修 士 論 文

メトリクスの測定による
リファクタリング支援の自動化の研究

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

田畑 敦史

2008 年 3 月

修 士 論 文

メトリクスの測定による
リファクタリング支援の自動化の研究

指導教官　鈴木正人 准教授

審査委員主査　鈴木正人 准教授
審査委員　落水浩一郎 教授
審査委員　青木利晃 特任准教授

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

610057 田畑 敦史

提出年月: 2008 年 2 月

概 要

ソフトウェア開発の過程において、複雑になり保守・拡張性が損なわれたソースコードを改善する手段としてリファクタリングがある。リファクタリングとは、ソフトウェアの外的振る舞いを保ったままで内部の構造を改善していく作業で、これを行いソフトウェアのソースコード内の複雑な構造を改善することにより、保守・拡張性を向上させる。しかしながら、ソースコード全体からリファクタリングの対象である複雑な箇所を発見するのは困難である。ソースコードの複雑度を調べる手段の１つとして、ソフトウェアメトリクスがある。ソフトウェアメトリクスは、ソースコードの評価を数値的に行うもので、複雑度も数値的に求めることができる。本研究では、ソフトウェアメトリクスを利用することで、複雑な箇所を自動的に求め、その解決方法を提示することによるリファクタリング支援を提案する。

目 次

図 目 次

表 目 次

第1章　序論

1.1　背景

ソフトウェア開発の過程において、過程が進む毎にソースコードが複雑になり、ソースコードの可読性が失われることや、ソフトウェアの変更や拡張の際に発生する変更箇所が増加することから、ソフトウェアの保守・拡張が困難なる問題がある。

開発の際にはこの問題を解決してソフトウェアの保守・拡張を行いやすくする必要があり、その方法の1つとしてリファクタリングというものがある。リファクタリングは、ソフトウェアの外的振る舞いを保ったままで内部の構造を改善していく作業であり、リファクタリングによってソースコード内の複雑になった箇所を改善することにより、この問題を解決しようというものである。

しかしながら、ソースコード全体を見て、改善箇所を発見することは困難である。更にソフトウェア開発現場において、スケジュールの関係からリファクタリングには長い時間を与えることは難しく、これらの事から実際にリファクタリングを行うことは避けられがちである。

1.2　目的

本研究では、メトリクスを利用した手法によりソースコード内からリファクタリングを行わなければならない場所を見つける作業、および、それを改善するために行わなければならないリファクタリング操作候補の選出を自動化することにより、リファクタリングを行う際の支援を行うことを目的としている。

従来リファクタリングは、改善箇所の困難さ困難さと時間上の都合からソフトウェア開発現場では避けられてきた。そこで、本研究ではリファクタリングを行う際の支援を自動化することにより、その困難さと時間的都合による不都合を解消し、リファクタリングを実用的なものにすることを目的としている。

1.3　構成

本研究の構成は以下のようになっている

- 第2章では、従来行われてきたリファクタリング支援について紹介する。

- 第 3 章では、リファクタリング支援をどのように自動化するかについて紹介する。
- 第 4 章では、自動化したリファクタリング支援を実装するための設計を行う。
- 第 5 章では、自動化したリファクタリング支援の実装を行う。
- 第 6 章では、実装した支援を利用してどの程度リファクタリングが行えるかの実験を行う。
- 第 7 章では、6 章の実験から本研究のリファクタリング支援の評価を行う。
- 第 8 章では、本研究のまとめと今後の課題について述べる。

第2章　従来研究によるリファクタリング支援

2.1　リファクタリングとは

外部から見たときの振る舞いを保ちつつ、理解や修正が簡単になるように、ソフトウェアの内部構造を変化させること。[1]

2.2　FowlerによるBadSmellとその解決方法の定義

Fowler はソースコードを改善する際のリファクタリングの開始と終了の判断基準として、BadSmell を定義することによる支援方法の提案を行った。

BadSmell とは、ソースコード上において改善すべき箇所であり、この BadSmell がソースコード上に存在するかどうかでリファクタリングを終了させるかどうかの判断を行う。

また、Fowler の定義する BadSmell にはそれを解決するためのリファクタリング操作が提示されており、このリファクタリング操作をソースコードに施すことを繰り返し、ソースコードから BadSmell を消滅させることにより、ソースコードの改善を行う。

2.2.1　BadSmellの詳細

BadSmell はコードが重複している、メソッドが長すぎて理解し難いといった、ソースコードで改善したほうがよい部分を指す。Fowler は、BadSmell の種類毎に個別に対応したリファクタリング操作を提示している。以下にその BadSmell の一部と対応したリファクタリング操作を紹介する。

- Long Method
- Long Parameter List
- Switch Statements
- Feature Envy

Long Method

Long Method は、メソッドのコードの行数が長すぎることによりコードが読み辛くなっている BadSmell である。

例えば図 2.1 の様な 30 行の長さのメソッドに対して、リファクタリングを行うものがそのコードが長すぎることが原因で内容が理解し難いと判断すれば、そのメソッドは Long Method となる。

```
 1  int foo(){
 2   int x = 0;
     ...
     ...
12   if(flag) x++;
     ...
     ...
29   return x;
30  }
```

図 2.1: Long Method の例

Long Parameter List

Long Parameter List は、メソッドの引数の数が多すぎることによりコードが読み辛くなっている箇所を指す BadSmell である。

図 2.2 の様に引数の数が 5 つのメソッドに対して、リファクタリングを行うものが引数が多すぎることでそのコードの内容が理解し難いと判断すれば、そのメソッドは Long Parameter List である。

```
void foo(int n1, int n2, int n3, double d1, double d2, String str){
 ...
 ...
 ...
}
```

図 2.2: Long Parameter List 例

Switch Statements

Switch Statements は、コードのあちこちに switch 文あるいは if 文による分岐があり、機能の追加を行う際に複数の箇所を変更しなければならないクラスの箇所を指す BadSmell である。

図 2.3 の場合、enum WEAPON による分岐が複数の箇所で発生している故、enum WEAPON に識別子を 1 つ追加する場合にメソッド foo1 と foo2 の両方に処理を追加しなければならないため Switch Statements となる。

```
class Foo{
 enum WEAPON{TONFA, NUNCHAKU, TJR};
 int foo1(WEAPON type){
  switch(type){
   case TONFA:
    return 1;
   case NUNCHAKU:
    return 2;
   case TJR:
    return 3;
   default:
    Throw new WeaponException("Unknown Weapon");
  }
 }
 String foo2(WEAPON type){
  switch(type){
   case TONFA:
    return "Tonfa kick !";
   case NUNCHAKU:
    return "Nunchakus union";
   case TJR:
    return "Three Joint Rod";
   default:
    Throw new WeaponException("Unknown Weapon");
  }
 }
}
```

図 2.3: Switch Statements の例

Feature Envy

Feature Envy は、メソッドが自分のクラスのリソースをまったく使用せず、且つ他のクラスのリソースを使用しているメソッドを指す BadSmell である。

例えば、図 2.4 のメソッド envy の様に、所属する Foo クラスのフィールド変数やメソッドをまったく使用せず、Point クラスのメソッドを使用している状態がそれにあたる。

```
class Foo{
    private int t1, t2, t3;
    int hoge1(){
        ...
    }
    String hoge2(){
        ...
    }
    int envy(Point p){
        return p.getX() * p.getY();
    }
}
class Point{
    private int x;
    private int y;
    int getX(){
        return x;
    }
    int getY(){
        return y;
    }
    ...
}
```

図 2.4: Feature Envy の例

2.2.2 リファクタリング操作

Fowler は、定義した BadSmell に対して、それを解決するためのリファクタリング操作の提案を行っている。

表 2.1 は Fowler が提案した BadSmell とリファクタリング操作の対応である。

以下に、そのリファクタリング操作の一部を紹介する。

表 2.1: BadSmell を解決するためのリファクタリング操作の候補一覧

Long Method	メソッドの抽出 一時変数のメソッドへの置き換え
Long Parameter List	メソッドによる引数の置き換え オブジェクトそのものの受け渡し 引数オブジェクトの導入
Feature Envy	メソッドの移動

メソッドの抽出

メソッドの抽出は、図 2.5 の様にメソッド内のコードの一部を別のメソッドとして定義する。結果としてメソッドの行数を減らすリファクタリング操作である。

```
int foo(){
 int x = 0;
 ...
 for(int i=0;i<10;i++){
  ...
 }
 ...
 return x;
}
```

→

```
int foo(){
 int x = 0;
 ...
  foo2();
 ...
 return x;
}

void foo2(){
 for(int i=0;i<10;i++){
  ...
 }
}
```

図 2.5: メソッドの抽出の例

メソッドの移動

メソッドの移動は、メソッドを今所属するクラスとは別のクラスに移動させるリファクタリング操作である。

図??の例では envy メソッドをクラス Foo からクラス Point に移動している。

```
class Foo{
    private int t1, t2, t3;
    int hoge1(){
        ...
    }
    String hoge2(){
        ...
    }
    int envy(Point p){
        return p.getX() * p.getY();
    }
}
class Point{
    private int x;
    private int y;
    int getX(){
        return x;
    }
    int getY(){
        return y;
    }
    ...
}
```

→

```
class Foo{
    private int t1, t2, t3;
    int hoge1(){
        ...
    }
    String hoge2(){
        ...
    }
}
class Point{
    private int x;
    private int y;
    int getX(){
        return x;
    }
    int getY(){
        return y;
    }
    int envy(){
        return x * y;
    }
    ...
}
```

図 2.6: メソッドの移動の例

一時変数の置き換え

一時変数の置き換えは、図 2.7 の様にメソッド内で一度しか使用されない一時変数に着目し、その定義および値の決定を行っている場所を別のメソッドにすることにより、メソッドの行数を減らすリファクタリング操作である

```
void foo(int n1, int n2){
 int result = n1 + n2;
 System.out.println(result);
}
```
→
```
void foo(int n1, int n2){
 System.out.println(result(n1, n2));
}

int result(int n1, int n2){
 return n1 + n2;
}
```

図 2.7: 一時変数の置き換えの例

メソッドによる引数の置き換え

メソッドによる引数の置き換えは、図 2.8 の様にメソッドに引数として渡さなくてもよい場合に、その引数を削除してメソッドの引数を減らすリファクタリング操作である。

```
void foo1(){
 int a = getParam();
 System.out.println("["+foo2(a)+"]");
}

int foo2(int a){
 return a * 2;
}
```
→
```
void foo1(){
 System.out.println("["+foo2()+"]");
}

int foo2(){
 return getParam() * 2;
}
```

図 2.8: メソッドによる引数の置き換えの例

オブジェクトそのものの受け渡し

オブジェクトそのものの受け渡しは、図 2.9 の様に、オブジェクトから値を取り出さずに直接オブジェクトをメソッドに渡すことにより、メソッドの引数を減らすリファクタリング操作である。

```
int n1 = param.getOne();
int n2 = param.getTwo();
foo(n1, n2);
```
→
```
foo(param);
```

図 2.9: オブジェクトそのものの受け渡しの例

引数オブジェクトの導入

引数オブジェクトの導入は、図 2.10 の様に複数の引数の値を保持する新しいクラスを導入することによりメソッドの引数を減らすリファクタリング操作である。

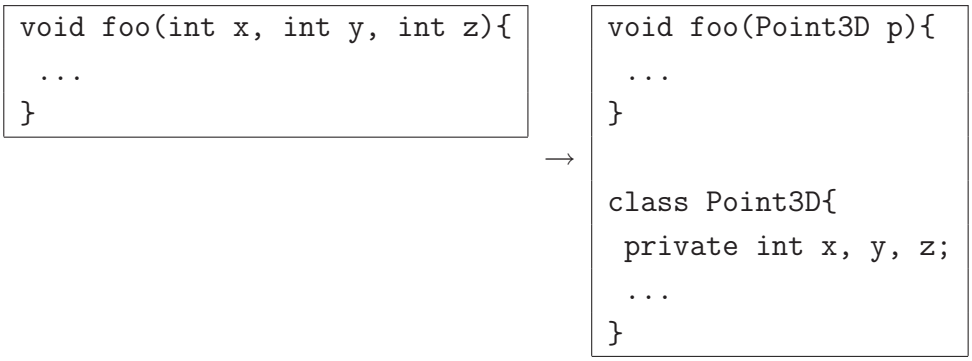


図 2.10: 引数オブジェクトの導入の例

2.3 ソフトウェアメトリクスによるリファクタリング支援

[2] 先行研究として、ソフトウェアメトリクスの値の変化から、リファクタリング操作によってソースコードがどの程度改善されたかを、測定するリファクタリング支援がある。

リファクタリング操作の前後におけるソフトウェアメトリクスの値の変化を、最適なリファクタリング操作を求める際に利用する。

2.3.1 ソフトウェアメトリクス

先行研究などで使用されているメトリクスの一部を紹介する。

- Method Lines Of Code
- Number Of Parameter
- McCabe Cyclomatic Complexity
- Number Of Using My Class Resource

Method Lines Of Code

Method Lines Of Code(以下 MLOC) は、コメントや空白を除いたメソッドの論理行数を表すメトリクスである。

Number Of Parameter

Number Of Parameter(以下 PAR) は、メソッドの引数の数を表すメトリクスである。

McCabe Cyclomatic Complexity

McCabe Cyclomatic Complexity(以下 MCC) は、メソッドの複雑度を表すメトリクスである。

MCC の基本値は 1 で、メソッド内で分岐が発生する度にその分岐の数だけ増える。図 2.11 では、基本値の 1 に加え if 文による分岐で 1、for 文による分岐で 1 とカウントされ MCC の値は 3 となる。

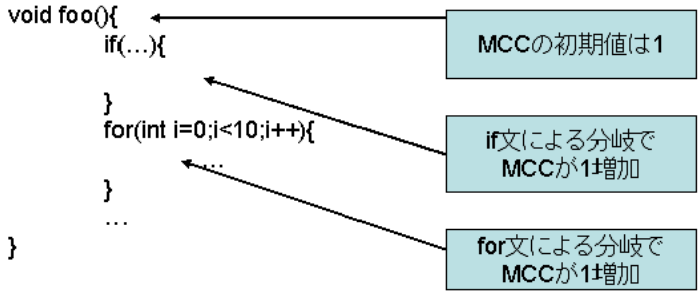


図 2.11: MCC の測定例

Number Of Using My Class Resource

Number Of Using My Class Resource(以下 MCR) は、メソッドが使用している自分の所属するクラスのフィールドおよびメソッドの数を表すメトリクスである。

自分の所属するクラスのフィールドおよびメソッドを 1 つも使用しておらず、且つ標準 API 以外のオブジェクトを利用している場合は-1 となる。

2.3.2 メトリクスの値の変化に基づくリファクタリング操作の評価の例

Long Method に対してリファクタリングを行った場合

図 2.12 は、BadSmell の一つである Long Method の状態であるメソッド foo に対して、メソッドの抽出というリファクタリング操作を行った際のコードの変化の様子を表したものであり、この時のメソッド foo のメトリクスの値の変化の結果が表 2.2 である。

メソッドの抽出というリファクタリング操作を行ったことにより、メソッド foo の Method Lines Of Code が 30 から 20 に減少してメソッドの行数が 10 行減り、更に McCabe Cyclomatic Complexity が 5 から 3 に減少したことによりメソッド foo の複雑度が減少したことから、メソッド foo に対してメソッドの抽出はメトリクスの値の減少に一定の効果がある。

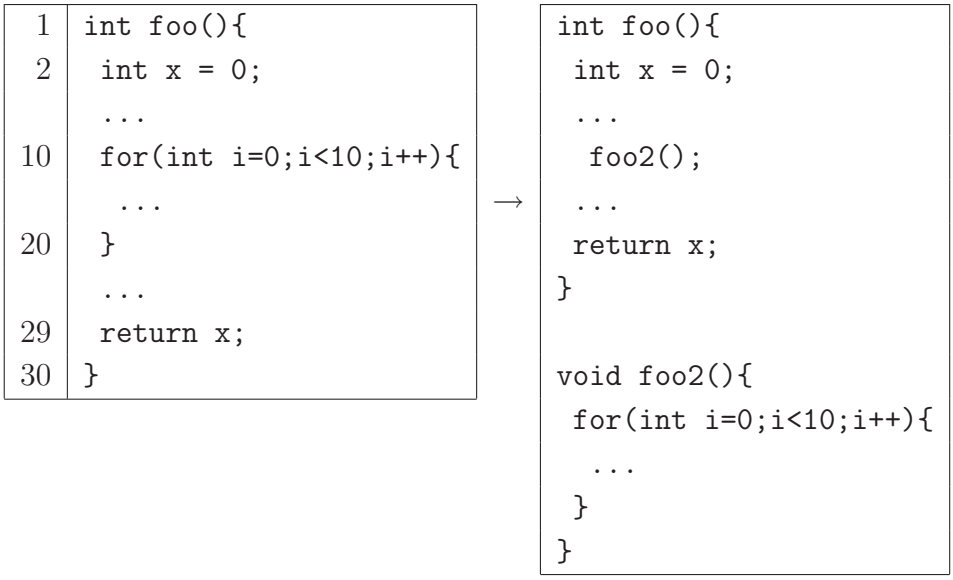


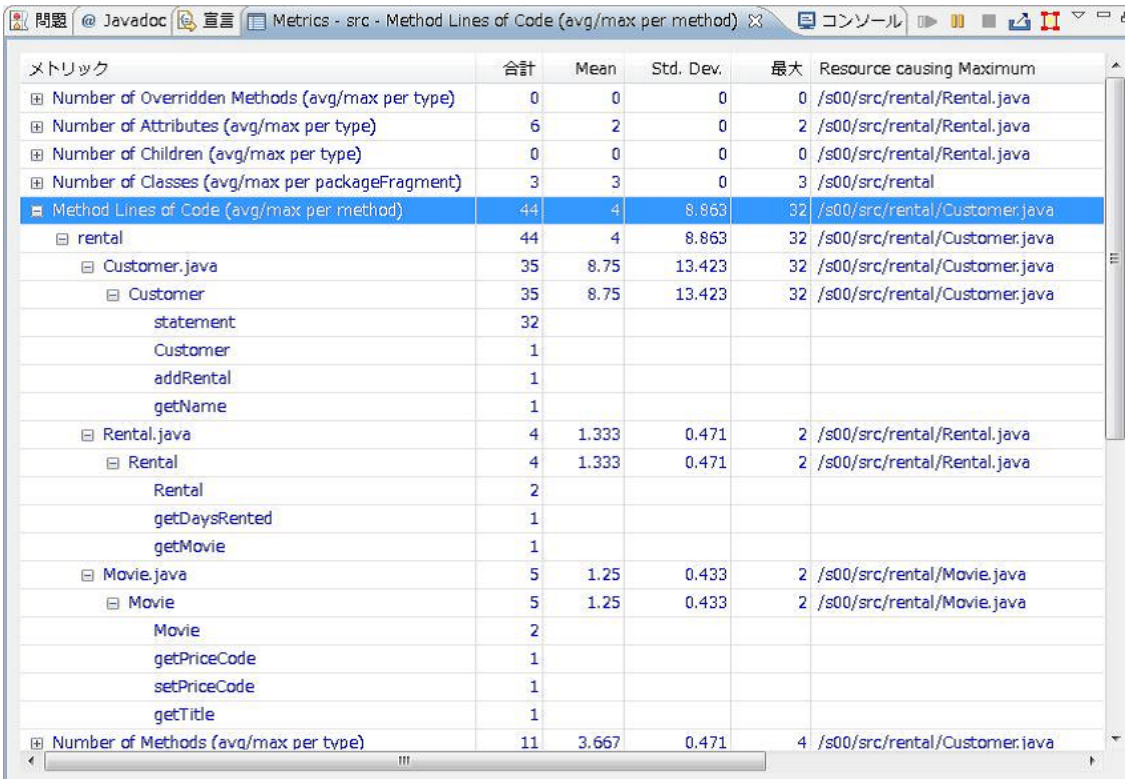
図 2.12: Long Method に対してメソッドの抽出を行った場合

表 2.2: Long Method に対してメソッドの抽出を行った場合のメソッド foo のメトリクスの変化

	Method Lines Of Code	McCabe Cyclomatic Complexity
リファクタリング前	30	5
リファクタリング後	20	3

2.3.3 メトリクス測定ツールの利用

リファクタリング操作前後のメトリクスの値の変化を求めるとき、図2.13 の様な [3] ソースコードのメトリクスの測定を自動的に行う測定ツールを使用することができる。



メトリック	合計	Mean	Std. Dev.	最大	Resource causing Maximum
Number of Overridden Methods (avg/max per type)	0	0	0	0	/s00/src/rental/Rental.java
Number of Attributes (avg/max per type)	6	2	0	2	/s00/src/rental/Rental.java
Number of Children (avg/max per type)	0	0	0	0	/s00/src/rental/Rental.java
Number of Classes (avg/max per packageFragment)	3	3	0	3	/s00/src/rental
Method Lines of Code (avg/max per method)	44	4	8.863	32	/s00/src/rental/Customer.java
rental	44	4	8.863	32	/s00/src/rental/Customer.java
Customer.java	35	8.75	13.423	32	/s00/src/rental/Customer.java
Customer	35	8.75	13.423	32	/s00/src/rental/Customer.java
statement	32				
Customer	1				
addRental	1				
getName	1				
Rental.java	4	1.333	0.471	2	/s00/src/rental/Rental.java
Rental	4	1.333	0.471	2	/s00/src/rental/Rental.java
Rental	2				
getDaysRented	1				
getMovie	1				
Movie.java	5	1.25	0.433	2	/s00/src/rental/Movie.java
Movie	5	1.25	0.433	2	/s00/src/rental/Movie.java
Movie	2				
getPriceCode	1				
setPriceCode	1				
getTitle	1				
Number of Methods (avg/max per type)	11	3.667	0.471	4	/s00/src/rental/Customer.java

図 2.13: メトリクス測定ツール

2.3.4 問題点

操作毎の改善効果の判断基準が曖昧

リファクタリング操作の前後において複数のメトリクスの値が変化するが、実際にどのメトリクスが BadSmell に、どの様に変化することが最適があるかが分かっていないため、結果として判断基準が曖昧であり、実際にはどのリファクタリング操作が最適であるかの評価が行えない。

第3章　メトリクスを利用したBadSmellの検出とその解決支援の自動化

3.1　メトリクスを利用したBadSmellの測定

これまで、BadSmellを解消する際のリファクタリング操作に対する評価として利用してきたソフトウェアメトリクスであるが、これをソースコード内のBadSmellの発見に利用することへの提案を行う。

BadSmellの発見への利用に変更することによって、曖昧であった改善効果の判断基準を明確なものにし、リファクタリング操作前後のメトリクスの値の変化の観察を簡単なものにする。

又、Fowlerが定義したリファクタリング操作と対応がとれているBadSmellをメトリクスを利用して発見する事により、メトリクスからソースコードに施すリファクタリング操作の候補まで求める事が可能となる。

3.1.1　BadSemllの分類

メトリクスとFowlerが定義したBadSmellを対応させる際、BadSmellをその特徴から大きく3つに分類した。分類することにより、これまで曖昧であったメトリクスからBadSmellを判断する際のメトリクスの値の基準値を明確にする。

- 可読性を損なうBadSmell
- コードの保守性を損なうBadSmell
- コードの重複に関与するBadSmell

以下にその詳細を記す

可読性を損なうBadSmell

可読性に関与するBadSmellは、メソッドの行数が長すぎる、メソッドの引数が多すぎる等の理由により、ソースコードが読み辛いという主観的な基準により判断されるのが特徴である。

そのため、判断に用いるメトリクスの境界値はリファクタリングを行うユーザの経験に基づく、あるいは会社等の組織で定めた値をリファクタリングを行う際に設定する必要がある。

コードの保守性を損なう BadSmell

コードの保守・拡張に関連する BadSmell は、解決することによりソフトウェアの保守・拡張の難易度を下げる BadSmell であり、これらの BadSmell が発生する原因にはコードの複雑度が関係しており、複雑度を測定するメトリクスによって判断を行う。

複雑度を示すメトリクスは学術的に判断基準が定められており、例えばソースコードの複雑度・テストケースの数を表すメトリクスであるサイクロマチック数がそれに該当する。

コードの重複に関与する BadSmell

コードの重複に関与する BadSmell は、その名の通りソースコード上で内容が重複している箇所を指す。

コードの重複に対応するメトリクスが現時点では見つかっていない。

3.1.2 メトリクスと BadSmell の対応

メトリクスから BadSmell を求めるため、BadSmell の特徴からそれに対応するメトリクスを求めた。

以下にその例を示す。

- Long Method と Method Lines Of Code
- Long Parameter List と Number Of Parameter
- Switch Statements と McCabe Cyclomatic Complexity
- Feature Envy と Number Of Using My Class Resource

Long Method と Method Lines Of Code

Long Method はメソッドのコード行数が多い事が特徴である。そこで、メソッドの行数を表すメトリクスである Method Lines Of Code(MLOC) を利用し、MLOC の値が大きいメソッドを BadSmell とすることにより、両者を対応させている。

Long Parameter List と Number Of Parameter

Long Parameter List はメソッドの引数の数が多すぎる事が特徴である。そこで、メソッドの引数の数を表すメトリクスである number of PARameter(PAR) を利用し、PAR の値が大きいメソッドを BadSmell とすることにより、両者を対応させている。

Switch Statements と McCabe Cyclomatic Complexity

Switch Statements はメソッドの複雑度が高くなることが特徴である。そこで、メソッドの複雑度を表すメトリクスである McCabe Cyclomatic Complexity(MCC) を利用する。[4]MCC 値が 10 以下であれば良い構造であるため、値が 10 を超えるメソッドを BadSmell とすることにより、両者を対応させることにする。

Feature Envy と Number Of Using My Class Resource

Feature Envy は、対象となるメソッドが所属するクラスのフィールドおよびメソッドをまったく使用しておらず、且つ標準 API を使用していないことが特徴であるため、Number Of Using My Class Resource(MCR) を利用し、MCR が-1 の時に BadSmell とすることにより、両者を対応させている。

3.1.3 リファクタリングの流れ

メトリクスを利用して BadSmell を求めることにより、支援は図 3 の様になる。

1. ソースコードのメトリクスを測定し、そこから BadSmell を求める。
2. BadSmell が存在する場合は、それを解決するリファクタリング操作の候補を求める。
3. 求めた操作候補の幾つかを BadSmell が消えるまで実行する。
4. 以上をすべての BadSmell が無くなるまで繰り返す。

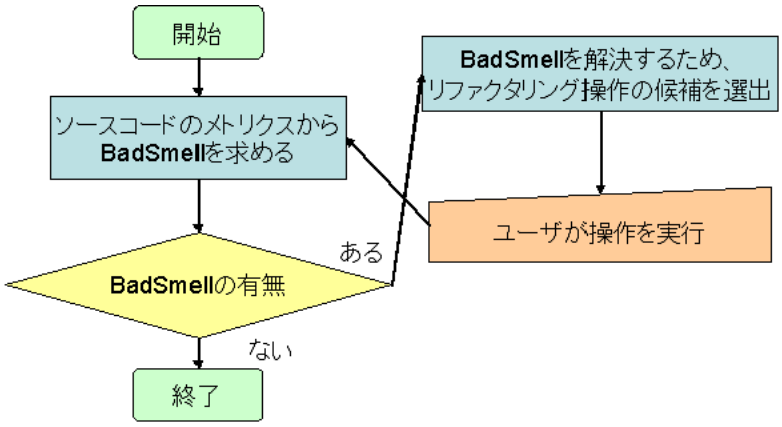


図 3.1: 支援の自動化によるリファクタリングの流れ

3.2 支援の自動化

リファクタリングによるオーバヘッドを減少させるため、その支援の一部の自動化を行う。

今回、自動化を行うのはソースコードから BadSmell を求める作業、および BadSmell を解決するためのリファクタリング操作候補の選出である。この 2 つを自動で行えるようにするため、以下の自動化を行う。

1. メトリクスの測定
2. メトリクスから BadSmell を求める作業
3. BadSmell から対応するリファクタリング操作を求める作業

3.2.1 メトリクスの測定の自動化

BadSmell を求める際に、メトリクスの値が必要となる。そこで、メトリクスの測定を自動化することにより、BadSmell の発見を自動的に行えるようにする。

3.2.2 メトリクスから BadSmell を求める作業の自動化

メトリクスと BadSmell との対応を元に、BadSmell を求める作業を自動化することにより、BadSmell の発見を自動化する。

3.2.3 BadSmell から対応するリファクタリング操作を求める作業の自動化

Fowler が提案する BadSmell を解決するためのリファクタリング操作候補を元に、解決のための操作候補を求める作業の自動化を行う。

第4章　自動化の環境の設計

4.1　リファクタリング支援ツール

第3章の支援の自動化を元に、リファクタリング支援を自動化したツールの設計を行う。

4.1.1　ツールが行う支援

ツールは以下の支援を行う。

- リファクタリングの開始・終了の判定の支援
- ソースコード内の BadSmell の表示
- BadSmell を解決するためのリファクタリング操作の候補の表示

リファクタリングの開始・終了の判定の支援

BadSmell の数や種類を視覚的に表示させることにより、ユーザがその有無による操作の開始・終了の判断を行いやすくしている。

ソースコード内の BadSmell の表示

ソースコード内の BadSmell を視覚的に表示させることにより、ユーザがその箇所を知る支援を行う

BadSmell を解決するためのリファクタリング操作の候補の表示

BadSmell を解決するためのリファクタリング操作の候補を表示させることにより、ユーザが解決のための操作を調べるための支援を行う。

4.1.2　リファクタリングの開始・終了の判定の流れ

リファクタリングの開始・終了の判定は図 4.1 の様に、ソースコード内の BadSmell によって行う。

ソースコード内に BadSmell がある場合はリファクタリングを開始し、BadSmell が無くなるまでリファクタリング操作を行う。

操作を繰り返しソースコード内に BadSmell が存在しない状態になれば、リファクタリングを終了してよい。

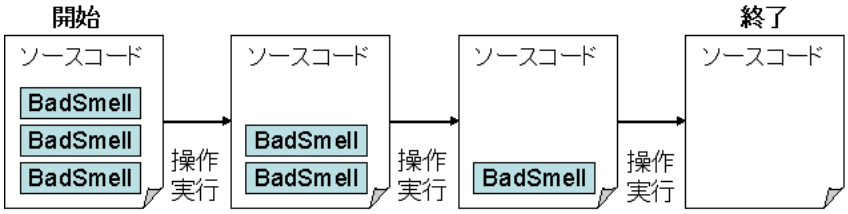


図 4.1: 開始・終了の判定

4.1.3　リファクタリング操作候補を求める流れ

リファクタリング操作候補は図 4.2 の様に、ソースコード内の BadSmell を発見することによって求める。

まず、BadSmell と対応しているメトリクスの値をまとめたメトリクス表から、ソースコードのメトリクスの値を求める。

次に、メトリクスと BadSmell の対応についてまとめた表を元に、先ほど求めたメトリクスの値から BadSmell を発見する。

そして、Fowler が提案した BadSmell とリファクタリング操作の対応をまとめた表から、BadSmell を解決するリファクタリング操作の候補を求める。

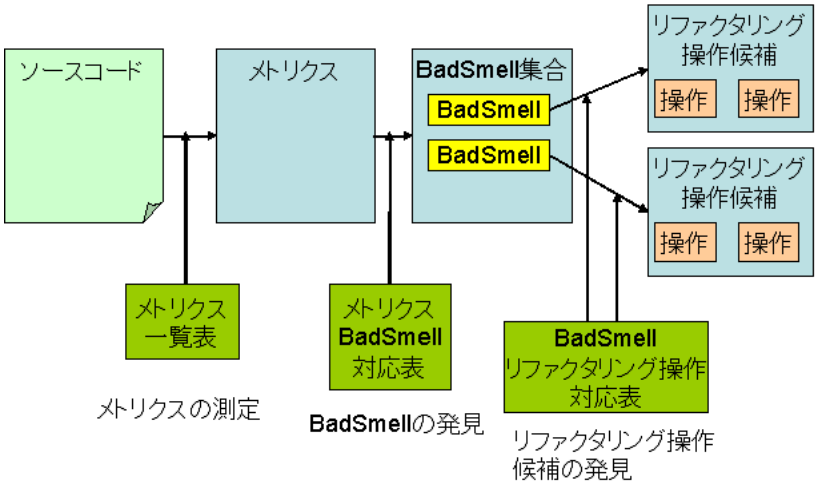


図 4.2: リファクタリング操作候補を求めるまでの流れ

メトリクス一覧表

BadSmell と対応していているメトリクスの種類の一覧である。
ツールでは、この表にあるメトリクスの値をソースコードから自動的に計算を行う。

メトリクス BadSmell 対応表

メトリクスと BadSmell との対応を記した表である。
この対応表を使用して、ツールはメトリクスの値から BadSmell を自動的に求める。

BadSmell リファクタリング操作対応表

Fowler の提案した BadSmell を解決するためのリファクタリング操作の一覧を記した表である。
この表から、ツールは BadSmell を解決するための操作候補を求め、ユーザに提示する。

4.2 既存のツールによるリファクタリング支援との違い

4.2.1 メトリクス測定ツール

メトリクス測定ツール、各種メトリクスの値は自動的に測定できるが、その値から BadSmell は自動化されていない。
また、メトリクスによるリファクタリング操作への評価による支援を行う際には、ユーザが操作前後でメトリクスの値を調べる必要がある。

4.2.2 Eclipse のリファクタリング機能

Eclipse のリファクタリング機能は、ソースコードにリファクタリング操作の適応を自動で行う機能であり、今回の BadSmell とそれを解決するためのリファクタリング操作の候補選出とは直接関係は無い。

しかし、今回のツールで求めたリファクタリング操作を実行する際にこの機能を利用することは有用である。

4.3 ツールの仕様

4.3.1 BadSmell 判定のメトリクス基準値の設定

可読性に関与する BadSmell をメトリクスを利用して発見する場合は、その基準値を予め設定する必要がある。

可読性に関与する BadSmell を判断するための基準値を、BadSmell を求める処理を行う前にリファクタリングを行うユーザに設定させる。

4.3.2 BadSmell 一覧の表示

ツールにより発見した BadSmell を一覧表示させ、ユーザに BadSmell の有無を確認させる機能を設定する。

4.3.3 BadSmell のコード上の位置の表示

ツールにより BadSmell の位置をユーザに視覚的に表示させる。

第5章　自動化の環境の実装

5.1　Eclipseプラグインによる実装

支援の自動化をEclipseプラグインとして実装し、リファクタリング支援ツールとする。

5.1.1　Eclipseプラグインにする理由

自動化の環境を実装するに当たってEclipseプラグインにするには、以下の理由からである。

- ソースコードのプロジェクト管理を利用できる。
- エディタを利用してBadSmellの位置を視覚的に表現することができる。

5.1.2　プロジェクト単位でのBadSmellの発見

Eclipseが提供するプロジェクトを利用することにより、ソースコードのファイル単体ではなくソフトウェア全体のソースコードを扱えるようにする。全体を扱うことで、複数のファイル間のソースコードに関係するBadSmellを扱うことができる。

5.1.3　BadSmellの表現

発見したBadSmellは、Eclipseが提供するソースコードの警告を登録するProblem Viewに登録する。

登録することにより、ユーザはProblem ViewからBadSmellの有無を、エディタからソースコード上のBadSmellの位置を確認することができる。

5.1.4　BadSmellを解消するリファクタリング操作候補の表現

Eclipseの提供する警告を改善するための操作を登録するQuick Fixを利用し、BadSmellの警告に対してそれを解決するリファクタリング操作の候補をQuick Fixとして登録することにより、ユーザに操作候補を伝える。

5.1.5 全体構成図

実装したツールを使用して行う支援の全体構成図は図 5.1 となる。

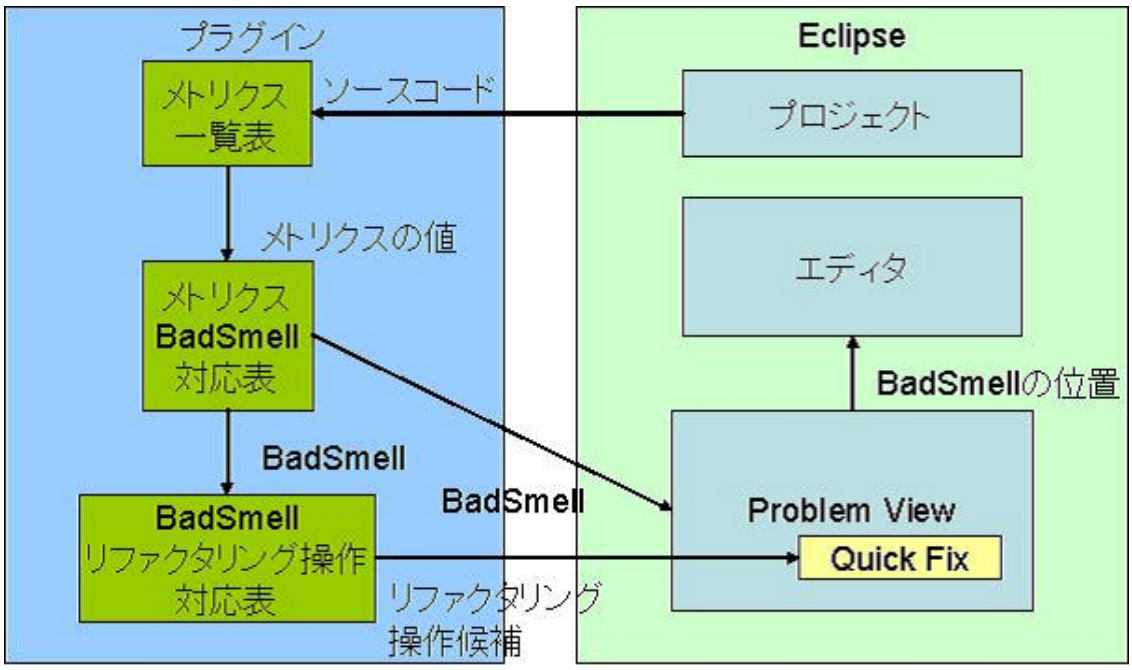


図 5.1: ツールによる支援の全体構成図

5.2 各部品の実装

5.2.1 メトリクス表の実装

ソースコードを読み込み、BadSmell に対応しているメトリクスを測定を行う。

メトリクスの種類毎に測定するための部品を作製し、BadSmell を求める際に対応するメトリクスを測定する部品を使用する。

測定された結果が表立ってユーザに表示されることは無い。

5.2.2 メトリクス BadSmell 対応表の実装

BadSmell の存在とその箇所を確認するため、それに対応したメトリクスの値をメトリクス表から求める部品を作製する。

部品は BadSmell 毎に作製し、それを使用して各 BadSmell を求める機能を Eclipse のビルダーに追加することにより、ソースコードの更新毎に BadSmell の有無の確認を行うようにする。メトリクスの値から BadSmell の存在と箇所が確認された場合、Eclipse の

Problem View にそれを登録し、Problem View からは BadSmell の有無の確認を、エディタ上では BadSmell の箇所を視覚的に確認することができるようにする。

5.2.3 BadSmell リファクタリング操作対応表の実装

Problem View に BadSmell を登録する際に、Eclipse の Quick Fix に BadSmell を解決するためのリファクタリング操作の名前を登録することにより、ユーザはそれを解決するためのリファクタリング操作の候補を確認することができる

5.3 ツールによるリファクタリングの手順

5.3.1 BadSmell の発見

ソースコード内の BadSmell は、図 5.2 の様に Eclipse の ProblemView へと登録される。

ユーザは、Problem View からソースコード内の BadSmell の有無を視覚的に確認することができる。更にリスト内の BadSmell をダブルクリックすることにより、該当部分のソースコードをエディタで開くことができ、BadSmell のソースコード上の位置を確認することができる。

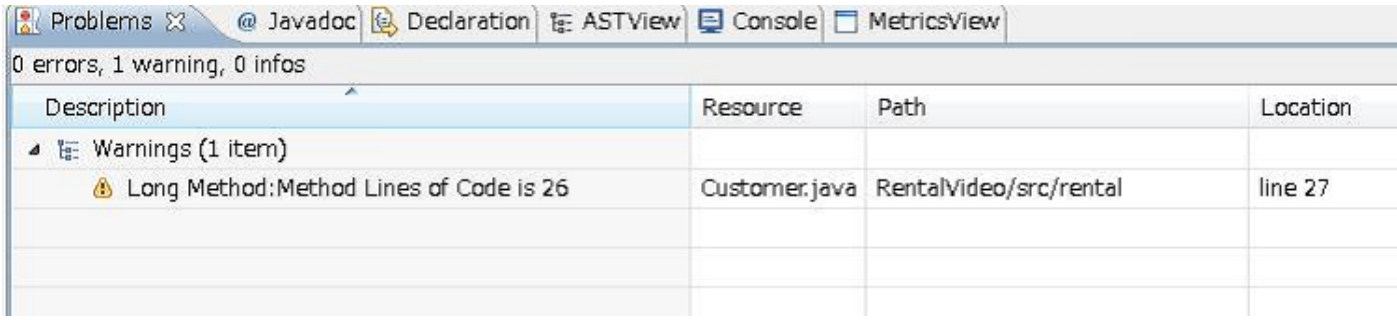


図 5.2: ProblemView への BadSmell の登録

5.3.2 BadSmell の確認

BadSmell を ProblemView に登録されることにより、図 5.3 の様に Ecilpse のエディタで BadSmell のある箇所がマーカで表示される。

ユーザは、マーカから BadSmell の具体的な位置を視覚的に確認することができる

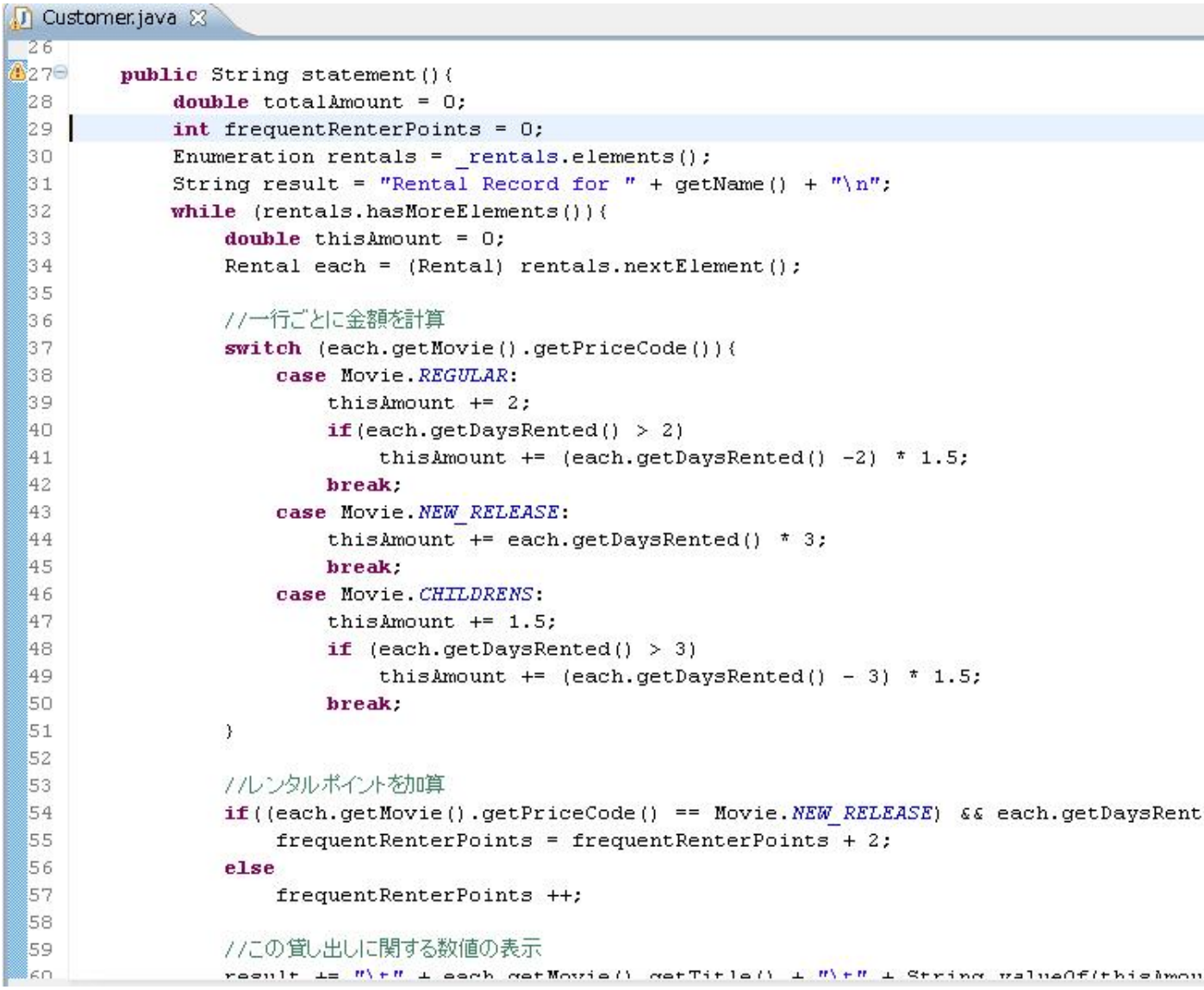


図 5.3: エディタ部分に対しての BadSmell の視覚的表現

5.3.3 BadSmell を解消するためのリファクタリング操作候補の表示

Problem View へと登録された BadSmell に対する Quick Fix にリファクタリング操作候補の名前を登録する。

ユーザは警告として登録された BadSmel の Quick Fix を呼び出すことにより、図 5.4 の様にリファクタリング操作候補の一覧が表示され、BadSmell を解決するための操作候補を知ることができる。

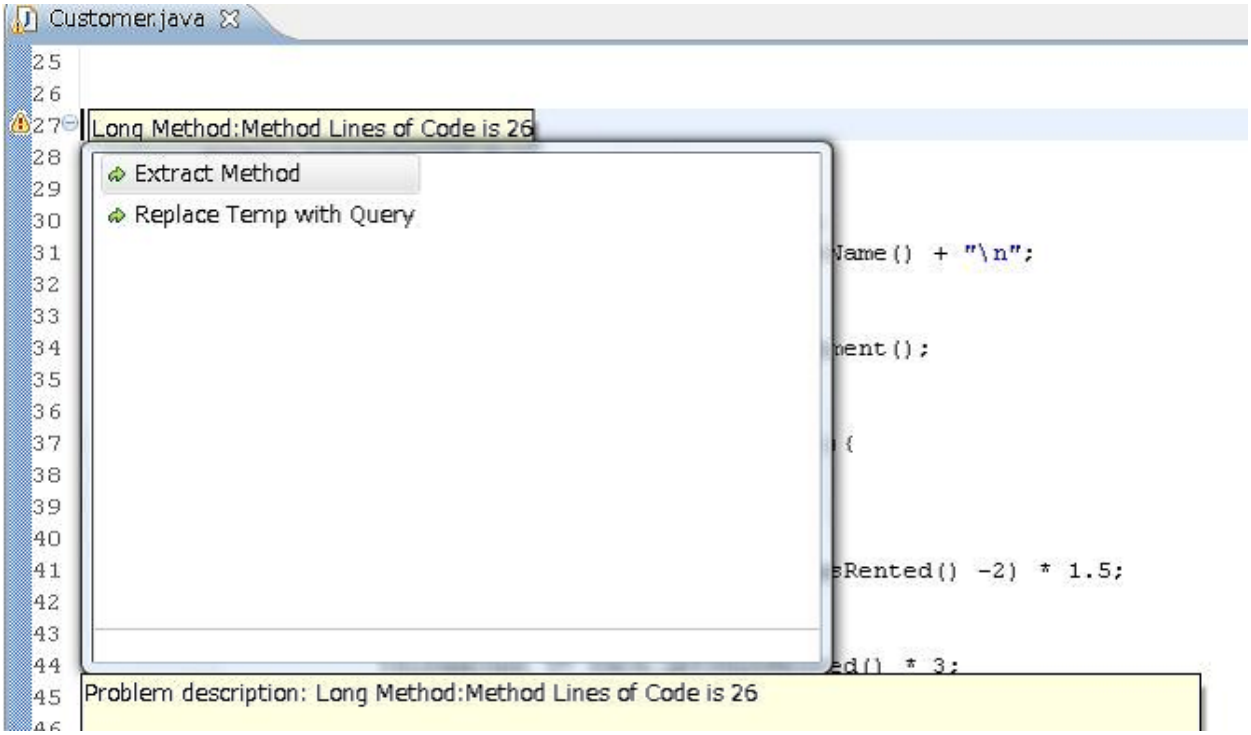


図 5.4: BadSmell を解消するためのリファクタリング操作候補の表示

5.3.4 リファクタリング操作による BadSmell の解消

リファクタリング操作により BadSmell が解消されると、図 5.5 の様に BadSmell のマーカが消える。

ユーザは、このことから BadSmell が解消されたことを視覚的に確認することができる。

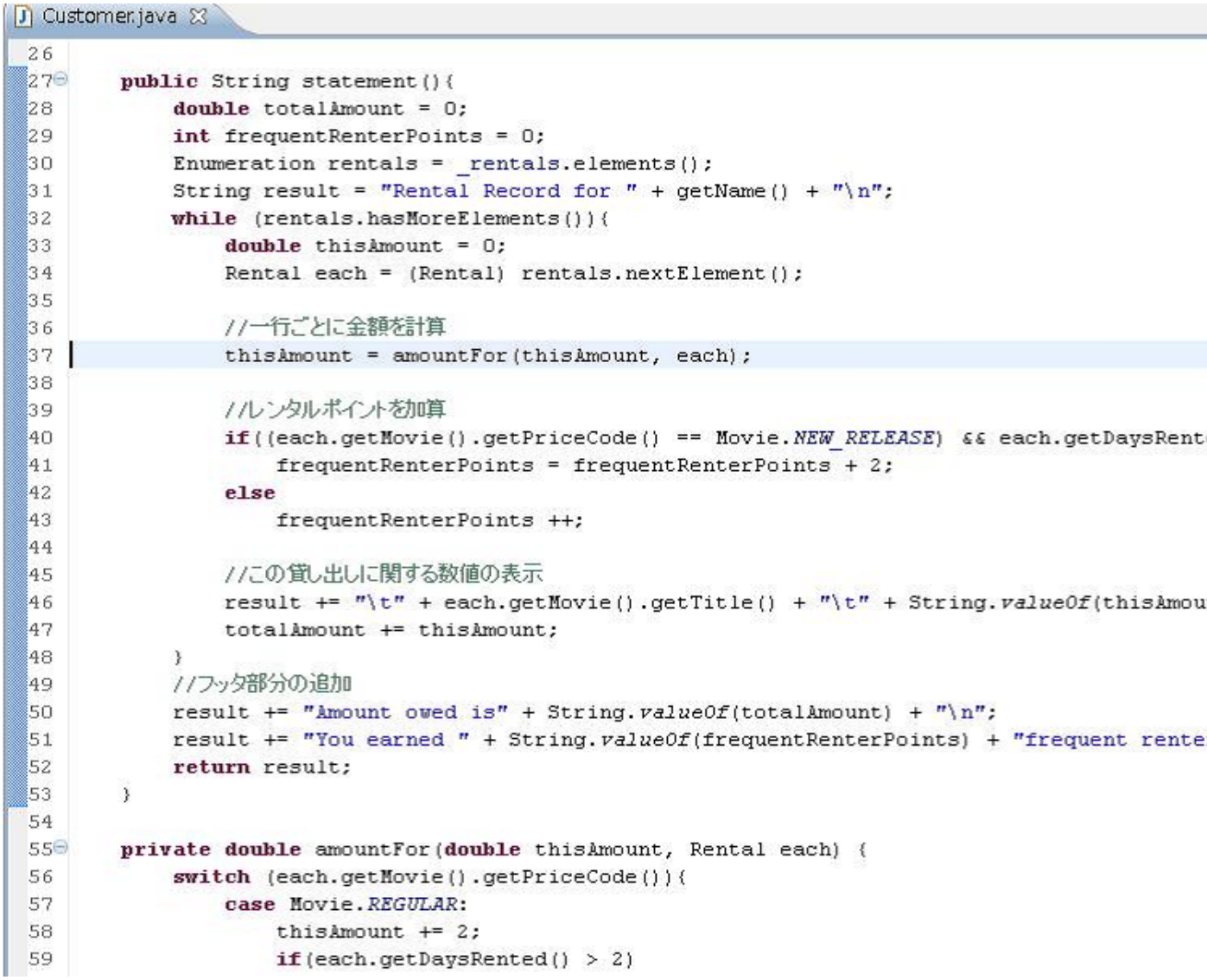


図 5.5: BadSmell の解消

5.3.5 リファクタリングの終了

ProblemView を確認し、BadSmell がなければリファクタリングの終了となる。

第6章　支援環境の適用実験

6.1　ツールによるリファクタリングの適用例

Eclipse のプラグインとして実装した、メトリクスの測定によるリファクタリング支援の自動化を実際に行い、その有用性の確認を行う。

6.1.1　対象とするソースコード

Customer.java

```
package rental;
import java.util.Enumeration;
import java.util.Vector;

public class Customer {
    private String _name;
    private Vector _rentals = new Vector();
    public Customer (String name){
        _name = name;
    }
    public void addRental(Rental arg){
        _rentals.addElement(arg);
    }
    public String getName(){
        return _name;
    }
    public String statement(){
        double totalAmount = 0;
        int frequentRenterPoints = 0;
        Enumeration rentals = _rentals.elements();
        String result = "Rental Record for " + getName() + "\n";
        while (rentals.hasMoreElements()){
```

```
            double thisAmount = 0;
            Rental each = (Rental) rentals.nextElement();
            switch (each.getMovie().getPriceCode()){
                case Movie.REGULAR:
                    thisAmount += 2;
                    if(each.getDaysRented() > 2)
                        thisAmount += (each.getDaysRented() -2) * 1.5;
                    break;
                case Movie.NEW_RELEASE:
                    thisAmount += each.getDaysRented() * 3;
                    break;
                case Movie.CHILDRENS:
                    thisAmount += 1.5;
                    if (each.getDaysRented() > 3)
                        thisAmount += (each.getDaysRented() - 3) * 1.5;
                    break;
            }
            if((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) && each.getDaysR
                frequentRenterPoints = frequentRenterPoints + 2;
            else
                frequentRenterPoints ++;
            result += "\t" + each.getMovie().getTitle() + "\t" + String.valueOf(thisA
            totalAmount += thisAmount;
        }
        result += "Amount owed is" + String.valueOf(totalAmount) + "\n";
        result += "You earned " + String.valueOf(frequentRenterPoints) + "frequent re
        return result;
    }
}
```

Rental.java

```
package rental;

public class Rental {
    private Movie _movie;
    private int _daysRented;
    public Rental(Movie movie, int daysRented){
        _movie = movie;
```

```
        _daysRented = daysRented;
    }
    public int getDaysRented(){
        return _daysRented;
    }
    public Movie getMovie(){
        return _movie;
    }
}
```

Movie.java

```
package rental;

public class Movie {
     public static final int CHILDRENS = 2;
     public static final int NEW_RELEASE = 1;
     public static final int REGULAR = 0;
     private String _title;
     private int _priceCode;
     public Movie(String title, int priceCode){
         _title = title;
         _priceCode = priceCode;
     }
     public int getPriceCode(){
         return _priceCode;
     }
     public void setPriceCode(int arg){
         _priceCode = arg;
     }
     public String getTitle(){
         return _title;
     }
}
```

6.1.2 実験対象のコードに含まれる BadSmell

今回の実験対象のコードに含まれる BadSmell は表 6.1 のとおりである。

表 6.1: 今回のコード上存在する BadSmell

BadSmell	発生箇所
Long Method	Constemr クラス statement メソッド
Feature Envy(1)	Constemr クラス statement メソッド
Feature Envy(2)	Constemr クラス statement メソッド
Switch Statements	Constemr クラス statement メソッド

6.1.3 ツールによる BadSmell の発見

Long Method の基準値は 15 行以上とする。

1. Long Method(26) に対してメソッドの抽出を行う
2. Feature Envy に対してメソッドの移動を行う
3. Long Method(15) に対してメソッドの抽出を行う
4. Feature Envy に対してメソッドの移動を行う

初期状態

図 6.1 の Problem View から、Customer クラスの statement メソッドが 26 行であるゆえに Long Method であることが確認でき、このソースコードに対してリファクタリングが必要であるとわかる。

又、図 6.2 のエディタのマーカから、Long Method である箇所のを位置を視覚的に確認することができる。

Long Method を解決するためのリファクタリング操作を確認したところ、図 6.3 の結果となり、メソッドの抽出 (Extract Method) と一時変数の置き換え (Replace Temp With Query) で解決できる事がわかる。

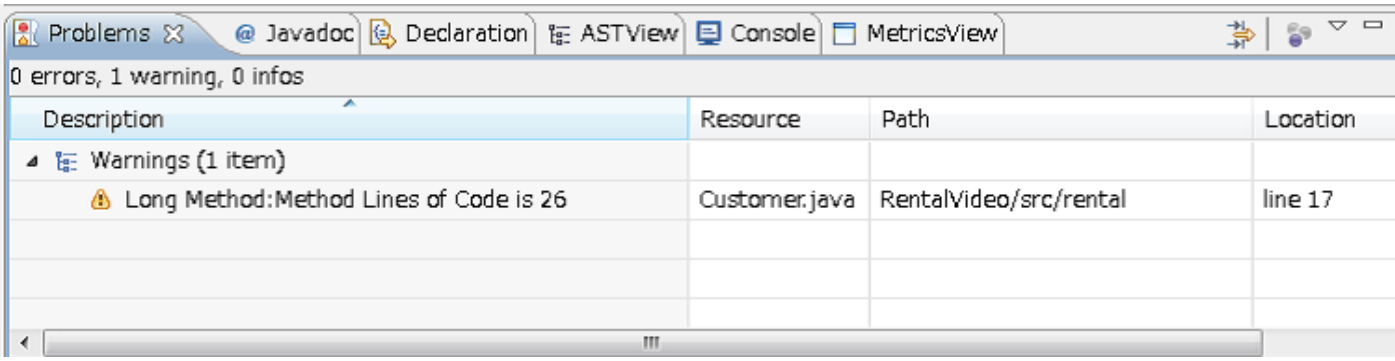


図 6.1: Problem View

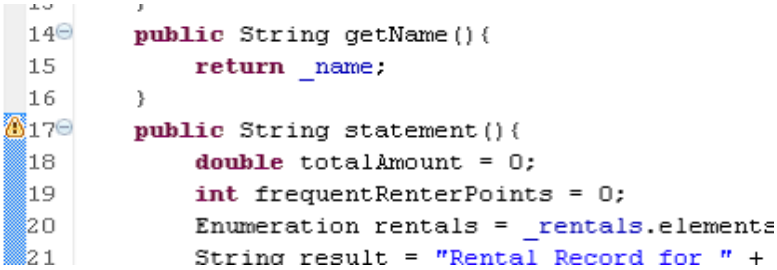


図 6.2: エディタによる Long Method の箇所の表現

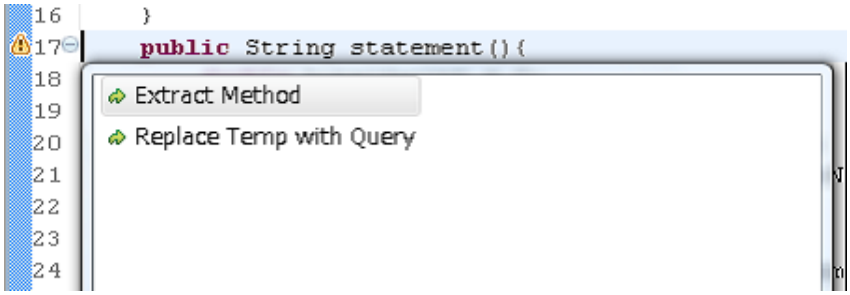


図 6.3: Long Method を解決するリファクタリング操作

Long Method に対してメソッドの抽出を行う

Long Method を解決するため、6.4 のとおりメソッドの抽出を行い、statement メソッドの一部を amountFor メソッドとして抽出した。

その結果、図 6.5 の Problem View から statement メソッドは 15 行まで行数が少なくなったものの、依然として Long Method であることが確認できる。

更に、抽出したメソッドである amountFor が Feature Envy であることが確認でき、先ほどのメソッドの抽出により statement メソッドに潜んでいた BadSmell を発見することができた。

新たに発見した Feature Envy を解決するためのリファクタリングを確認したところ、図 6.3 の結果となり、メソッドの移動 (Move Method) で解決できることがわかる。

```
public String statement(){
  ...
  while (rentals.hasMoreElements()){
    double thisAmount = 0;
    Rental each = (Rental)ren ...;
    switch (each.getMovie() ...){
      case Movie.REGULAR:
        ...
    }
    ...
  }
  ...
}
```

→

```
public String statement(){
  ...
  while (rentals.hasMoreElements()){
    Rental each = (Rental) rens ...
    double thisAmount = amountFor(each);
    ...
  }
  ...
}
public double amountFor(Rental each) {
  double thisAmount = 0;
  switch (each.getMovie() ...)){
    case Movie.REGULAR:
      ...
  }
  return thisAmount;
}
```

図 6.4: amountFor メソッドを抽出

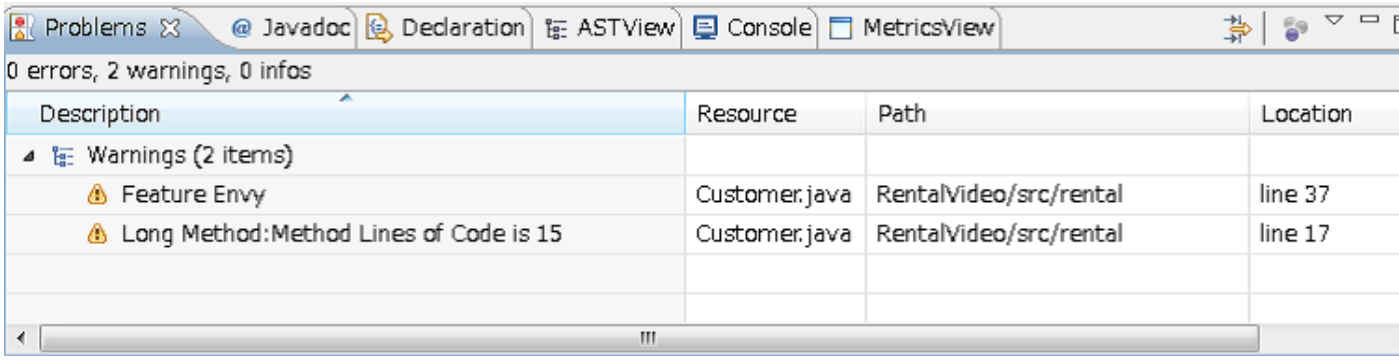


図 6.5: Problem View

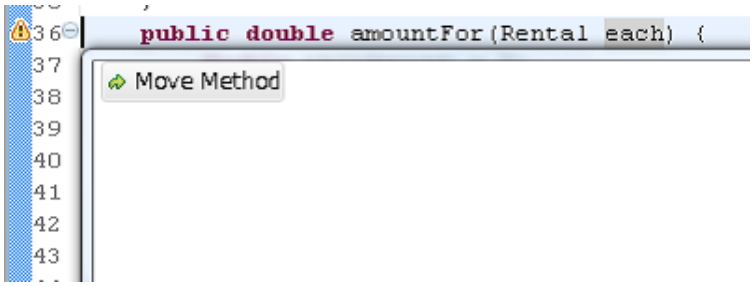


図 6.6: Feature Envy を解決するリファクタリング操作

Feature Envy に対してメソッドの移動を行う

Feature Envy を解決するため、6.7 のとおりメソッドのを移動を行い、amountFor メソッドを Costomer クラスから Rental クラスへと移動した

その結果、図 6.8 の Problem View から Feature Envy が消えたことが確認でき、リファクタリング操作により BadSmell が消滅したことがわかる。

しかし、未だ Long Method の方は消滅していないため、引き続きリファクタリングを行う必要がある。

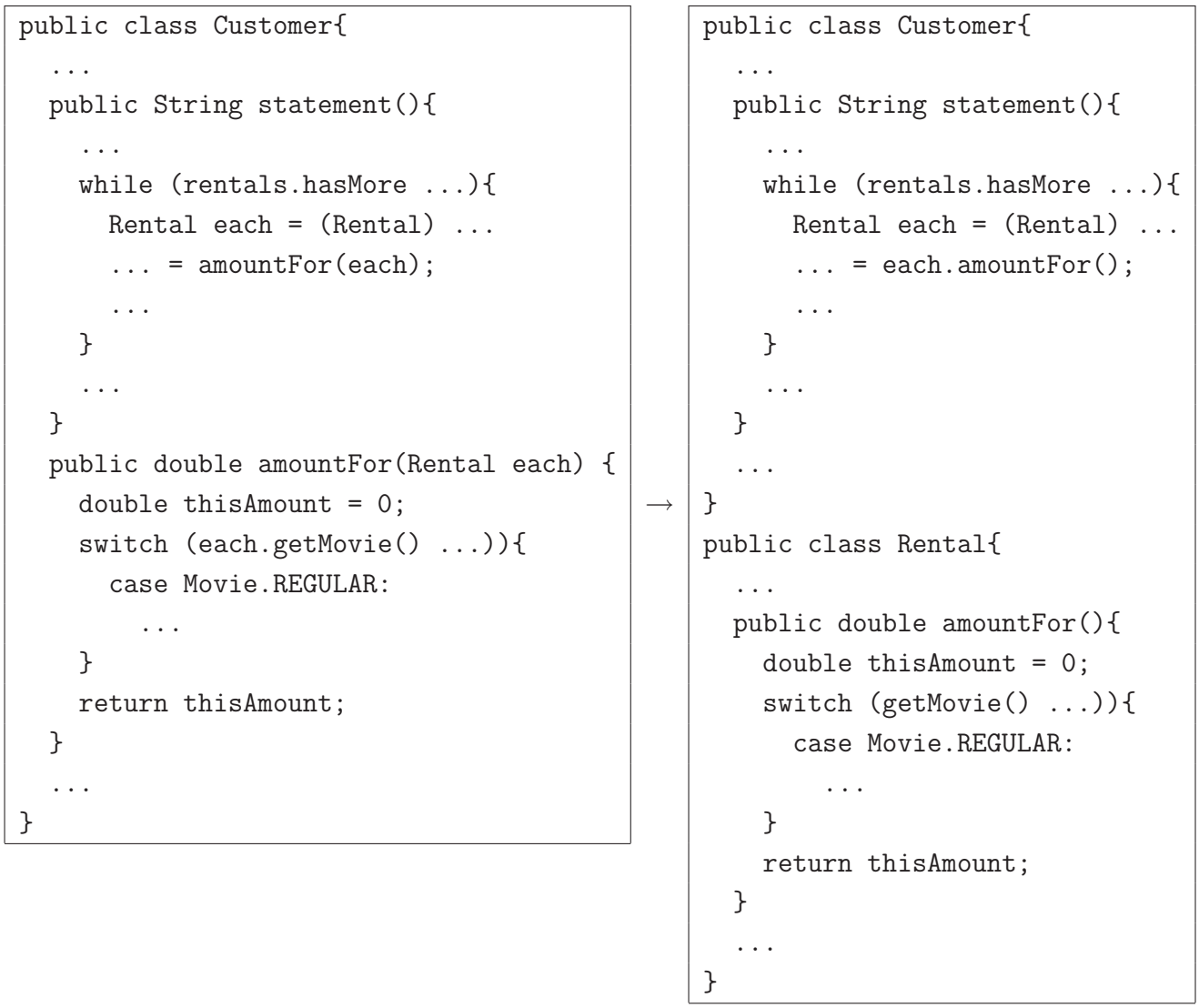


図 6.7: amountFor メソッドの移動

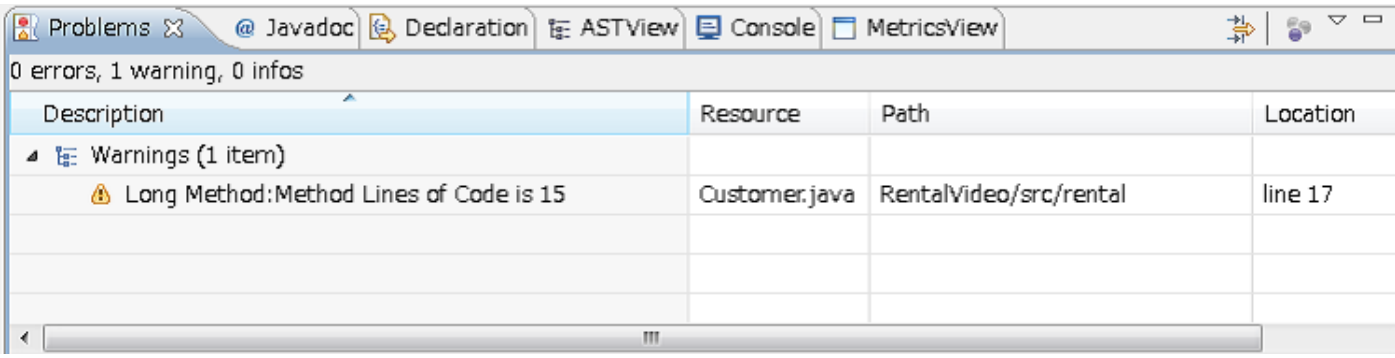


図 6.8: Problem View

Long Method に対してメソッドの抽出を行う

Long Method を解決するため、6.9 のとおりメソッドの抽出を行い、statement メソッドの一部を getFrequentRenterPoints メソッドとして抽出した。

その結果、図 6.10 の Problem View から Long Method が消えたことが確認でき、リファクタリング操作により BadSmell が消滅したことがわかる。

しかしながら、抽出したメソッド getFrequentRenterPoints が Feature Envy であるため、これを解消するためリファクタリングを続けなければならない。

```
public class Customer{
  ...
  public String statement(){
    ...
    while (rentals.hasMore ...){
      ...
      if((each.getMovie() ...

        frequentRenterPoints = ...
      else
        frequentRenterPoints++
      ...
    }
    ...
  }
  ...
}
```

→

```
public class Customer{
  ...
  public String statement(){
    ...
    while (rentals.hasMore ...){
      ...
      frequentRenterPoints =
         getFrequentRenterPoints(...
      ...
    }
    ...
  }
  ...
  public int getFrequentRenterPoints(...{
    if((each.getMovie() ...
      frequentRenterPoints = ...
    else
      frequentRenterPoints++
    return frequentRenterPoints;
  }
  ...
}
```

図 6.9: amountFor メソッドの移動

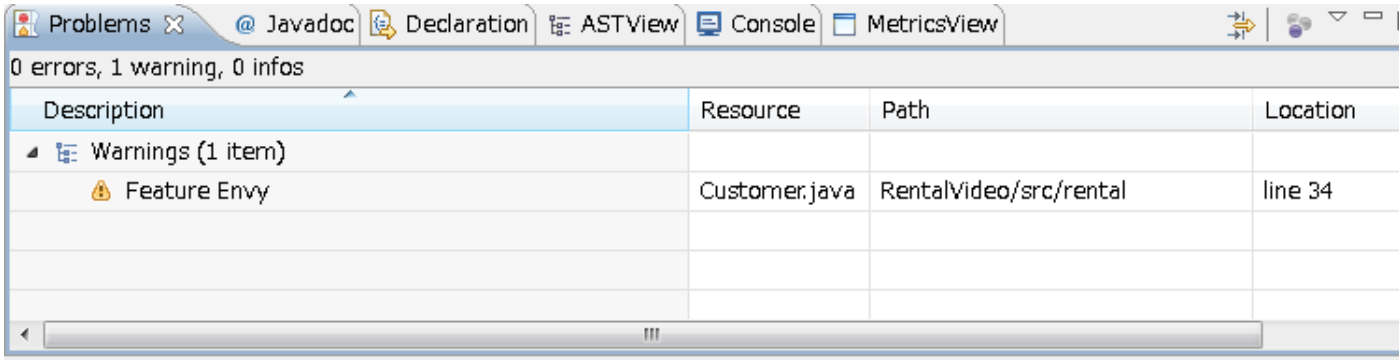


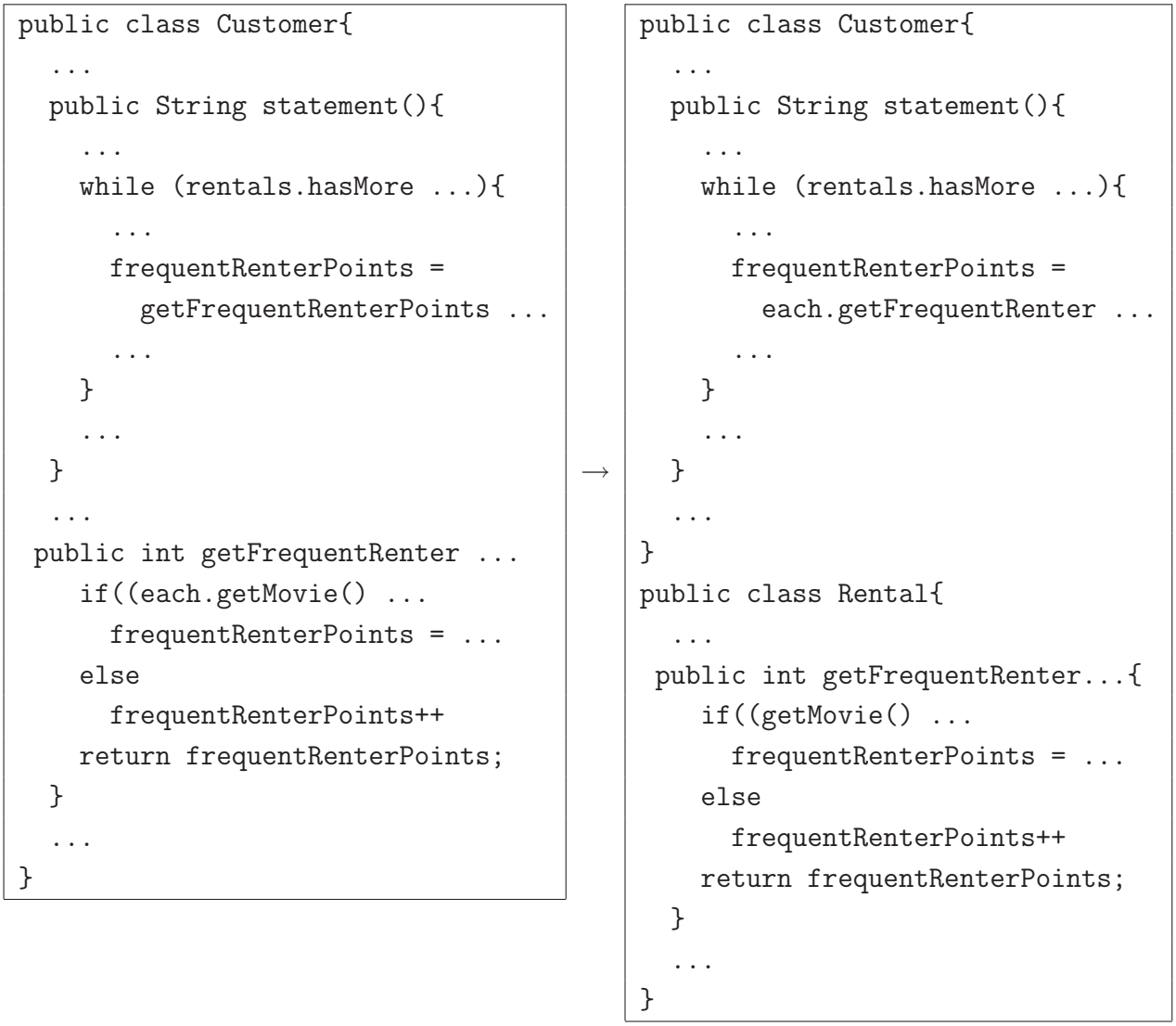
図 6.10: Problem View

Feature Envy に対してメソッドの移動を行う

Feature Envy を解決するため、6.11 のとおりメソッドのを移動を行い、getFrequentRenterPoints メソッドを Costomer クラスから Rental クラスへと移動した

その結果、図 6.12 の Problem View から Feature Envy が消えたことが確認でき、リファクタリング操作により BadSmell が消滅したことがわかる。

又、これ以上 BadSmell が見つからないことから、リファクタリングを終了する。



```
public class Customer{
  ...
  public String statement(){
    ...
    while (rentals.hasMore ...){
      ...
      frequentRenterPoints =
        getFrequentRenterPoints ...
      ...
    }
    ...
  }
  ...
 public int getFrequentRenter ...
    if((each.getMovie() ...
      frequentRenterPoints = ...
    else
      frequentRenterPoints++
    return frequentRenterPoints;
  }
  ...
}
```

→

```
public class Customer{
  ...
  public String statement(){
    ...
    while (rentals.hasMore ...){
      ...
      frequentRenterPoints =
        each.getFrequentRenter ...
      ...
    }
    ...
  }
  ...
}
public class Rental{
  ...
 public int getFrequentRenter...{
    if((getMovie() ...
      frequentRenterPoints = ...
    else
      frequentRenterPoints++
    return frequentRenterPoints;
  }
  ...
}
```

図 6.11: amountFor メソッドの移動

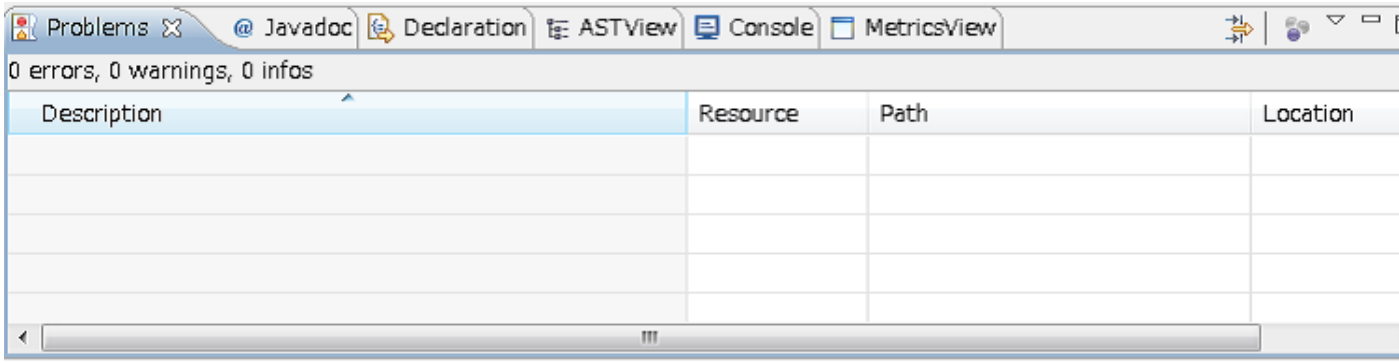


図 6.12: Problem View

6.1.4 今回発見できた BadSmell

今回のリファクタリングで発見した BadSmell は、表 6.2 の通り、Long Method 1 つと Feature Envy 2 つの合計 3 つである。

Long Method の発見

Long Method 発見は、メソッドが長すぎることにより可読性が損なわれる行数を予め設定することにより、メソッドの行数を司るメトリクスの値によって自明的に BadSmell の存在が明らかとなり発見することができた。

Feature Envy の発見

Feature Envy 2 つの発見は、Long Method であるメソッドからその一部を抽出することにより、Feature Envy の部分が明らかとなり、その特徴を司るメトリクスである Number Of Using My Class Resource の値が-1 になることにより、初めて発見することができた。

メソッド内のコードの一部が Feature Envy の場合、それをメトリクスで発見するには、該当部分のコードを抽出して新たなメソッドを作る必要がある。

今回の様に Long Method を解決する際に抽出したメソッドが偶然 Feature Envy の部分であった場合はメトリクスにより発見することができるが、今回の例で Long Method である行数の上限が 16 行以上であったり抽出する箇所が違った場合であれば、2 つめの Feature Envy を発見することはできなかった。

Switch Statements の発見

今回発見することのできなかった Switch Statements は、該当するメトリクスの値が小さく BadSmell であるかの判別を行うことができなかった。

判別できなかった原因は、この BadSmell によるコードの保守・拡張性の低下の効果が低いことから、メトリクスの値が小さかったことによると考えられる。

表 6.2: 今回の支援により発見した BadSmell

BadSmell	発生箇所	発見できたか
Long Method	Constemr クラス statement メソッド	○
Feature Envy(1)	Constemr クラス statement メソッド	○
Feature Envy(2)	Constemr クラス statement メソッド	○
Switch Statements	Constemr クラス statement メソッド	×

第7章　支援の評価

7.1　既存のリファクタリング支援との比較

7.1.1　従来のメトリクスを利用した支援との比較

メトリクスを利用したリファクタリング支援ではリファクタリング後の BadSmell の改善度を計る目的として利用されたが、今回の支援では BadSmell の有無を調べる目的で利用している。

リファクタリングの終了判定を、BadSmell の改善度から Fowler の提案する BadSmell の有無にしたことにより、これまで曖昧であったメトリクスの値の基準を明確化させることにし、その結果としてリファクタリングの開始および終了の評価の自動化が実現可能となった。

7.1.2　既存のツールを使用したメトリクスを利用した支援との比較

既存のツールを使用したメトリクスによる支援では、リファクタリング操作後の評価を行うため、リファクタリングを行うユーザがソースコードのメトリクスの値の変化を逐一観察する必要があった。

しかし、今回のツールの場合は、メトリクスの変化を観察する必要がなく、変わりに BadSmell がソースコード上から消滅したかどうかを確認するだけで、終了の判断を行うことができるため、ユーザのメトリクスを調べるという負担が無くなった。

7.1.3　リファクタリング操作候補選出の支援

これまで、BadSmell を解決するためのリファクタリング操作候補を選出するには、BadSmell の定義者が提案した操作候補を調べる、あるいはそれを覚える必要があった。しかし、今回のツールで操作候補を自動的に選出することにより、操作候補を調べる手間が無くなった。

7.2 メトリクスを利用した支援の自動化の評価

7.2.1 BadSmell 発見の評価

メトリクスを利用した BadSmell の発見では、コードの保守・拡張を行う際の支障が小さい BadSmell は、正常なコードとメトリクス値発見することができなかった。

このことから、メトリクスを利用した支援は、コードの保守・拡張を行う際の支障が大きいもの箇所を探すことには向いているが、支障が小さいものを探すには向いていないと評価できる。

第8章　結論

8.1　メトリクスを利用したリファクタリング支援の限界

第5章で行ったツールを利用したリファクタリングから、メトリクスを利用したリファクタリング支援では、BadSmellとしての特徴が顕著であり、コードの保守・拡張に支障がでるものを見つけるには向いているが、現段階では保守・拡張に支障がでないものの、今後のコードの拡張において支障がでるものになる恐れのある箇所を見つけることには向いていない。

そのため、この支援はコードを保守・拡張性を大きく損なう緊急性の高いBadSmellを改善する目的での利用が主になる。

8.2　まとめ

今回、ソフトウェアメトリクスを利用することより、Fowlerの提案するリファクタリング支援の一環であるBadSmellの発見とその解決のためのリファクタリング操作候補の選出の自動化を行った。

BadSmellの発見を自動化するにあたって、メトリクスを利用したリファクタリング支援を見直し、メトリクスの利用をリファクタリング操作によって変化したソースコードの評価から、ソースコード内のBadSmellの発見に利用することにした。

BadSmellの発見の自動化では、リファクタリングの開始・終了の判断において、ソースコードからBadSmellを発見する作業の支援を行い、メトリクスによるリファクタリング支援でのBadSmellの発見が容易になった上、これまで行っていたメトリクスの値の変化の観察の必要が無くなった。

BadSmell解決のためのリファクタリング操作候補の選出の自動化では、ユーザが操作候補を調べる手間が無くなった。

以上のことから、本研究における支援の自動化をりようすることにより、作業時間を短縮が可能となり、ソフトウェア開発の過程においてリファクタリングを行う際、コストの削減が期待できる。

8.3　今後の課題

今回のリファクタリング支援において、リファクタリング操作候補の選出の自動化を行ったが、選出した操作に対する支援を実装して選出した操作と対応を図ることにより更なる支援を期待できる。

謝辞

本研究を行うにあたり終始御指導賜りました鈴木正人准教授に心より深く感謝申し上げます。

本研究を行うにあたり大変有益な御助言をいただきました落水浩一郎教授に心より感謝申し上げます。

また研究を進めるに当たり貴重な意見をいただきました研究室の皆様に心より感謝いたします。

関連図書

[1] Fowler,M.,(児玉公信, 友野晶夫, 平澤章, 梅沢真史), リファクタリング, ピアソンエデュケーション, 2000.

[2] 五嶋 秀章, 複数のソフトウェアメトリクスを用いたリファクタリング支援手法に関する研究, 北陸先端科学技術大学院大学情報科学研究科, 2007.

[3] Frank Sauer, Eclipse Metrics Plugin（Frank Sauer）, http://metrics.sourceforge.net/.

[4] Capers Jones,(鶴保 征城, 富野 寿), ソフトウェア開発の定量化手法, 構造計画研究所, 1998.