SRA GROUP
GSLetterNeo Vol.115
2018年2月

リファクタリングのススメ

土屋 正人

◆はじめに

C++が登場してから 30 年、Java が登場してから 20 年が過ぎました。現在でもプログラミング言語のメインストリームのひとつだと思いますが、過去に作られたソースコードの中には、業務の変更などに伴う要求仕様変更により改訂が繰り返し行われ、年月を重ねた結果、メソッドやクラスが巨大化して保守が難しくなっている、という問題を抱えているものもあります。

◆ソースコード保守の問題

機能追加や仕様変更の際に問題となるのは、影響範囲の特定です。ソースコードを読んでいて、

- 何をやっているのかわからない
- 何のために必要なのかわからない

ということになると、下手に手を付けることが出来なくなります。仕様――すなわち存在理由を明確にしてから着手すべきですが、

- 仕様書も設計書も存在せず、あるのはソースコードだけ
- 仕様書も設計書も、開発当初のものはあるが更新されていない

ということが往々にしてあります。そのため、

- 新規機能は、似ている既存機能をコピー＆ペーストしたものをベースにして作る
- 機能変更は、現状はそのままにして分岐を入れて分岐先に作る

ということになりがちです。メソッドはますます巨大化し、理解困難なコードが増大し、悪循環を助長します。

- ひとつのメソッドを読むために画面をスクロールしなければならない
- インデントが深いため改行されて表示される（または横スクロールしなければならない）

読むのが苦痛になります。美しくありません。

- 同じロジックが他のメソッドやクラスにも出てくる
- 同じデータ構造が他のメソッドやクラスにも出てくる

変更の影響範囲が広がります。

- 引数の範囲チェックを呼び出す側と呼び出される側双方で行っている
- 特定の条件の時だけしか使わない引数を渡している／受け取っている

モジュール間の依存関係が強まります。
さらに細かく見ていくと、

- ひとつのメソッド内で、同じ変数が異なる目的で何度も使われている
- 変数に基本データ型を使っているため、必ず範囲チェックを行っている

等々、挙げ出したら切りがありません。
何かおかしい…と感じることがあれば、リファクタリングをする機会だと思います。

◆リファクタリングのはじめの一歩

プログラムの保守性を向上させる場合、要素と要素間の関係を見直す必要がありますが、既に動いていて使われているプログラムの場合、コストやデグレード等、外的影響の関係から、なかなか作り直しという結論にはならないものです。

その場合、有益なのはメソッドレベルから始めるリファクタリングだと思います。

リファクタリングとは、**外から見えるふるまいを変えずに、ソフトウェアを分かりやすくし、安いコストで変更できるようにするためにソフトウェアの内部構造に加えられる変更あるいは、これを行なうこと**です。

前節で挙げたコードの問題点を改善していきます。

- **変数名は省略しない**
- **同じ変数を使い回すのではなく、目的・役割が違うなら変数も分ける**

これだけでも、分かりやすく――つまり、やっていることが見えてくるようになります。更に、

- **手順になっているものは、段落ごとにメソッドにする**
- **コレクションを扱うループをメソッドにする**

元のメソッドは小さくなり、読みやすくなります。更に、

- **範囲チェックが必要な値を小さなクラスにする（バリューオブジェクト）**
- **コレクションを扱うメソッドをクラスにする（ファーストクラスコレクション）**

元のクラスも小さくなり、読みやすくなります。特定の関心事に特化した小さなクラスにすることで、

- **値の範囲の制限チェック**
- **特定の条件における処理**

を、そのクラスのひとつあるいは複数のメソッドにすることが出来ます。範囲チェックや条件チェックが局所化されることで重複ロジックがなくなり、範囲や条件に関する仕様変更に対して、影響範囲を特定しやすくなります。

コメントなしでもソースコードを読むだけで、何のために、何をやっているか、わかるようになり、ソースコードが説明書になって行きます。

リファクタリングを更に進めるには、ThoughtWorks のコンサルタントたちのエッセイ集『ThoughtWorks アンソロジ』に Jeff Bay が書いている「オブジェクト指向エクササイズ」が参考になります。ソフトウェア設計を改善するために、以下の 9 つのルールを適用してみることを勧めています。

1. **1 つのメソッドにつきインデントは 1 段落にすること**
2. **else 句を使用しないこと**
3. **すべてのプリミティブ型と文字列型をラップすること**
4. **1 行につきドットは 1 つまでにすること**
5. **名前を省略しないこと**
6. **すべてのエンティティを小さくすること**
7. **1 つのクラスにつきインスタンス変数は 2 つまでにすること**
8. **ファーストクラスコレクションを使用すること**
9. **Getter,Setter,プロパティを使用しないこと**

◆設計改善への一歩

前節で記した「特定の関心事に特化した小さなクラス」にすることが出来ると、設計が改善されます。

オブジェクト指向設計原則の代表的なものとして、単一責務原則（SRP:Single Responsibility Principle）があります。これは、

- **変更する理由が同じものは集める**
- **変更する理由が違うものは分ける**

というもので、「**クラスを変更する理由は複数存在してはいけない**」ことになり、仕様変更による影響範囲が局所化されて特定しやすくなります。

最初の開発時の設計から意識するに越したことはありませんが、リファクタリングによっても到達することが出来ます。

また、同じくオブジェクト指向設計原則として、オープン・クローズド原則（OCP:Open-Closed Principle）があります。これは、

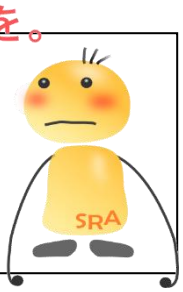
- **モジュールは拡張に対して開いている**（Open）
- **モジュールは修正に対して閉じている**（Closed）

を共に満たすというもので、これも SRP 同様、リファクタリングによっても到達することが出来ます。

設計原則なるものは他にも多数ありますが、上記の 2 つを満たすことが出来れば、冒頭に挙げた保守の難しさは、かなり改善されると思います。

とはいえ、設計原則は指針であり手段。それらが目的ではないことは、言うまでもありません。

GSLetterNeo Vol.115
2018 年 2 月 20 日発行
発行者●株式会社 SRA 先端技術研究所
編集者●土屋正人



株式会社SRA
〒171-8513 東京都豊島区南池袋２－３２－８