リファクタリング後の素数テスター

```
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);
    }
}
```

2 番目のバージョンをより関数型に近づけるために、特殊なライブラリーや言語を一切使用していないことに注目してください。関数型へのリファクタリングは、コードを再利用するために、連結を使うのではなく合成を使用して行いました。リスト 9 とリスト 10 ではどちらも `Factors` クラスを使用していますが、このクラスは個々のメソッドの内部に完全に含まれています。

連結と合成の違いは微妙なものですが、その違いは重要です。この記事で説明した単純な例では、コード構造のスケルトンが明らかに見て取れますが、大規模なコード・ベースのリファクタリングとなると、連結が至るところに現れてきます。オブジェクト指向の言語では、連結が再利用メカニズムの 1 つとなっているからです。至るところで連結された構造を理解する難しさが、オブジェクト指向言語での再利用の妨げとなっていることから、効果的な再利用は、オブジェクト・リレーショナル・マッピングおよびウィジェット・ライブラリーなどの明確に定義された技術ドメインに限られています。比較的曖昧に構造化された Java コード (例えば、ビジネス・アプリケーションで作成するコードなど) を作成する場合にも、効果的な再利用を実現するのは困難です。

リファクタリング中に IDE が提示するオプションを丁重に断って、代わりに合成を使用していれば、命令型バージョンを改善できていたはずです。

まとめ

関数型プログラマーのように考えるということは、コード作成のあらゆる側面に関して違った考え方を持つということを意味します。コードを再利用することは、開発での明らかな目標となりますが、命令型の抽象化でこの問題を解決する方法は、関数型プログラマーの問題解決方法とは異なる傾向にあります。今回の記事では、コードの再利用に対する 2 つのプログラミング・スタイルとして、継承による連結とパラメーターによる合成を対比しました。次回の記事でも引き続き、この重要な違いを詳しく探ります。

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。