第 1 回 リファクタリングゼミ

ゼミ担当者　：　石田 裕幸，上田 祐一郎，菅原 麻衣子
指導院生　：　朝山 絵美，柴田 優
開催日　：　2006 年 5 月 25 日

1　リファクタリングとは

リファクタリングとは，ソフトウェアの外部的振る舞いを保ちつつ，理解や修正が簡単になるように，内部構造を改善することである．リファクタリングは，デザインパターンなどと並んで，現代における開発手法として注目されてる．

本ゼミでは，リファクタリングの必要性とともに，プログラム改善の流れ，リファクタリングの種類，および eclipse のリファクタリングツールについて説明する．

2　リファクタリングの必要性と効果

2.1　リファクタリングの必要性

一度作成されたプログラムは，修正，機能追加などによって変更が繰り返される．変更の繰り返しにより，プログラムは次第に読みづらく，また理解しにくいものになってしまう．それに伴って，次の変更への対応が遅れることや，潜在的なバグを発生させることにつながる．上記のような問題を解決するには，プログラマの意図を明確に読み取れるシンプルなコードを保つ必要がある．その際に必要となるのがリファクタリングである．

2.2　リファクタリングの効果

リファクタリングすることにより得られる効果について，以下に示す．

- 変更に対する柔軟性の向上
  重複部分，複雑な部分を排除することにより，新たな変更に対しても柔軟に対処できるようになる．

- プログラム構成の向上
  コードがシンプルかつ読み易くなるため，理解し易くなる．

- バグの顕在化
  コードがシンプルになるのに伴い，潜在的なバグも発見し易くなる．

- システム開発の高速化
  変更に対する柔軟性が向上するため，修正や機能追加が正確かつ，高速に行うことができるようになる．

3　開発プロジェクトの中でリファクタリングを行う時期

ソフトウェア開発においては，リファクタリングを行うべき適した時期が存在する．主な時期を以下に 3 つ示す．

- 3 度目の認識時
  コーディングを行う際,「前にも同じロジックを書いた」と感じることがあったとする．更にコーディングを進めて「また同じロジックを書いた」と感じたとする．つまり 3 度同じロジックを書いたと認識した時にリファクタリングを行う．

- 機能を追加する前
  機能追加をする前にリファクタリングを行うことによってコードを整理し易くなり，それに伴って機能追加し易くなる．なお，機能追加とリファクタリングは決して同時にはやらない．

- バグフィックスの作業後
  バグが発生した際，プログラムの理解のためにリファクタリングを行う．

4　リファクタリングを用いるプログラミング

4.1　テスト

リファクタリングは，必ずテストと組み合わせて行わなければならない．リファクタリングは正常に作動しているプログラムを書き直すこともあるため，プログラムがリファクタリング後も正常に作動することを保証しなければならないためである．テストの自動化を行うフレームワークに JUnit がある．JUnit は Eclipse に標準でバンドルされている．

4.2　リファクタリングを用いたプログラミングの流れ

リファクタリングする際に行うテストは，クラスを全て実装した後に行うものではない．1 つのメソッドの実

装ごとにテストし，プログラムが正常であることを検証する.「小さく作って小さく動かす」が基本である.

- クラスが完成するまでの流れ
  クラス全体の実装中にリファクタリングを行う場合のプログラミングの流れは，Fig.1 のようになる．リファクタリング後も，テストを行うことによりプログラムが正常に動作することを確認しなければならない.

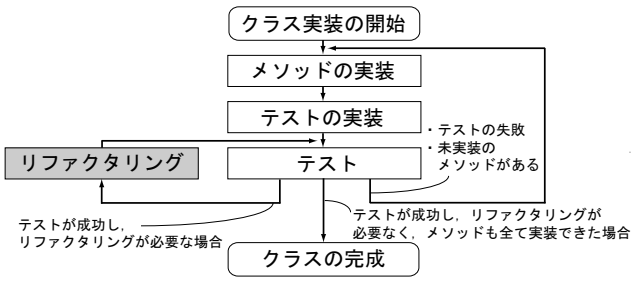

Fig. 1　クラス完成までのプログラミングの流れ

- 機能の追加の流れ
  実装済みのクラスに，機能追加が発生したときのプログラミングの流れは Fig.2 のようになる．この場合も，リファクタリング後に必ずテストを行い，正常に作動することを確認しなければならない.

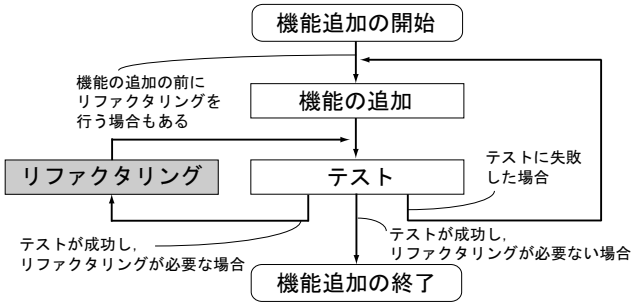

Fig. 2　機能追加のプログラミングの流れ

4.3　リファクタリングが必要な場合

3 章に示した時期以外に，リファクタリングが必要とされる時期がある．理解や修正が簡単ではないコードからは,「不吉な匂い」がすると表現されており，このときリファクタリングが推奨される．この「不吉な匂い」は以下に示すように分類整理されている.

- 重複したコード
  同じような処理が 2 箇所以上記述されている.
- 長すぎるメソッド
  長くて理解しづらいメソッドがある.
- 巨大なクラス
  余りに多くのことを行い過ぎているクラスがある.
- 多過ぎる引数
  引数が多過ぎる，また引数に同じ型が続いている.
- 変更の発散
  1 つの変更要求に対し，1 クラス内の複数のメソッドに修正が必要になる.
- 変更の分散
  1 つの変更要求に対し，複数のクラスに修正が必要になる.
- 属性，操作の横恋慕
  他のクラスの特定の属性や操作を過剰に利用している.
- データの群れ
  同じ組み合わせのデータが様々な場所に現れる.
- 基本データ型への執着
  オブジェクトにできる場合でも基本型を利用してプログラミングされている.
- スイッチ文
  スイッチ文を利用している.
- パラレル継承
  新しいサブクラスを定義するたびに，別の継承木にもサブクラスを定義する必要がある.
- 怠け者クラス
  十分な仕事をせず，理解や保守のためのコストにも見合わないクラスがある.
- 疑わしき一般化
  いつか必要になると思われて作られたが，未だに必要とされていないクラスやメソッドがある.
- 一時的属性
  特定の状況にならなければ設定されないインスタンス変数がある.
- メッセージの連鎖
  メッセージが次から次へとオブジェクトに転送される過剰なメッセージを連鎖している.
- 仲介人
  メッセージを受け取って次のオブジェクトにメッセージを渡すだけのオブジェクトがある.
- 不適切な関係
  密に結合しているオブジェクトがある.
- クラスのインターフェース不一致
  シグニチャは異なるが処理が同じメソッドがある.
- 未熟なクラスライブラリ
  必要なメソッドが不足しているクラスライブラリがある.
- データクラス
  属性とアクセサしか持たないクラスがある.

- 接続拒否
  親クラスの属性と操作を有効に活用していないサブクラスがある．
- コメント
  コメントは丁寧だがコードは読みにくいものとなっている．

これらの「不吉な匂い」は次章で挙げるようなリファクタリング方法を組み合わせることによって解決する．「不吉な匂い」の詳細や，「不吉な匂い」の詳しい解決方法に関しては，参考文献や他の書籍を参照されたい．

5　リファクタリングの方法

代表的なリファクタリングの方法について以下に示す．

5.1　メソッドの抽出

メソッドの抽出とは，冗長で一まとめにできるコードの断片をメソッドにし，その意図に応じた名前を付けることである．以下に示す場合において行う．

- コードが理解しにくい場合
  Fig.3 のリファクタリング前のプログラムでは，getUser メソッドの中での機能が複雑で，理解しにくいコードがある．そこで，処理を理解できるような，適切な名前のメソッドを定義してメソッドの呼び出しに置き換える．
- 同じ処理のコードが書かれている場合
  複数の場所で同じ処理のコードが書かれている場合は，複数の場所で行われている処理を 1 つのメソッドにまとめる．

●リファクタリング前
```
public User getUser(int userID){
    User user = userManager.getUserByPk(userID);
    logger.write("getUser() in" + user.toString());
    return user;
}
```

●リファクタリング後
```
public User getUser(int userID){
    User user = userManager.getUserByPk(userID);
    log("getUser() in", user);
    return user;
}
public void log(String arg, Object obj){
    logger.write(arg + obj.toString());
}
```

Fig. 3　メソッドの抽出

5.2　メソッドのインライン化

メソッドのインライン化とは，メソッド本体が，名前を定義して使用する必要もない程分かり易い場合，メソッド本体を 1 つの大きなメソッドにインライン化し，再度抽出することである．Fig.4 のリファクタリング前の isTokyo メソッドの中の機能は，メソッドにまとめるほど複雑ではない．よって，isTokyo メソッドの呼び出し機能のインライン化を行い冗長を防ぐ．

●リファクタリング前
```
public double getTaxRate(){
    return (this.isTokyo()) ? HIGH_TAX_RATE
        : DEFAULT_TAX_RATE;
}
private boolean isTokyo(){
    return this.prefecture.isTokyo();
}
```

●リファクタリング後
```
public double getTaxRate(){
    return (this.prefecture.isTokyo()) ? HIGH_TAX_RATE
        : DEFAULT_TAX_RATE;
}
```

Fig. 4　メソッドのインライン化

5.3　メソッドの移動

メソッドの移動を行う．メソッドの移動を行う場合の例を Fig.5 に示す．Fig.5 のリファクタリング前の Customer クラスで定義されている amountFor メソッドでは，Customer クラスのフィールドやメソッドよりも，Rental クラスのフィールドやメソッドを多用している．そこで，amountFor メソッドを Rental クラスへ移動する．これにより，外部のクラスのメソッド呼び出し回数が少なくなり，読みやすくシンプルなプログラムになる．

5.4　一時変数の分離

一時変数の分離を行う．一時変数の分離を行う場合の例を Fig.6 に示す．Fig.6 のリファクタリング前では，1 つの temp という変数に，円の面積の値や，円周の長さが代入されている．そのため，変数 temp に何の値が格納されているのか分かりにくく，また，どのような計算を行っているか理解しづらい．そこで，複数回代入される変数がある場合，代入ごとに一時変数名を変えることによって，変数名を見ただけで何の値か分かるようにする．

5.5　シンボリック定数によるマジックナンバーの書き換え

シンボリック定数によるマジックナンバーの書き換えを行う．Fig.7 に例を示す．Fig.7 のリファクタリング前のコードにある”3.14”のように，特別な意味を持つ数字直接が書かれていても，それが何を意味するのか分かり

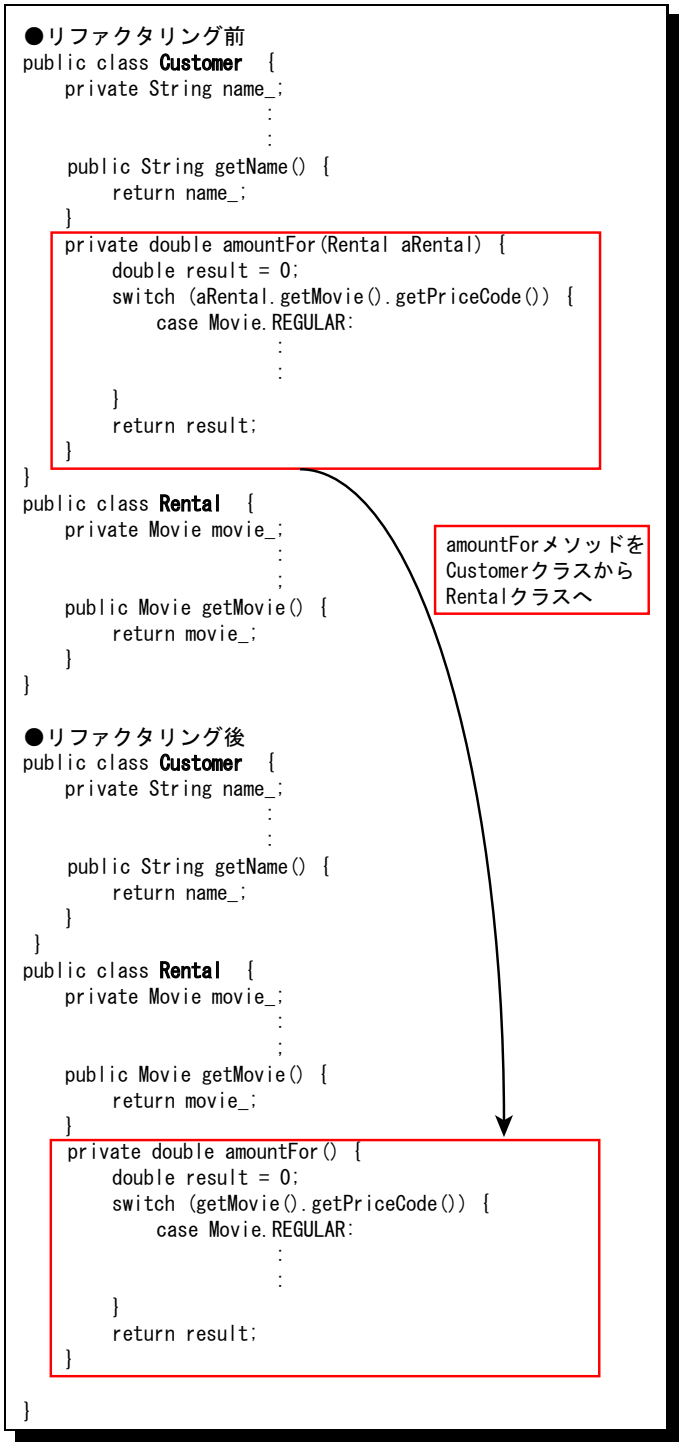

Fig. 5　メソッドの移動

にくい．そこで，”3.14”を”PI”というシンボリック変数に書き換え，定数に意味をもたせることで，プログラムを読み易くする．

5.6　フィールドのカプセル化

フィールドのカプセル化とは，クラス外からアクセス出来ないようにしたい公開フィールドがある場合，それを非公開にして隠蔽し，そのアクセサを用意することである．フィールドのカプセル化の例を Fig.8 に示す．

```
●リファクタリング前
double temp = PI * radius * radiius;
System.out.println(temp);
temp = 2.0 * PI * radius;
System.out.println(temp);

●リファクタリグ後
double area = PI * radius * radiius;
System.out.println(area);
length = 2.0 * PI * radius;
System.out.println(length);
```

Fig. 6　一時変数の分離

```
●リファクタリング前
double area = radius * radius * 3.14;

●リファクタリング後
double area = radius * radius * PI;
```

Fig. 7　シンボリック変数によるマジックナンバーの書き換え

Fig.8 のリファクタリング前において，public 変数 size は，他のクラスから直接参照することができてしまう．そこで，size のアクセス指定子を private にすることで外部クラスから隠蔽し，メソッドを経由して size を参照することでカプセル化を行う．

```
●リファクタリング前
public String size;

●リファクタリング後
private int size;
public int getSize() {return this.size;}
public void setSize(int size) {this.size = size;}
```

Fig. 8　フィールドのカプセル化

5.7　クラス，メソッド，フィールド，属性の名称変更

クラス，メソッド，フィールド，属性の名称が目的を正しく表現していない場合，その機能が分かりにくい．そこで，誰にでも意図が分かるような名称に変更することで，理解しやすいプログラムにする．

5.8　メソッドの引き上げ

メソッドの引き上げを行う．Fig.9 のリファクタリング前ように，複数のサブクラスが同じ機能を行うメソッドを持っている場合，Fig.9 のリファクタリング後のように，スーパークラスにそのメソッドを移動すると，シンプルなプログラムになる．

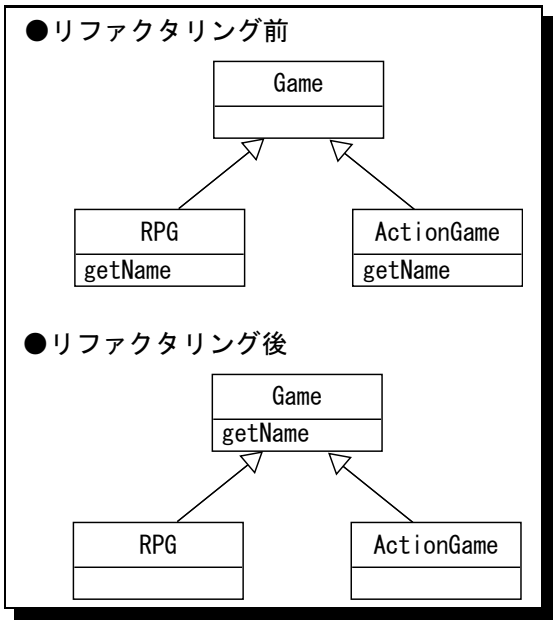

Fig. 9　メソッドの引き上げ

5.9　フィールドの引き上げ

フィールドの引き上げを行う．フィールドの引き上げの例を Fig.10 に示す．Fig.10 のリファクタリング前のように，複数のサブクラスが同じフィールドを持っている場合，リファクタリング後のように，スーパークラスでそのフィールド宣言することで，記述をまとめることができ，シンプルなプログラムになる．

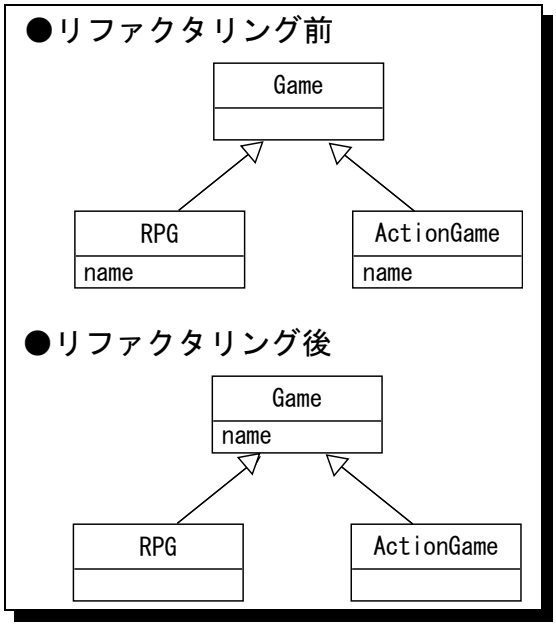

Fig. 10　フィールドの引き上げ

5.10　メソッドの引き下げ

スーパークラスのメソッドが，一部のサブクラスにしか関係していない場合，そのメソッドが定義されている場所は適切ではない．そこで，Fig.11 のように，そのメソッドをサブクラスに移動することによって，適切な表現にする．

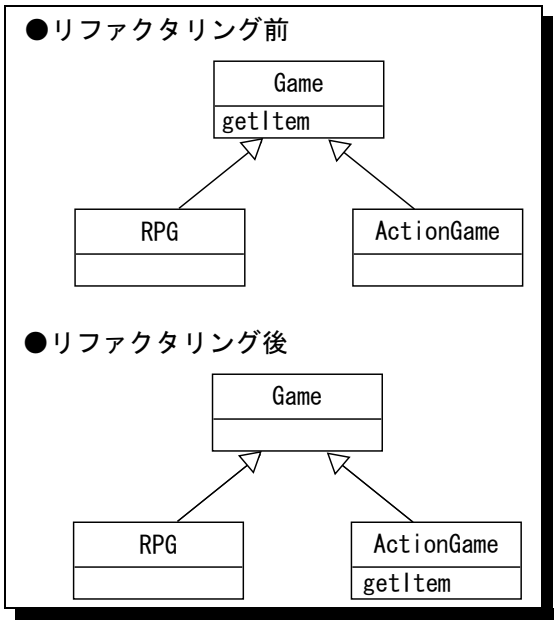

Fig. 11　メソッドの引き下げ

5.11　フィールドの引き下げ

フィールドの引き下げを行う．スーパークラスのフィールドが，一部のサブクラスにしか使われていない場合，そのフィールドが宣言されている場所は適切ではない．そこで，Fig.12 のように，そのフィールドをサブクラスに移動することによって，適切な表現にする．

5.12　デザインパターンを利用したリファクタリング

デザインパターンは「うまくプログラムを作るためのテクニックやコツ」であるため，リファクタリングを行う際には，デザインパターンの適用を考慮する．ここでは，デザインパターンの 1 つである Templete Method パターンを用いたリファクタリングについて示す．

Fig.13 の CustomorController クラスと StaffController クラスは，互いに同じような振舞いをするクラスであり，同じような処理をするメソッドを持っている．しかし，2 つのクラスが何の関連もなく別々に定義されているため，繁雑であり，バグが入り易い．そこで，Templete Method パターンを適応する．CustomorController クラスと StaffController クラスのスーパークラスとして，抽象クラスの ControllerBase クラスを定義

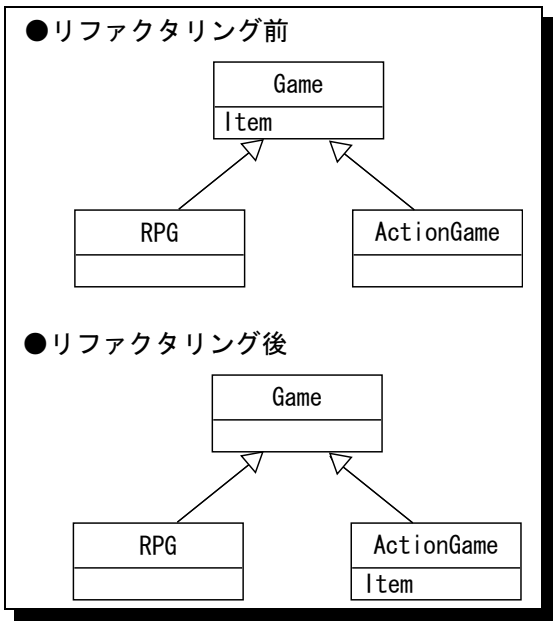

Fig. 12　フィールドの引き下げ

する．ControllerBase クラスでは，CustomorController クラスと StaffController クラスに共通する大まかな処理を行うメソッドを定義することで，ロジックを共有する．また，そのメソッドの内部で呼び出される分割した処理を抽象メソッドとして宣言することで，ControllerBase クラスを継承した CustomorController クラスと StaffController クラスは，この抽象メソッドのみをオーバーライドするだけで目的の処理が記述できる．

このように，デザインパターンをリファクタリングに適用することで，クラス構造を上手に整理し，保守し易くなる．しかし，デザインパターンは，適切な場面に使うと綺麗なプログラムが書けるが，乱用すると逆に理解しにくいプログラムになる．また，デザインパターンをきちんと理解していない人が使うと，逆にバグが入りやすいプログラムになる可能性が高いので忠実にリファクタリングを行うことが大切である．

6　eclipse によるリファクタリング

eclipse では，第 5 章で示したようなリファクタリングの一部を自動化するツールが備わっている．手作業でリファクタリングを行うより有効的であり，様々な利点を有する．本章では，eclipse におけるリファクタリングツールの利用方法を紹介する．

6.1　メソッドの抽出

「メソッドの抽出」では，長いコードのメソッドの一部を抽出し，新たなメソッドを作成することで，より分かり易いコードを保つ．これは，5.1 節に対応している．以下に,「メソッドの抽出」の手順について示す．

1. 抽出したい範囲をドラッグによって選択する．

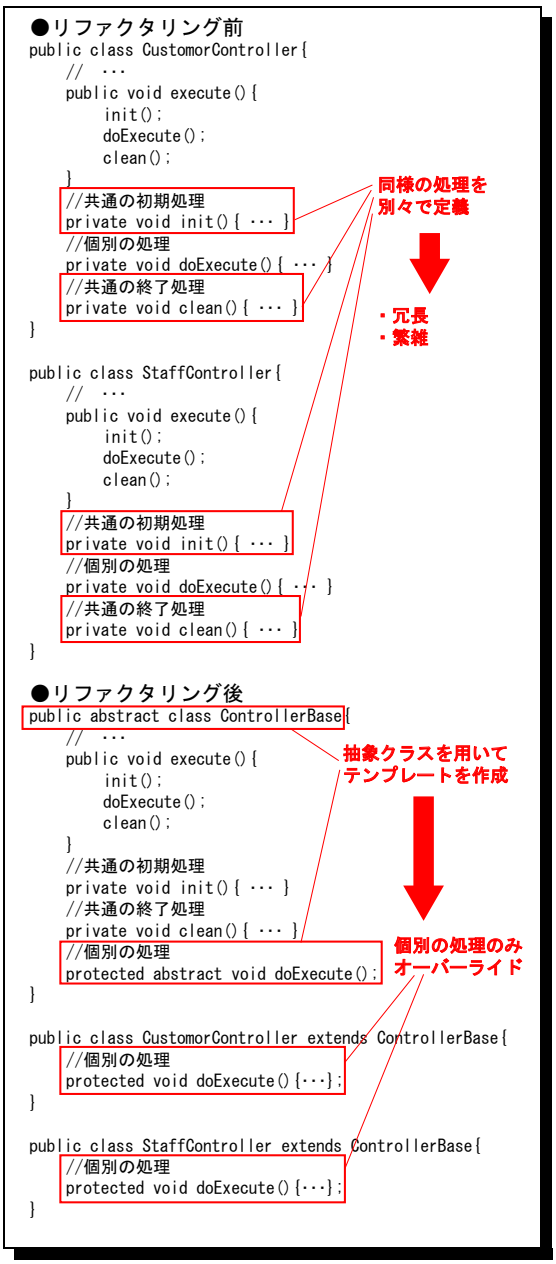

Fig. 13　Templete Method の形成

2. 【右クリック】→【リファクタリング】→【メソッドの抽出】を選択する (Fig. 14)．
3. 変更したい対象を変更する (Fig. 15)．
   - 「メソッド名 (N):」の部分に新しい名前を入力する．
   - 【編集】ボタンをクリックして，対象の引数の型や名前を変更する．
4. 【プレビュー】をクリックし，変更前と変更後のソースを確認する．
5. 【OK】ボタンをクリックする．

6.2　移動

「移動」では，クラス，メソッド，およびフィールドの移動を行う．これは，5.3 節に対応している．インス

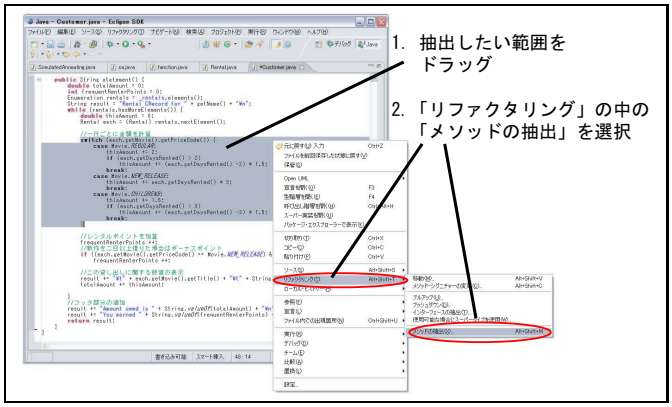

Fig. 14　メソッドの抽出手順

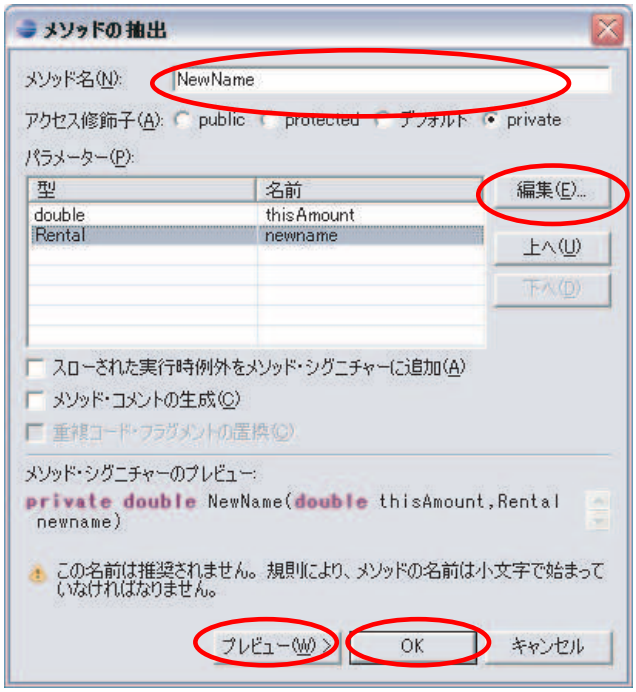

Fig. 15　メソッドの抽出

タンスメソッドやフィールドの移動は移動対象が限られているため，手作業で不適切な位置に移動を行うと，エラーを生じる可能性がある．したがって，このツールを用いることで上記の問題を回避することができる．以下に,「移動」の手順について示す.

1. 移動させたい対象を「パッケージ・エクスプローラー」などで選択する.
2. 【右クリック】→【リファクタリング】→【移動】を選択する (Fig. 16).
3. 【参照 (B):】をクリックし，移動先の名前を入力する (Fig. 17).
4. 【OK】ボタンをクリックする.
5. 【プレビュー】をクリックし，変更前と変更後のソースを確認する.
6. 【OK】ボタンをクリックする.

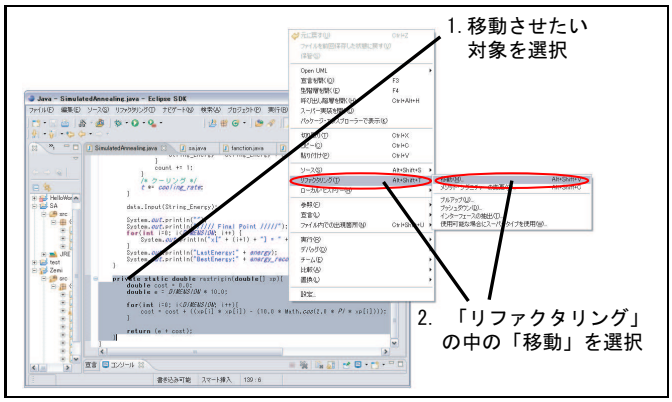

Fig. 16　移動手順

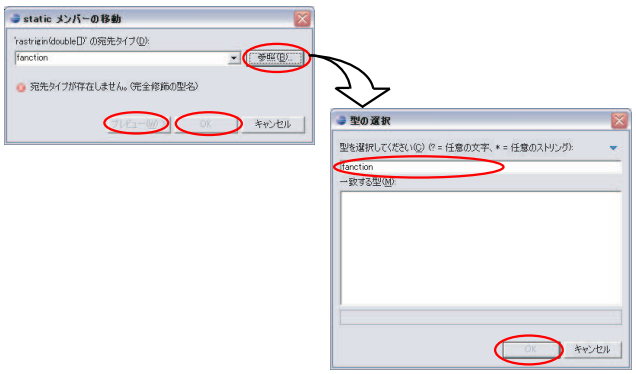

Fig. 17　移動

6.3　フィールドのカプセル化

「フィールドのカプセル化」では，フィールドへのアクセスをゲッタやセッタに置き換える．これは，5.6 節に対応している．以下に,「フィールドのカプセル化」の手順について示す.

1. フィールドをドラッグによって選択する.
2. 【右クリック】→【リファクタリング】→【フィールドのカプセル化】を選択する (Fig. 18).
3. 変更したい対象を変更する (Fig. 19).
   - 「Getter 名 (G):」,「Setter 名 (S):」の部分にそれぞれの名前を入力する.
   - 「新規メソッドを次の後に挿入 (I):」の部分で，Getter および Setter の挿入部分を指定する.
4. 【プレビュー】をクリックし，変更前と変更後のソースを確認する.
5. 【OK】ボタンをクリックする.

6.4　名前変更

「名前変更」では，Java 要素（パッケージ，クラス，フィールド，メソッド）の名前の変更を行う．これは 5.7 節に対応している．また，単に名前を変更するだけでな

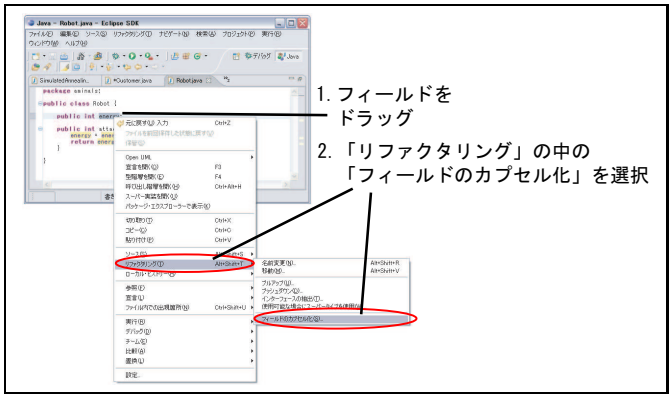

Fig. 18　メソッドのカプセル化手順

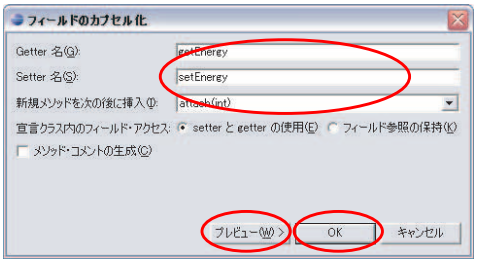

Fig. 19　メソッドのカプセル化

く，変更したクラスやメソッドなどの参照先も変更されるため，手作業によるリファクタリングよりミスが少ない．クラス，フィールド，およびメソッドの「名前変更」の手順を以下に示す．

1. 変更したい対象の名前をドラッグで選択する．
2. 【右クリック】→【リファクタリング】→【名前変更】を選択する (Fig. 20)．
3. 「新しい名前 (N):」の部分に新しい名前を入力する (Fig. 21)．
4. 【OK】ボタンをクリックする．

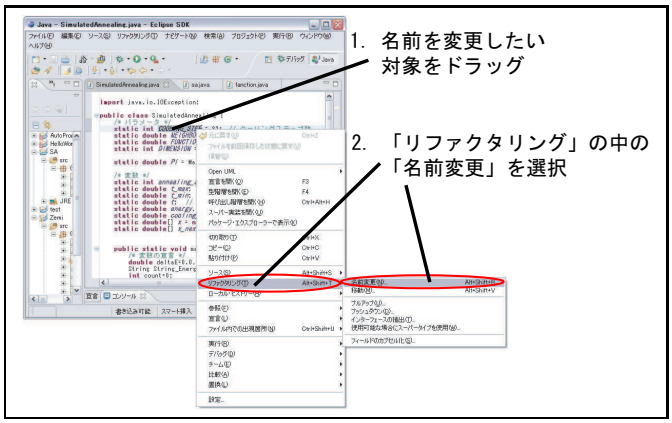

Fig. 20　名前変更手順

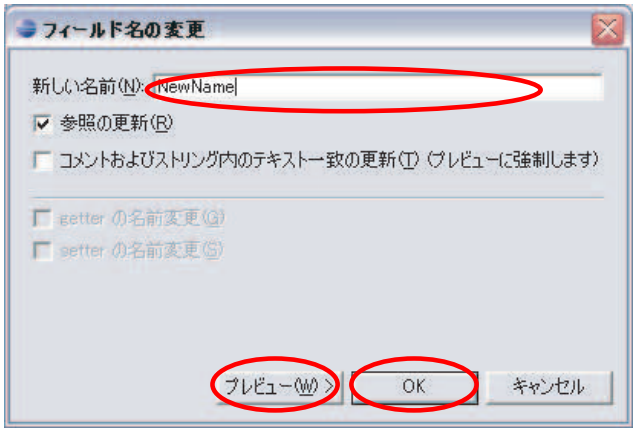

Fig. 21　名前変更

6.5　メソッド・シグニチャの変更

「メソッド・シグニチャの変更」では，メソッドの名前だけでなく，アクセス修飾子や戻り値の型，引数の型や名前も変更することができる．「メソッド・シグニチャの変更」の手順を以下に示す．

1. 変更したい対象のメソッドをドラッグで選択する．
2. 【右クリック】→【リファクタリング】→【メソッド・シグニチャの変更】を選択する (Fig. 22)．
3. 変更したい対象を変更する (Fig. 23)．
   - 「アクセス修飾子 (C):」で修飾子を選択する．
   - 「戻りの型 (T):」，「メソッド名 (N):」のそれぞれに新しい型および名前を入力する．
   - 【追加】または【編集】ボタンをクリックして，対象の引数の型や名前を追加もしくは変更する．
4. 【プレビュー】をクリックし，変更前と変更後のソースを確認する．
5. 【OK】ボタンをクリックする．

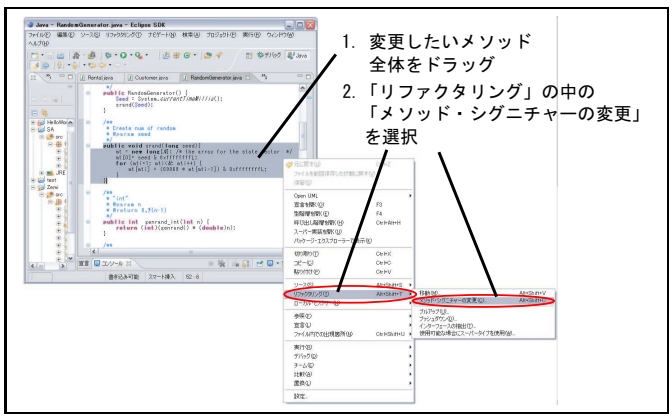

Fig. 22　メソッド・シグニチャの変更手順

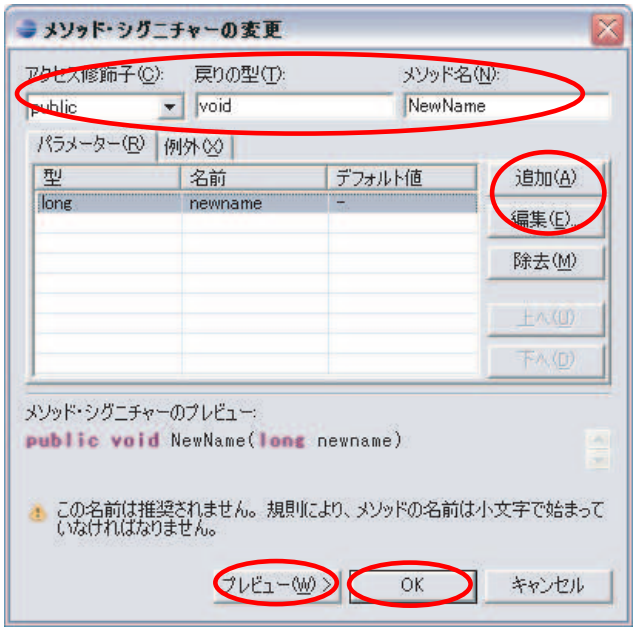

Fig. 23　メソッド・シグニチャー変更

7　おわりに

本ゼミでは，リファクタリングの基本的な手法について説明した．プログラムが大規模になるほど，リファクタリングの効果は顕著となり，プログラムの変更や修正を繰り返す場面でリファクタリングは必須のものとなる．理解や修正が容易なプログラム作成を心掛けて，リファクタリングを習慣づけていただきたい．

参考文献

1) 長谷川 裕一，齋川 博文：『プログラムの育てかた 現場で使えるリファクタリング入門』，ソフトバンクパブリッシング株式会社，2005

2) マーチン・ファウラー：『リファクタリング プログラミングの体質改善テクニック』，株式会社ピアソン・エデュケーション，2000

3) オブジェクト倶楽部
http://www.objectclub.jp/