リファクタリングによる ソフトウェアの品質向上に関する考察

石川 洋

ソフトウェア開発において，オブジェクト指向を導入することは，最終成果物の品質を向上させる一つの手段と考えられている．しかしながら，それを利用してもはじめから高品質なソフトウェアを設計，実装するのは困難である．オブジェクト指向に基づくプログラムの品質を向上させる方法の一つにリファクタリングがある．リファクタリングは既存のソフトウェアの動作を変更せずに，規則にしたがってプログラムを書き換える作業である．本稿では，Java言語で記述されたオブジェクト指向によるプログラムにおけるリファクタリング手法を2つ紹介する．その作業により，プログラムの品質が向上したことを客観的に判断するための考察を行う．
【キーワード】Java，リファクタリング，オブジェクト指向プログラミング，品質向上

1 はじめに

ソフトウェア開発において，オブジェクト指向に基づく設計やプログラミングが活用されるようになってきている．オブジェクト指向の特徴である，カプセル化，継承，クラス間の関係，メソッドの多相性などは，ソフトウェアの再利用性や保守性，すなわち，ソフトウェアの品質を向上させると考えられている．

しかしながら，はじめから高品質なソフトウェアを構築するのは困難であり，初期段階として，要求されている仕様を満たしているソフトウェアを構築し，それを土台として設計やプログラムの見直しを行い，より品質の高いソフトウェアに仕上げていくことが少なくない．

より品質の高いソフトウェアに仕上げていく方法の一つににリファクタリングがある [1]．リファクタリングは，ソフトウェアの動作を変えずにプログラムを洗練していく作業である．最近は，オブジェクト指向プログラミング，特に Java 言語 [2] で記述されたプログラムのためのリファクタリングが数多く紹介されている (例えば [1][5])．本稿では，リファクタリングのうち，次の 2 つについて解説し，プログラムの品質が向上したと判断するための指標をどのように設定すればよいかについて考察する．取り上げるリファクタリング手法の一つ目は，ある処理の事前条件や事後条件のチェックを行うアサーションの導入である．これは，プログラムを記述する際のささやかなではあるが重要な工夫である．もう一つは，あるクラスのイ

ンスタンスごとに振る舞いを変えたい場合に，そのクラスのサブクラスを利用する手法であり，これを考慮すると，プログラムは少なからず書き換えられる．これらの例で使用するプログラムは Java 言語で記述されている．

本稿は以下のような構成である．まず，リファクタリングの概要を述べ，次に，2 つのリファクタリング手法の具体例を述べる．その後，それらの手法を用いたプログラムの変更を行った場合に，プログラムの品質が向上したと判断するための指標を提案する．最後に，本稿のまとめを記述する．

2　リファクタリング

リファクタリングとは，ソフトウェアの外部的振る舞い（利用者が見える部分）を保ったままで，内部の構造（プログラム）を改善する作業のことをいう．したがって，予め仕様通りに動作するソフトウェアが存在することを前提としている．リファクタリングはすべてのプログラムに対して適用可能であるが，Fowler らは，オブジェクト指向に基づくプログラムを対象とした，統制された方法論を提案した [1]．リファクタリングの手法は，その目的や手順がカタログ化されており，数多くが提案，紹介されている．内容については割愛する (ただし,「アサーションの導入」と「クラスによるタイプコードの置き換え」については後述する) が，それらのいくつかを例に挙げると,「シンボリック定数によるマジックナンバーの置き換え」,「アサーションの導入」,「ヌルオブジェクトの導入」,「メソッドの抽出」,「クラスの抽出」,「クラスによるタイプコードの置き換え」,「サブクラスによるタイプコードの置き換え」,「State/Strategy によるタイプコードの置き換え」,「例外によるエラーコードの置き換え」などがある [5].

リファクタリングを行う場合，一度に一つずつの変更を考え，適用するといった慎重さが求められる.「一度に一つずつ」とは，先に挙げた手法を複数同時に使用しないということと，あるリファクタリング手法で定められている作業手順を複数同時に行わないということを意味している．このように，リファクタリングのエッセンスは，動作しているプログラムを，エラーを発生させず，着実に高品質なプログラムに書き換えることにある．次に，2 つのリファクタリング手法を例に挙げ，その具体的な手順を紹介する．

2.1　アサーションの導入

この手法は，ある処理の前後 (事前，事後) で成立するはずの条件を調べ，Java 言語が提供している `assert` を用いてプログラム中に記述するものである．`assert` の記述方法は，`assert expr;` または `assert expr : option;` の 2 通りがある．前者は条件式 `expr` が真

であれば何も起こらず，偽であればエラーが発生する．後者は，条件式 expr が真であれば何も起こらず，偽であればエラーが発生し，option も評価される．

アサーションの導入は次の手順で行われる．

- ソースコード中で成立することが前提となっている条件を見つけ出す．
- その条件を含むアサーションを記述する．この際，アサーションの中に副作用がある式 (expr) を書いてはならない．必要ならば，条件が成立しなかった場合のオプション (option) を記述してもよい．
- そこの書かれているコメントが無意味になってしまったら，コメントを削除する．
- コンパイルしてテストする．

ここでは，配列に格納された値を昇順に並べ替える (ソート) の例を挙げる [5]．実際に配列の要素を入れ換える処理の前後にアサーションを記述している．一つ目の assert を用いて，配列の指定された要素が，配列の指定された範囲内で最小値であることを確認している．二つ目は，配列の指定された範囲内で，要素が昇順になっているかどうかを確認している．

```
public void sort() {
  for(int x=0; x<_data.length-1; x++) {
    int m=x;
    for(int y=x; y<_data.length; y++) {
      if(_data[m]>_data[y]) { m=y; }
    }
    assert isMin(m, x, _data.length-1);                    // assertion
    int v=_data[m];  _data[m]=_data[x];  _data[x]=v; // 要素の入れ換え
    assert isSorted(0, x+1);                               // assertion
  }

  private boolean isMin(int pos, int start, int end) {
    配列の指定された要素が，配列の指定された範囲内で最小値であるかを調べる．
    最小値であれば true を，そうでなければ false を返す．
  }

  private boolean isSorted(int start, int end) {
    配列の指定された範囲内で，要素が昇順にソートされているかを調べる．
    ソートされていたら true を，そうでなければ false を返す．
  }
}
```

図 1 ：アサーションを導入したソースコード例

アサーションの導入により，アサーションを記述した箇所で成立する条件が明確になり，ソースコードの可読性が向上し，バグが早期に発見できるといった利点が期待できる．

2.2 サブクラスによるタイプコードの置き換え

あるクラスのインスタンスを生成するとき，インスタンスの特性を判断するために，タイプコードと呼ばれる数値データを持たせることがある．例えば，図形のクラス Shape のインスタンスとして線分，矩形，楕円があり，それぞれを区別するために，0，1，2 という数値をデータとして割り当てるという考え方である．この場合，クラス Shape に定義される様々なメソッドは，実質的な処理を行う前に，インスタンスに割り当てられたタイプコードを確認する手間が必要となる．そのような場合，クラス Shape で扱う図形が増えた場合，その図形に対応する処理を追加することになり，クラス Shape 内のプログラム記述量が増加し，プログラムの可読性が低下しかねない．

そこで，オブジェクト指向の特徴である，継承とメソッドの多相性を利用し，タイプコードをサブクラスで置き換えることが考えられた．これにより，特定の図形に対する処理は，タイプコードを判断して決定するのではなく，特定の図形を表現するサブクラスで定義しているメソッドの多相性を利用できる．

サブクラスによるタイプコードの置き換えの手順は次の通りである．

- タイプコードを間接的にアクセスさせるメソッドを用意する．
- タイプコードを元にインスタンスを生成している場合は，そのコンストラクタを間接的に呼ぶメソッドを用意する．
- タイプコードのそれぞれの値ごとにサブクラスを定義する．

ここでは先に例に挙げた図形を扱うクラス Shape のリファクタリング前のプログラムとリファクタリング後のプログラムを紹介する [5]．

```
public class Shape {
  // タイプコードの定義
  public static final int TYPECODE_LINE=0;      // 直線
  public static final int TYPECODE_RECTANGLE=0; // 矩形
  public static final int TYPECODE_OVAL=0;      // 楕円

  // フィールドの定義
  private final int _typecode;
  private final int _startx, _starty;
  private final int _endx, _endy;

  // コンストラクタ
  public Shape(int typecode, int startx, int starty,
                             int endx, int endy) {
    _typecode = typecode;
    _startx   = startx;    _starty   = starty;
    _endx     = endx;      _endy     = endy;
```

```
  }

  // タイプコードを取得するメソッド
  public int getTypecode() { return _typecode; }

  // タイプコードによる図形の名称を獲得するメソッド
  public String getName() {
    switch(_typecode) {
    case TYPECODE_LINE:      return "LINE";
    case TYPECODE_RECTANGLE: return "RECTANGLE";
    case TYPECODE_OVAL:      return "OVAL";
    default:                 return null;
  }

  // タイプコードによる図形描画メソッドの呼び出し
  public void draw() {
  switch(_typecode) {
  case TYPECODE_LINE:      drawLine(); break;
  case TYPECODE_RECTANGLE: drawRectangle(); break;
  case TYPECODE_OVAL:      drawOval(); break;
  default:                 ;
  }

  // 図形ごとの描画メソッド
  private void drawLine() { 直線を描画する処理 }
  private void drawRectangle() { 矩形を描画する処理 }
  private void drawOval() { 楕円を描画する処理  }
}
```

図 2 ：クラス Shape のリファクタリング前のプログラム

図 2 で与えたプログラムを先に述べた手順に基づいてリファクタリングを行った結果が次のプログラム (図 3) である．変更されなかった部分については省略している．

サブクラスによるタイプコードの置き換えにより，多相的なメソッドがサブクラスごとに定義できるようになる．プログラムの記述量は確実に増加するが，可読性，拡張性，保守性の向上が期待でき，結果としてソフトウェアの品質が向上する．

```
public abstract class Shape {
  // タイプコードの定義はリファクタリング前と同じ

  // フィールドの定義で _typecode は削除，他はリファクタリング前と同じ

  // サブクラスを生成するコンストラクタを間接的に呼び出すメソッド
  public static Shape createShpae(int typecode,
                                  int startx, int starty,
                                  int endx, int endy) {
```

```
    switch(typecode) {
    case TYPECODE_LINE:       // 直線のためのサブクラス
      return new ShapeLine(statx, starty, endx, endy);
    case TYPECODE_RECTANGLE: // 矩形のためのサブクラス
      return new ShapeRectangle(statx, starty, endx, endy);
    case TYPECODE_OVAL:       // 楕円のためのサブクラス
      return new ShapeOval(statx, starty, endx, endy);
    default:                  // 例外処理
      throw new IllegalArgumentException("typcode="+typecode);
    }
  }

  // クラス Shape のコンストラクタはリファクタリング前と同じ

  // 以下は抽象メソッドとして定義し，実際の処理はサブクラスで記述
  public abstract int getTypecode();
  public abstract String getName();
  public abstract void draw();
}

// 直線のためのサブクラス ShapeLine
public class ShapleLine extends Shape {
  // コンストラクタ
  protected ShapeLine(int startx, int starty, int endx, int endy) {
    super(startx, starty, endx, endy);
  }

  // 抽象メソッドの実装 (オーバーライド)
  @Override public int getTypecode() { return Shape.TYPECODE_LINE; }
  @Override public String getName() { return "LINE"; }
  @Override public String draw() { drawLine(); }

  // メソッド drawLine の定義は，リファクタリング前のコードを
  // サブクラス内に移動
}

// サブクラス ShapeRectangle, ShapeOval も同様に定義
```

図 3 ：クラス Shape のリファクタリング後のプログラム

3 品質向上を評価する指標

リファクタリングを行うことで，ソフトウェアの可読性，拡張性，保守性が向上すると考えられているが，これらの評価にはやや主観的な側面がある．ソフトウェアの品質の変化を示す指標や客観的な評価方法があると，リファクタリング作業の必要性や重要性が強調できる．前節では 2 つのリファクタリング手法を説明したので，これらの作業によってソフトウェアの品質がどの程度，どのように向上したのかを判断するための指標について考察する．

3.1 アサーションの導入

assert の使用は，言語処理系をより効果的に利用していると言える．アサーションの導入は，リファクタリング手法として紹介しているが，プログラムを記述する初期段階で使用するのがより効果的であろう．アサーションが導入されているプログラムならば，検査目的の処理が正しい事前，事後条件の元で動作していることが一目で確認できる．したがって，この手法の場合は，assert が使用されているので，ある処理の事前，事後条件に関して十分留意してプログラムを記述しており，品質を向上させるための努力は行っているという，やや主観的な評価を与えることができる．

3.2 サブクラスによるタイプコードの置き換え

この手法では，リファクタリング前の，クラス内で使用しているタイプコードの数と，リファクタリング後に定義されたサブクラスの数とが一致する．サブクラスの数が多ければ，プログラム記述量も増加するが，クラスのスリム化，メソッドの多相性を利用することで，プログラムの可読性，拡張性が向上する．タイプコードの数が多ければ，品質もより向上したとみなしてよい．そこで，この手法による品質向上を評価する指標の候補として，タイプコードの数が考えられる．この値を指標とすることで，客観的な判断が可能となる．

4 まとめ

本稿では，オブジェクト指向に基づくプログラムに対するリファクタリングについて説明し，2 つの手法については具体例を示した．また，それぞれの手法によってプログラムが書き換えられたときに，その品質の変化を表し，評価するための指標について考察した．アサーションの導入では，プログラムの構造を書き換えるものではないため，品質向上に留意したプログラムを作成していることが主張される．そのため，客観的な評価のための指標を設定するのは困難だと考えられる．サブクラスによるタイプコードの置き換えでは，リファクタ

リング前のクラス内で使用しているタイプコードの数を品質向上を評価する指標の候補として提案した．具体的な数値が指標として利用できるので客観的な判断材料としては適切であると考えられる．

リファクタリング作業を対話的に行うツールがフリーソフトで提供されている (例えばEclipse[3]，JUnit[4] など)．ツールが提供されている．これらのツールと，本稿で議論したプログラムの品質向上を評価する指標を組み合わせて使用することで，より効果的で，より品質向上を重視したリファクタリングが可能になると考えられる．

参考文献

[1] M. Fowler, *Refactoring: Improving The Design of Existing Code*, Addison-Wesley, 1999.

[2] B. Joy, G. Steele, J. Gosling, G. Bracha, *The Java Language Specification, third Edition*, Addison-Wesley, 2005.

[3] Eclipse のホームページ，*http://www.eclipse.org/*.

[4] 福島竜，*JUnit* と単体テスト技法, ソフト・リサーチ・センター, 2006.

[5] 結城浩，*Java* 言語で学ぶリファクタリング入門, ソフトバンククリエイティブ, 2004.

リファクタリングによるソフトウェアの品質向上に関する考察

A Discussion about Quality Improvements of Software by Refactoring

Hiroshi ISHIKAWA

ABSTRACT: Designing and programming based on object-oriented methodology have potential for improving quality of software, however, it is difficult to obtain the software with high quality from the start.
Refactoring is one of useful methods to improve the quality of software based on object-oriented methodology. Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior.
In this paper, two activities of refactoring for source codes written in Java, one of object-oriented programming languages. Ideas how to evaluate the quality of Source codes after refactoring on an objective criterion are discussed.
Key words: Java, Refactoring, Object-Oriented Programming, Quality Improvements