リファクタリングゼミ

ゼミ担当者　：　西田 健，田中 美里，田中 慎吾，山田 幸史朗
指導院生　：　木浦 正博，天白 進也
開催日　：　2007 年 5 月 26 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，内部構造を改善することである．これはプログラムの理解や修正を容易とすることを目的としている．リファクタリングのテクニックはデザインパターンなどと並び，現代における有効な開発手法として利用されている．

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について解説する．また，ゼミの後半では，eclipse のリファクタリングツールを利用し，実際のリファクタリング作業を行う．

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

一度は完成したプログラムであっても，修正，機能追加などによって長い時間をかけて変更される場合がある．このような場合，プログラムの当初の設計は崩れていき，プログラマにとっても読みづらく，かつ理解しにくいコードになってしまう．これは新たな変更への対応の遅れ，または潜在的なバグの発生などの問題へとつながる．

上記のような問題を解決するために，設計者の意図を明確に読み取れる，シンプルで構造的に優れたコードを保つ必要がある．その際にコードを整理するテクニックとして必要となるのがリファクタリングの技術である．

2.2　リファクタリングの効果

リファクタリングを行うことによって得られる効果を以下に示す．

- 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．

リファクタリングによるコードの変更は，人にとって理解し易くするための変更であり，このために一時的にプログラムの実行速度を落としてしまうこともある．しかし，リファクタリングを行うことによって，後々のパフォーマンスチューニングの効率はアップする．これは，リファクタリングによってコードの設計が明確になり，パフォーマンス解析の際に，どこが問題となっているのか，より細かな範囲に絞って解析できるためである．また，コードが明確であるということは，どのようなチューニングが有効であるか，選択し易いというメリットにもなる．

3　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時期が存在する．主な時期を以下に 4 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
  機能追加の前にリファクタリングを行うことによってコードが整理され，それに伴って機能の追加も行い易くなる．

- バグフィックスの作業時
  バグが発生した際，プログラムを理解するためにリファクタリングを行う．

- コードレビュー時
  レビュー時に，他者の書いたコードを理解するのに役立つ．

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行為である．そのため，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．テストの自動化を行うフレームワークとして JUnit がある．JUnit は Eclipse にプラグインとして提供されている．

4.2　リファクタリングを用いたプログラミングの流れ

リファクタリングを行う際のテストは，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．

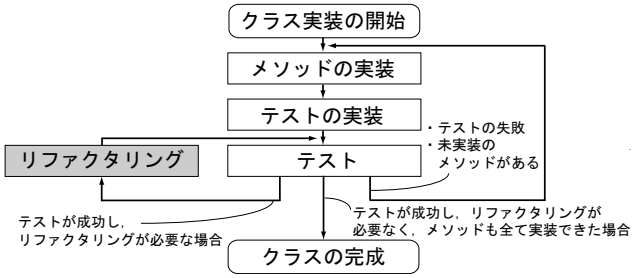

Fig. 1　クラス完成までのプログラミングの流れ (参考文献 5 より引用)

- 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．

4.3　リファクタリングが必要な場所

プログラムの中で理解や修正，拡張が難しい部分のことを「不吉な匂い」と呼ぶ．これはリファクタリングが必要な部分であり，その具体例を以下に挙げる．

- 長大な処理

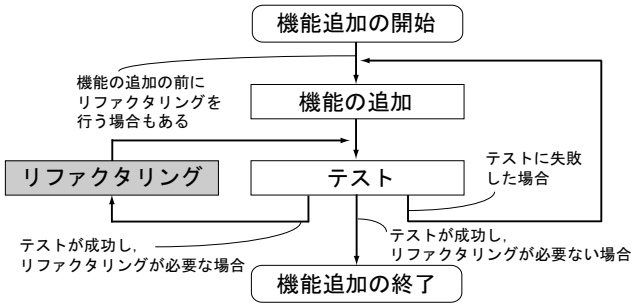

Fig. 2　機能追加のプログラミングの流れ (参考文献 5 より引用)

  - 長過ぎるメソッド
    長くて理解し難いメソッドがある．
  - 巨大なクラス
    余りに多くの処理を行い過ぎているクラスがある．

- 処理の重複
  - 重複したコード
    同様の処理が，2 箇所以上に記述されている．
  - 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．
  - 変更の分散
    1 つの変更要求に対し，複数のクラスへの修正が必要となる．
  - パラレル継承
    あるスーパークラスに対して新しいサブクラスを定義する度に，別のスーパークラスに対しても新しいサブクラスを定義しなければならない．
  - クラスのインタフェース不一致
    シグニチャ（メソッドの名前，型，引数）は異なるが，内部の処理は同じメソッドが存在する．
  - switch 文
    switch 文の使用は，処理の重複につながるケースが多い．

- 不要なコード
  - 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．

- 仲介人
  メッセージを受け取って，次のオブジェクトに渡すだけのメソッドが多い．

- 不適切なクラスの関係
  - 属性，操作の過剰な依存
    他のクラスの特定の属性や操作を，過剰に利用している．
  - 相続拒否
    スーパークラスの属性や操作を，有効に活用していないサブクラスがある．

- 不適切なクラス
  - 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（getter と setter）しか持たず，負った責務の少ないクラスがある．
  - 一時的属性
    オブジェクトは値を属性として常に保持するものであるにも拘らず，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存在する．

- 非オブジェクト指向
  - 基本データ型への執着
    int 型，double 型などの基本データ型を多用し，小さなオブジェクトの利用を嫌がる．
  - 過剰な公開
    private として指定すべき，フィールド変数やメソッドを public として公開している．

- その他
  - 不適切な名前
    メソッドの機能と名前が一致していない．
  - 多過ぎる引数
    引数が多過ぎる．または引数に同じ型が続いている．
  - コメント
    処理が複雑であるのを補うために，丁寧なコメントを書いている．

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，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．

- 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，同一の処理を 1 つのメソッドにまとめる．

- メソッドが長すぎる場合
  メソッドは，スクロールせずに画面に全てを表示できる長さが理想的である．長すぎるメソッドの一部を抽出して，別のメソッドにまとめる．

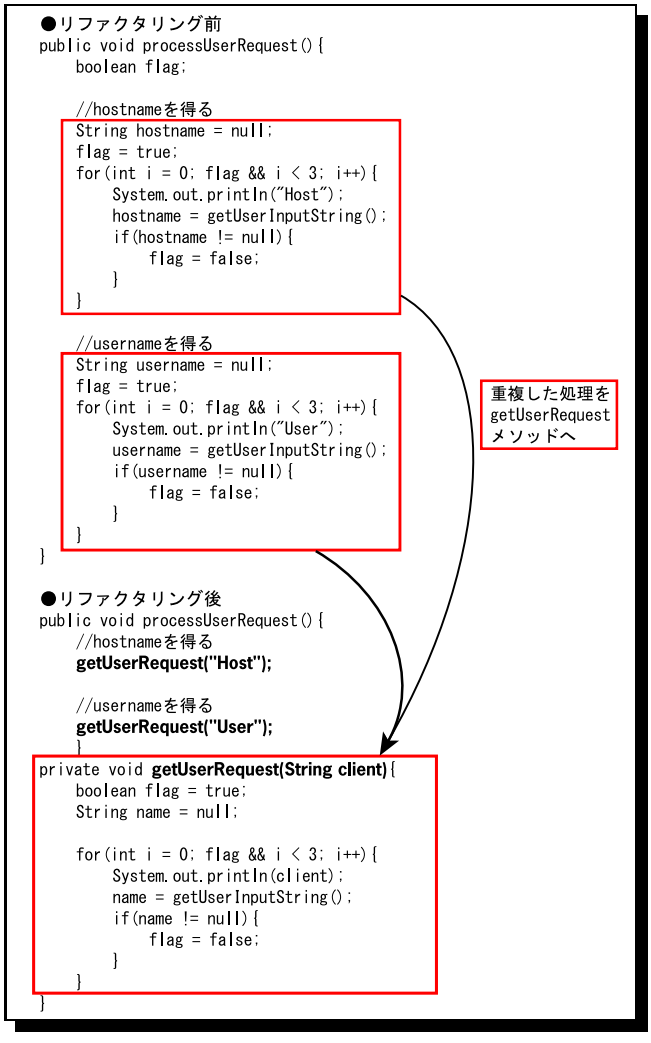

Fig. 3　メソッドの抽出 (出展：自作)

5.2　メソッドの移動

リファクタリングにおけるメソッドの移動の例を Fig. 4 に示す．Fig. 4 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変数やメソッドよりも，Rental クラスのフィールド変数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．

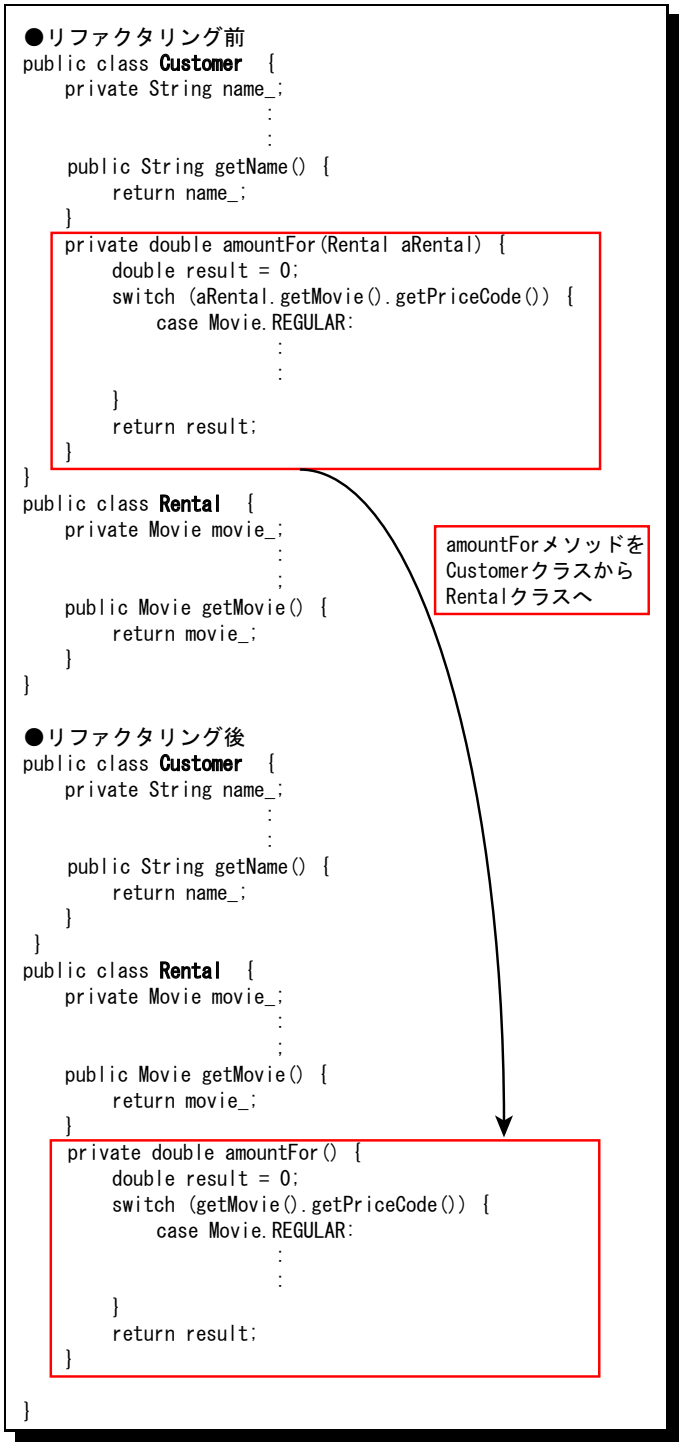

Fig. 4　メソッドの移動 (参考文献 5 より引用)

5.3　メソッドのインライン化

メソッドのインライン（文中）化とは，あるメソッドの中身をコール元へと組み込むことを言う．メソッドの機能が，名前を定義する必要もないほど分かり易い場合に用いる．Fig. 5 におけるリファクタリング前の largerThanFive メソッドの中の機能は，あまり複雑ではない．よって，largerThanFive メソッドの機能のインライン化を行って冗長を防ぐ．

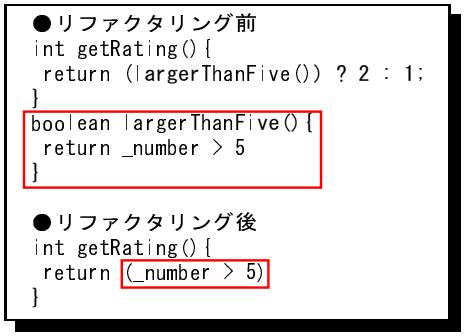

Fig. 5　メソッドのインライン化 (出展：自作)

5.4　シンボリック定数によるマジックナンバーの書き換え

プログラム中で特別な意味を持つ数字をマジックナンバーといい，その数値を示す定数のことをシンボリック定数という．例を Fig. 6 に示す．リファクタリング前における数値 100 は，変数 length に代入できる最大値を示している．しかし，このままでは数値の意図が分かりにくく，また最大値を変更する場合，全ての 100 を書き換えなければならない．そこで，MAX__LENGTH というシンボリック定数に値を代入することで，プログラムの可読性を高め，変更を簡単にする．

Fig. 6　シンボリック定数によるマジックナンバーの書き換え (出展：自作)

5.5　一時変数の分離

一時変数の分離を行う場合の例を Fig. 7 に示す．Fig. 7 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難い．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．

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

Fig. 7　一時変数の分離 (参考文献 5 より引用)

5.6　クラス，メソッド，フィールド変数，属性の名称変更

クラス，メソッド，フィールド変数，属性の名称が目的を正しく表現していない場合，その機能が分かるように名称を変更する．メソッドは getSize など「動詞+名詞」の形にすることが多い．

5.7　フィールド変数のカプセル化

フィールド変数のカプセル化とは，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変数がある場合，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．フィールド変数のカプセル化の例を Fig. 8 に示す．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 int 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　フィールド変数のカプセル化 (参考文献 5 より参照)

5.8　メソッド，フィールド変数の引き上げ

複数のサブクラスが同じメソッドやフィールド変数を持っている場合，Fig. 9 に示す「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したり，スーパークラスでそのフィールド変数宣言をすることで，シンプルなプログラムになる．

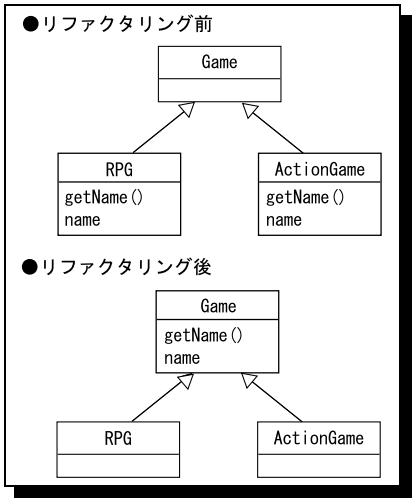

Fig. 9　メソッドとフィールド変数の引き上げ (参考文献 5 より参照)

5.9　メソッド，フィールド変数の引き下げ

スーパークラスのメソッドやフィールド変数が，一部のクラスにしか使われていない場合，Fig. 10 に示したように，そのメソッドやフィールド変数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．

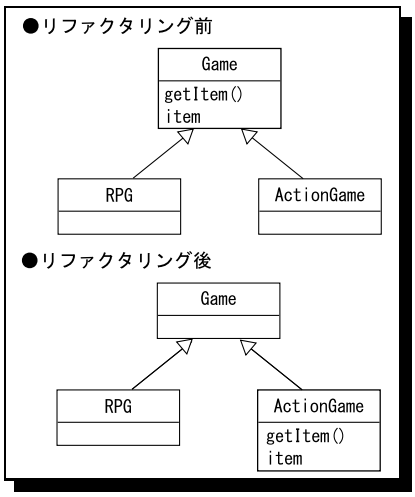

Fig. 10　メソッドとフィールド変数の引き下げ (参考文献 5 より参照)

5.10　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.

11 の CustomorController クラスと StaffController クラスは互いに同じような振る舞いをするクラスであり，同じような処理を行う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，煩雑であり，バグの要因にもなる．そこで，Templete Method パターンを適応する．CustomorController クラスと StaffContoroller クラスのスーパークラスとして，抽象クラスの ControllerBase クラスを定義する．ControllerBase クラスでは，CustomorController クラスと StaffController クラスに共通する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，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しておくことで，継承先の CustomorController クラスや StaffController クラスは，この抽象メソッドをオーバーライドすること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応することで，クラス構造が整理され，保守が簡単になる．しかし，デザインパターンは，適切な場面に使うと明確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，デザインパターンへの理解の浅い者が使うと，逆にバグが入りやすいプログラムになる恐れが高いため，忠実にリファクタリングを行うことが大切となる．

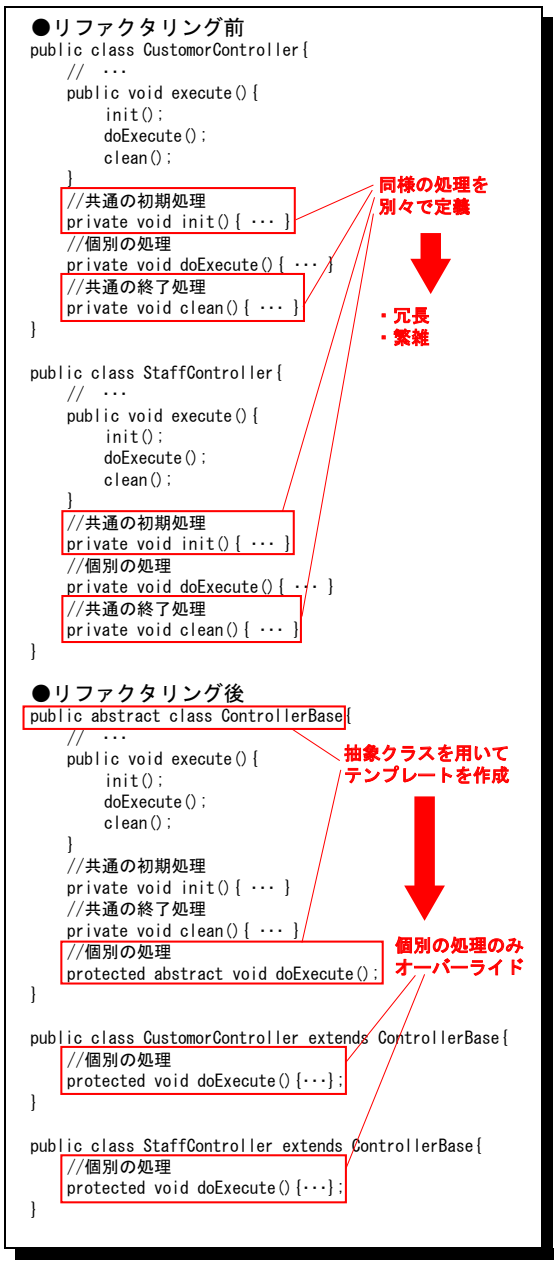

Fig. 11　Templete Method の形成 (参考文献 5 より引用)

6　リファクタリング演習

リファクタリングは本来手作業で行うものである．しかし，人の手によるソースコードの変更は，勘違いやミスによって新たなバグを埋め込んでしまう危険性も孕んでいる．よって，専用のツールを利用し，リファクタリング作業の一部を自動化することで，より安全にリファクタリングを行う事ができる．eclipse には，第 5 章で示したようなリファクタリング作業の自動化ツールが備わっている．本演習では，手作業とツールによるリファクタリングを実際に行い，作業手順について確認する．

6.1　リファクタリングの手順

リファクタリングを行う際，2 つ以上のリファクタリングを混在して進めてはならない．1 回に行うリファクタリングは 1 つに限り，ステップ・バイ・ステップで作業を進める．これによって，テストによるリファクタリングの成否の確認が簡単となり，リファクタリングに失敗した場合も，コードを元の状態に戻し易い．

具体的なリファクタリングの手順を以下に示す．

1. リファクタリング部位をコメントアウトする．
2. リファクタリング作業を行う．
3. テストする．テストの結果によって，そのステップのリファクタリング作業を終了する，もしくはコードを元に戻す．
4. リファクタリングの完了．

作業を後退する場合，リファクタリング前のコードが必要となる．よってコードを改変する前，コメントアウトをして以前のソースコードを保存して置くことが重要である．

6.2　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．
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. 12)．
3. 「メソッド名 (N):」の部分に新しい名前を入力する．
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．

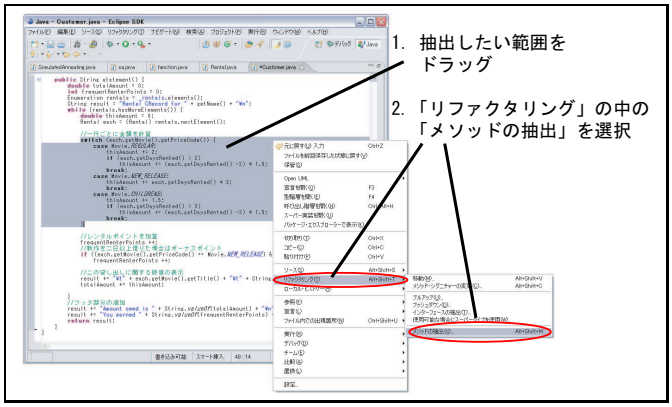

Fig. 12　メソッドの抽出手順 (参考文献 5 より引用)

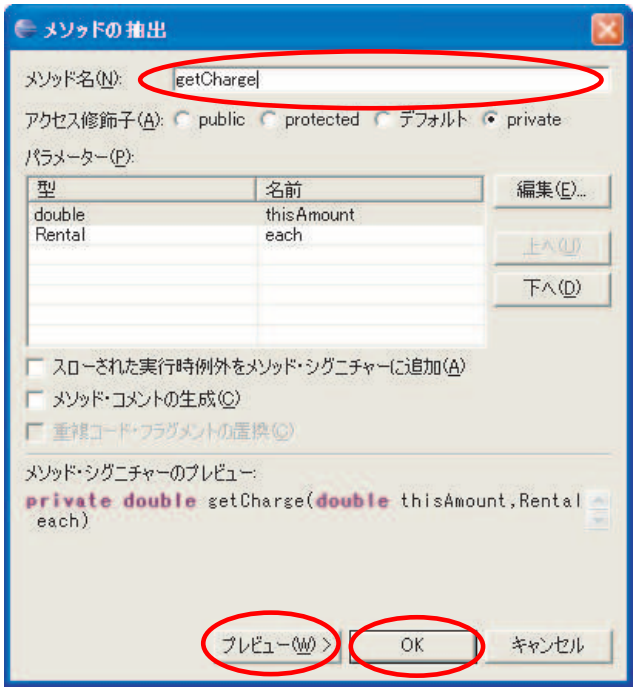

Fig. 13　メソッドの抽出 (出展：自作)

6.3　名前変更

「名前変更」では，Java 要素（パッケージ，クラス，フィールド変数，メソッド）の名前の変更を行う．これは 5.6 節に対応している．また，単に名前を変更するだけでなく，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，フィールド変数，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. 14)．
3. 「新しい名前 (N):」の部分に新しい名前を入力する (Fig. 15)．
4. 【OK】ボタンをクリックする．

6.4　メソッドの移動

「メソッド移動」は，5.2 節に対応している．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．そこで，このツールを用いることでその問題を回避することができる．以下に，「メソッドの移動」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. 【プレビュー】をクリックすると (Fig. 17)，変更前と変更後のソースが表示される．

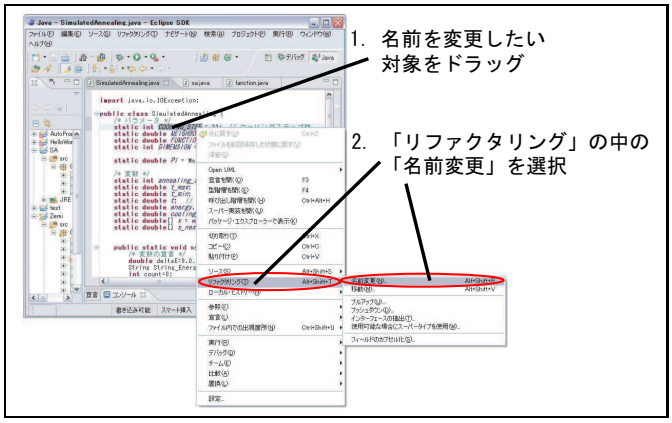

Fig. 14　名前変更手順 (参考文献 5 より参照)

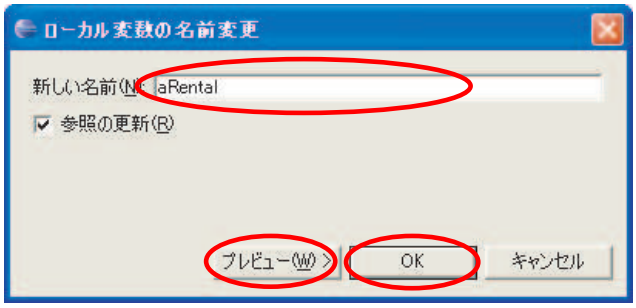

Fig. 15　名前変更 (出展：自作)

何か問題が検出された場合は Fig. 18 のような画面が表示されるので内容を見た後，継続かキャンセルを選択する．

4. ソース確認後【OK】ボタンをクリックする．

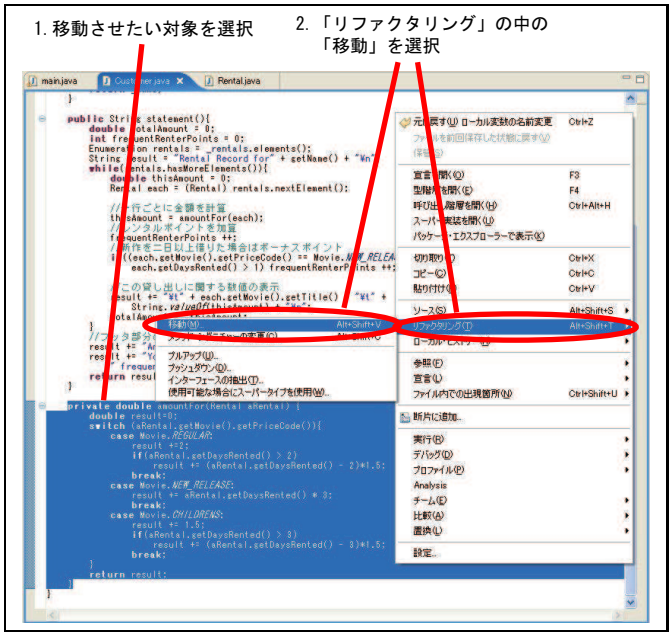

Fig. 16　移動手順 (出展：自作)

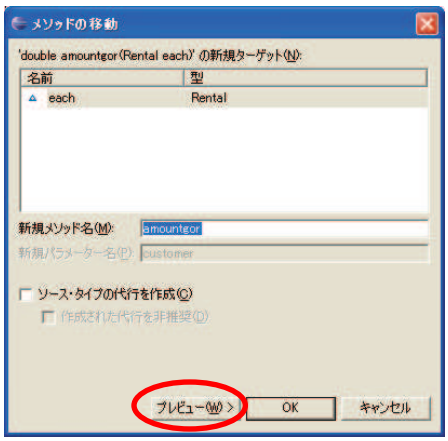

Fig. 17　移動 (出展：自作)

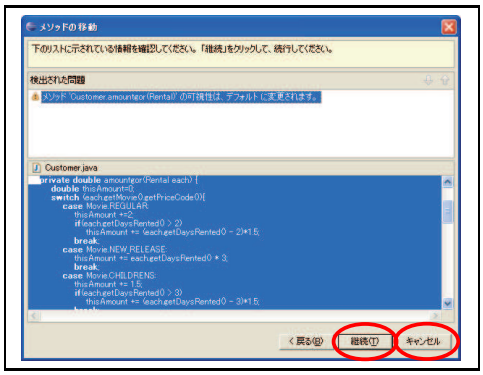

Fig. 18　問題検出 (出展：自作)

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/

4) 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株式会社，2007 年

5) 2006 年リファクタリングゼミ
http://mikilab.doshisha.ac.jp/dia/seminar/2006