修 士 論 文

リファクタリングの履歴を用いた
ソフトウェア開発支援ツールの研究

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

高橋克吏

2006 年 3 月

修 士 論 文

リファクタリングの履歴を用いた ソフトウェア開発支援ツールの研究

指導教官　鈴木正人 助教授

審査委員主査　鈴木正人 助教授
審査委員　落水浩一郎 教授
審査委員　片山卓也 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

410072 高橋克吏

提出年月: 2006 年 2 月

概 要

仕様の変更や機能の拡張によってソフトウェアは少しずつ複雑なものになり、保守性や開発効率が悪くなる。複雑なソフトウェアを改善する方法の一つにリファクタリングがある。リファクタリングとは、ソフトウェアの持つ機能を変えずに内部の構造を変更する作業を指す。これによりソフトウェアの設計を改善することができる。しかしながら、適用するリファクタリング操作の種類や順番によって結果が異なるので、リファクタリングするためには試行錯誤が避けられないという問題がある。本研究では、利用者が行ったリファクタリング操作の履歴を用いることで、より少ない試行でリファクタリングする手法を提案する。

目 次

第1章　背景

ソフトウェアの開発では、仕様の変更や機能の拡張がよく起こる。これらの要求にそのつど対応していくと、最初に設計したものから少しずつ離れ、ソフトウェアは複雑なものになっていく。その結果、開発効率が低下し保守や拡張が困難になる。このような状態に陥ったときの解決策の一つとしてリファクタリング [1] がある。

リファクタリングとは、ソフトウェアの持つ機能を変えずに、その内部構造を変更する作業を指す。リファクタリングを行って、ソフトウェアの内部構造を単純化すると、保守性や拡張性を改善することができる。また、エクストリームプログラミングなど、開発中にリファクタリングを行うことを前提とした開発方法もあり、リファクタリングの有効性が広く認知されている。

しかしながら、実際の開発現場では、ほとんどリファクタリングは行われていない。理由の一つに、リファクタリングによって目的を達成することが困難だということがあげられる。リファクタリングを行う場合は、機能を保持するためにソースコードを少しずつ変更していく。そのため複雑なソフトウェアを改善するには、多くのリファクタリング操作が必要になる。しかしながら、リファクタリングした結果は、適用したリファクタリング操作の種類や順番によって異なる。そのため、ソフトウェア開発の経験が少ない場合や、リファクタリングの経験がない場合は、容易に目的を達成することができない。その結果として目的を達成するためには、試行錯誤しながら少しずつ改善していく必要がある。

1.1　目的

これまで、複雑なソフトウェアをリファクタリングする場合は、何度も試行錯誤しながら少しずつ変更する必要があった。そこで、本研究の目的は、リファクタリング操作の履歴を用いることでより少ない試行でリファクタリングする手法を提案する。そして、この手法を利用した開発技術の確立と検証を行う。

第2章　リファクタリング

この章では、まず本研究で使う用語の定義を行い、次にリファクタリングの手順とその手順に従った例を示し、最後にリファクタリングの問題について示す。

2.1　用語の定義

リファクタリングという言葉は、ソフトウェアの持つ機能を変えずにその内部構造を変更する一連の作業を指す場合とその作業をするためにソースコードに対して行う操作を指す場合があるが、本研究ではそれぞれを区別して扱う。本研究で使う用語を以下のように定義する。

リファクタリング ソフトウェアの持つ機能を変えずにその内部構造を変更する一連の作業を指し、その作業により利用者の目標を達成することができる。

リファクタリング操作 機能を変えずにソースコードを変更する操作を指す。

目標 リファクタリングによって達成すべき内容を表したもの。利用者が設定するリファクタリングのゴール。

部分目標 目標を達成するためにはいくつかのステップに分けてリファクタリングする必要がある。そのときに設定するサブゴール。

基本操作 本研究で作成したリファクタリングツールで自動化したリファクタリング操作を指す。

操作履歴 利用者がリファクタリングするために行った基本操作を記録したもの。

2.2　リファクタリングの手順

リファクタリングは、次の手順で行われる。

1. リファクタリングによって達成すべき目標を決める
2. 問題がどこにあるか探す

3. 部分目標を決める

4. リファクタリング操作を適用する

5. 適用結果を確認する

手順の 1 番目は、リファクタリングによって達成すべき目標を設定する。目標は、リファクタリングに求められていることを適切に表すことが重要である。例えば、ソフトウェアに拡張性を持たせることは重要だが、必要もないのに拡張性を持たせるとかえってソフトウェアの構造を複雑にしてしまうので、要求を満たしかつ作りこみ過ぎないように目標を設定する。

手順の 2 番目は、目標を達成するために、改善すべき問題がどこにあるか探す。問題を探すときは、Fowler が示した不吉な匂いをもとに探す。

手順の 3 番目は、発見された問題を解決するために部分目標を決める。部分目標を決めるときは、最初から目標に向かって進むよりソフトウェアの複雑さを取り除いてから目標に向かうようにしたほうがその後の作業が簡単になる。理由は複雑さを取り除くとこれまで気づかなかった問題が表面化し、その後の作業を進めやすくなる。

手順の 4 番目は、部分目標を達成するためにリファクタリング操作を適用する。ソースコードを変更するときは、一度に多くの変更を行うのではなく少しずつ変更していく。このとき、一つの変更操作が終わったらコンパイルしてテストを行い、ソフトウェアの持つ機能が変わっていないことを確認する。

手順の 5 番目は、リファクタリング操作を適用した結果を確認する。適用した結果に納得がいかなければ、元に戻し別の操作を適用する。部分目標を達成した場合は目標を達成しているか調べる。目標が達成されていれば終了する。目標が達成されていない場合は次の部分目標を設定し操作を続ける。

2.3 リファクタリングの例

簡単なレンタルビデオショップのシステムを使ってリファクタリングの例を示す。

2.3.1 システムの仕様

このシステムは、顧客がビデオを借りたときにその内容をレシートに出力する。レシートは以下の仕様で生成される。

- 映画の種類と借りる日数を入力するとビデオのレンタル料金とレンタルポイントを計算して印刷する
- 映画の種類は一般向け、子供向け、新作の 3 つ
- 料金は、貸し出し日数に依存する

- レンタルポイントは、映画の種類に依存する

このシステムは、レシートを生成するために 3 つのクラスが定義されている。

- ビデオの情報を保持する Movie クラス
- 顧客がビデオを借りていることを表す Rental クラス
- 顧客情報を保持する Customer クラス

レシートに記述される内容は Customer クラスの statement メソッドで生成していて以下の順に処理を行っている。

1. 顧客の情報を出力する
2. レンタルしたビデオの情報を出力する
3. 合計金額と合計ポイントを出力する

プログラム 1 に、各クラスのソースコードを示す。

プログラム 1：Movie.java

```
public class Movie {
    public static final int CHILDRENS = 2;
    public static final int REGULAR = 0;
    public static final int NEW_RELEASE = 1;
    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;
    }
}
```

プログラム 1：Rental.java

```
public class Rental {
    private Movie _movie;
    private int _daysRented;
    public Rental(Movie movie, int daysRented) {
        _movie = movie;
        _daysRented = daysRented;
    }
    public int getFrequentRenterPoints() {
        if((getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
            getDaysRented() > 1)
                return 2;
        else
                return 1;
    }
    public int getDaysRented() {
        return _daysRented;
    }
    public Movie getMovie() {
        return _movie;
    }
}
```

プログラム 1：Customer.java

```
import java.util.LinkedList;
public class Customer {
    private String _name;
    private LinkedList<Rental> _rentals = new LinkedList<Rental>();
    public Customer(String name) {
        _name = name;
    }
    public void addRental(Rental arg) {
        _rentals.add(arg);
    }
    public String getName() {
        return _name;
    }
    public String statement() {
        double totalAmount = 0;
        int frequentRenterPoints = 0;
        StringBuffer result = new StringBuffer();
        result.append("Rental Record for ");
        result.append(getName());
        result.append("\n");
        for(Rental each : _rentals) {
            double thisAmount = 0;
            switch(each.getMovie().getPriceCode()) {
                case Movie.REGULAR:
                    thisAmount += 2;
                    if(each.getDaysRented() > 2) thisAmount +=
                          (each.getDaysRented() - 2) * 1.5;
                    break;
```

プログラム 1：Customer.java(つづき)

```
                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;
            }
            frequentRenterPoints += each.getFrequentRenterPoints();
            result.append("\t");
            result.append(each.getMovie().getTitle());
            result.append("\t");
            result.append(thisAmount);
            result.append("\n");
            totalAmount += thisAmount;
        }
        result.append("Amount owed is ");
        result.append(totalAmount);
        result.append("\n");
        result.append("You earned ");
        result.append(frequentRenterPoints);
        result.append("frequent renter points");
        return result.toString();
    }
}
```

2.3.2 リファクタリングの適用例

このシステムには、ビデオを借りたときに出力するレシートのフォーマットが一つしかなかったので、いくつかの新しい出力フォーマットを追加することになった。まだ新しいフォーマットの形式は決まっていないが、決まったらすぐに新しいフォーマットを追加できるようにリファクタリングを行って設計を改善する。

目標の設定

まず、このシステムに対して行うリファクタリングの目標を決める。この例では、新しいレシートのフォーマットを容易に追加できるように設計を改善することが目標である。そこで、目標を「レシートのフォーマットの追加を容易にする」に設定する。

問題の特定

次に、ソースコードから問題のある部分を探す。一番最初に目に付くのは、約 40 行のコードで構成されている statement メソッドである。このメソッドからは、「長すぎるメソッド」という不吉な匂いがする。この匂いは、メソッドが長すぎると処理の内容が分かりにくくなることを示している。この statement メソッドをよく見ると 3 つの処理を行っていることが分かる。それぞれを分割すれば処理内容を簡単に理解できそうだということが分かる。

1 回目の部分目標

長すぎるメソッドという不吉な匂いのする statement メソッドを分割する。長すぎるメソッドは、「Extract Method」を適用することで改善することができる。statement メソッドからは、ヘッダーとフッターを作成する部分とレンタルしたビデオの情報作成部分を抽出できそうなので、「statement メソッドを 3 つの処理に分割する」という部分目標を立てる。

1 回目の操作

部分目標を達成するために、まずヘッダー部分を「Extract Method」で抽出し、printHeader メソッドを作成した。その後フッター部分を「Extract Method」で抽出し printFooter メソッドを作成した。プログラム 2 に適用結果を示す。しかし、レンタルしたビデオの情報を作成している部分は抽出できなかった。理由は、statement メソッドのレンタルしたビデオの情報を出力している部分でレンタル料金の合計とレンタルポイントの合計を計算しているためであり、この 2 つの値は、フッター部分で利用している。メソッドから 2 つ以上の値を返す必要がある場合は、返す値を 1 つのデータオブジェクトにまとめる必要がある。しかし、新しくデータオブジェクトを作成するには別のリファクタリング操作が必要になるので、3 回目の Extract Method は失敗した。

適用した操作	適用箇所	処理内容
ExtractMethod	statement	ヘッダー部分の抽出
ExtractMethod	statement	フッター部分の抽出
ExtractMethod	statement	レンタルしたビデオの情報作成部分を抽出

プログラム 2：Customer.java

```
...
public String statement() {
    double totalAmount = 0;
    int frequentRenterPoints = 0;
    StringBuffer result = new StringBuffer();
    printHeader(result);
    for(Rental each : _rentals) {
        ....
    }
    printFooter(result, totalAmount, frequentRenterPoints);
    return result.toString();
}
private void printHeader(StringBuffer result) {
    result.append("Rental Record for ");
    result.append(getName());
    result.append("￥n");
}
private void printFooter(StringBuffer result, int totalAmount,
    double frequentRenterPoints) {
    result.append("Amount owed is ");
    result.append(totalAmount);
    result.append("￥n");
    result.append("You earned ");
    result.append(frequentRenterPoints);
    result.append("frequent renter points");
}
...
```

1 回目の評価

部分目標を達成するために 3 つの処理に分割しようとした結果、レンタルしたビデオの情報を出力する部分を抽出することに失敗した。この部分を抽出するためにはレンタル料金の合計とサービスポイントの合計を計算する部分を抽出する必要があることが分かった。そこで、適用した 2 つの操作をキャンセルして、先にこの 2 つを抽出する。

2 回目の部分目標

1 回目の操作結果から、レンタル料金の合計とサービスポイントの合計を計算している部分を先に抽出する必要があることが分かった。レンタル料金の計算部分に注目すると、ビデオ一つ一つの料金も statement メソッド内で計算していることが分かる。レンタル料金の合計を計算する前に、先にこちらを抽出することにする。またこの処理は、Rental クラスのフィールドだけを使って計算していことから、本来は Rental クラスにあるべき処理だと考えられる。そこで、部分目標を「レンタル料金の計算部分を statement メソッドから抽出し、Rental クラスへ移動する」にする。

2 回目の操作

レンタル料金の計算部分を「Extract Method」で抽出して、amountFor メソッドを作成する。抽出した amountFor メソッドの引数名が適切でなかったので「Rename」を適用して、each から aRental に変更する。同様に、amountFor メソッドの変数名にも「Rename」を適用して、thisAmount から result に変更する。amountFor メソッドに「Move Method」を適用して、Rental クラスへ移動しメソッド名を getCharge に変更する。一時変数をメソッド呼び出しに変更するために「Replace Temp with Query」を適用して、statement メソッドの thisAmount 変数を Rental クラスの getCharge メソッドの呼び出しに変更する。適用結果をプログラム 3 に示す。

適用した操作	適用箇所	処理内容
ExtractMethod	statement	レンタル料金の計算部分の抽出
Rename	amountFor	引数 each を適切な名前に変更
Rename	amountFor	変数 thisAmount を適切な名前に変更
MoveMethod	amountFor	amountFor メソッドを Rental へ移動
ReplaceTempWithQuery	statement	thisAmount をメソッド呼び出しに変更

プログラム 3：Rental.java

```
public class Rental {
    ...
    public double getCharge() {
        switch(getMovie().getPriceCode()) {
            case Movie.REGULAR:
                thisAmount += 2;
                if(getDaysRented() < 2) thisAmount +=
                    (getDaysRented() - 2) * 1.5;
                break;
            case Movie.NEW_RELEASE:
                thisAmount += getDaysRented() * 3;
                break;
            case Movie.CHILDRENS:
                thisAmount += 1.5;
                if(getDaysRented() > 3) thisAmount +=
                    (getDaysRented() -3) * 1.5;
                break;
        }
    }
    ...
}
```

プログラム 3：Customer.java

```
import java.util.LinkedList;
public class Customer {
    ...
    public String statement() {
        double totalAmount = 0;
        int frequentRenterPoints = 0;
        StringBuffer result = new StringBuffer();
        result.append("Rental Record for ");
        result.append(getName());
        result.append("\n");
        for(Rental each : _rentals) {
            frequentRenterPoints += each.getFrequentRenterPoints();
            result.append("\t");
            result.append(each.getMovie().getTitle());
            result.append("\t");
            result.append(each.getCharge();
            result.append("\n");
            totalAmount += each.getCharge();
        }
        result.append(totalAmount);
        result.append("\n");
        result.append("You earned ");
        result.append(frequentRenterPoints);
        result.append("frequent renter points");
        return result.toString();
    }
}
```

2 回目の評価

ビデオのレンタル料金の計算処理を statement メソッドから抽出し Rental クラスへ移動した結果、各クラスの役割が明確になった。また、余分な一時変数も取り除くことができた。この操作の結果、レンタル料金の合計金額を容易に抽出できるようになった。

3 回目の部分目標

2 回目の操作結果によって合計金額の計算部分を用意に抽出できるようになった。そこで部分目標を「合計金額の計算部分を statement メソッドから抽出する」にする。

3 回目の操作

レンタル料金の合計は、レンタルしたビデオの情報を表示するループの中で計算されているので、「Split Loop」を適用してこのループを分割する。レンタル料金の合計を計算するループに「Extract Method」を適用して抽出し、getTotalCharge メソッドを作成する。抽出した getTotalCharge メソッドの変数名に「Rename」を適用して、totalAmount から result に変更する。statement メソッドの一時変数 totalAmount に「Replace Temp with Query」を適用して getTotalCharge メソッドの呼び出しに変える。適用結果をプログラム 4 に示す。

適用した操作	適用箇所	処理内容
SplitLoop	statement	合計レンタル料金の計算部分の分割
ExtractMethod	statement	合計レンタル料金の計算部分の抽出
Rename	getTotalCharge	変数 totalAmount を適切な名前に変更
ReplaceTempWithQuery	statement	totalAmount をメソッド呼び出しに変更

プログラム 4：Customer.java

```
public class Customer {
    ...
    public String statement() {
        ...
        for(Rental each : _rentals) {
            frequentRenterPoints += each.getFrequentRenterPoints();
            result.append("\t");
            result.append(each.getMovie().getTitle());
            result.append("\t");
            result.append(each.getCharge();
            result.append("\n");
        }
        result.append(getTotalCharge());
        result.append(totalAmount);
        result.append("\n");
        result.append("You earned ");
        result.append(frequentRenterPoints);
        result.append("frequent renter points");
        return result.toString();
    }
    private double getTotalCharge() {
        double result = 0;
        for(Rental each : _rentals) {
            result += each.getCharge();
        }
    }
}
```

3 回目の評価

レンタル料金の合計を計算する部分を statement メソッドから抽出することができた。余分な一時変数を取り除くことができた。

4 回目の部分目標

statement メソッドには、まだ合計レンタルポイントの計算部分が残っている。そこで部分目標を「合計レンタルポイントの計算部分を statement メソッドから抽出する」にする。

4 回目の操作

サービスポイントの合計を計算している部分に「Split Loop」を適用してループを分離する。サービスポイントの合計を計算するループに「Extract Method」を適用して抽出し、getTotalFrequentRenterPoints メソッドを作成する。抽出した getFrequnentRenterPoints メソッドの変数名に「Rename」を適用して、frequentRenterPoints から result に変更する。statement メソッドの一時変数 frequentRenterPoints に「Replace Temp with Query」を適用して getTotalFrequentRenterPoints メソッドの呼び出しに変える。適用結果をプログラム 5 に示す。

適用した操作	適用箇所	処理内容
SplitLoop	statement	合計レンタルポイントの計算部分の分割
ExtractMethod	statement	合計レンタルポイントの計算部分の抽出
Rename	getTotalFrequent RenterPoints	変数 frequentRenterPoints を 適切な名前に変更
ReplaceTempWithQuery	statement	frequentRenterPoints を メソッド呼び出しに変更

プログラム 5：Customer.java

```
import java.util.LinkedList;
public class Customer {
    ...
    public String statement() {
        ...
        for(Rental each : _rentals) {
            result.append("\t");
            result.append(each.getMovie().getTitle());
            result.append("\t");
            result.append(each.getCharge();
            result.append("\n");
        }
        result.append(getTotalCharge());
        result.append("\n");
        result.append("You earned ");
        result.append(getTotalfrequentRenterPoints());
        result.append("frequent renter points");
        return result.toString();
    }
    private int getTotalFrequentRenterPoints() {
        int result = 0;
        for(Rental each : _rentals) {
            result += each.getFrequentRenterPoints();
        }
    }
    ...
}
```

4 回目の評価

サービスポイントの合計を計算する部分を statement メソッドから抽出することができた。余分な一時変数を取り除くことができた。また、3 回目と 4 回目の操作の結果、statement メソッドからレンタル料金の計算部分と、レンタルポイントの計算部分をすべて取り除くことができた。

5 回目の部分目標

3 回目の操作と 4 回目の操作で、statement メソッドからレシートの作成する処理以外の機能を取り除いたので、1 回目で設定した部分目標「statement メソッドを 3 つの処理に分割する」を改めて適用する。

5 回目の操作

statement からヘッダーの情報を作成している部分に「Extract Method」を適用して、printHeader メソッドを作成する。statement からフッターの情報を作成している部分に「Extract Method」を適用して、printFooter を作成する。最後に、レンタルしたビデオに関する情報を作成する部分に「Extract Method」を適用して、printContents を作成する。適用結果をプログラム 6 に示す。

適用した操作	適用箇所	処理内容
ExtractMethod	statement	ヘッダー部分の抽出
ExtractMethod	statement	フッター部分の抽出
ExtractMethod	statement	レンタルしたビデオの情報作成部分を抽出

プログラム 6：Customer.java

```
import java.util.LinkedList;
public class Customer {
    ...
    public String statement() {
        StringBuffer result = new StringBuffer();
        printHeader(result);
        printContents(result);
        printFooter(result);
        return result.toString();
    }
    private void printHeader(StringBuffer result) {
        ...
    }
    private void printContents(StringBuffer result) {
        for(Rental each : _rentals) {
            result.append(" ¥ t");
            result.append(each.getMovie().getTitle());
            result.append(" ¥ t");
            result.append(each.getCharge();
            result.append(" ¥ n");
        }
    }
    private void printFooter(StringBuffer result) {
        ...
    }
    ...
}
```

5 回目の評価

5 回目の操作の結果、statement メソッドは、ヘッダーの作成、レンタルしたビデオの情報作成、フッターの作成の 3 つの役割を持っていることが一目で分かるようになった。また、レンタル料金の合計やレンタルポイントの合計をメソッド呼び出しで取得できるようになったため、最初に設定した▲譽掘璽箸離俟　璽泪奪箸猟媛辰鰺動廚砲垢襖という目

的を達成できた。

2.3.3 適用例の考察

この例で示したリファクタリングの目標はレシートのフォーマットの追加を容易にすることだった。始めは1回のリファクタリング操作で目標が達成できるかにみえたが、この目標を達成するためには図2.1に示したように、失敗も含めて5回のリファクタリング操作を行っている。5回の操作のうち、2回目から4回目の操作は、1回目の操作が失敗したことで分かった問題を解決するために行った操作である。さらに1つのリファクタリング操作を行うために複数の基本操作を必要としている。このように、リファクタリングする場合は、問題のある部分を探し出すことが難しく、実際に操作を行うまで結果の予測がたてられない。また目標を達成するためには多くの操作が必要になる。

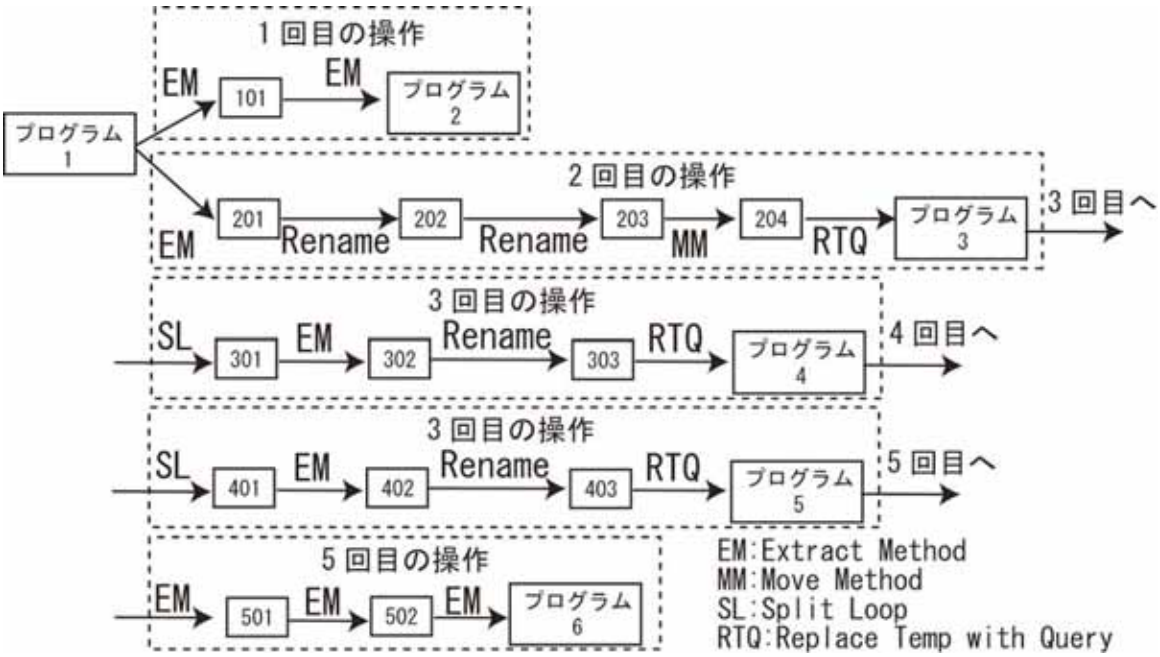

図 2.1: 例の操作手順

2.4 リファクタリングの問題

リファクタリングする場合の問題点として以下の2つがあげられる。1つ目は、目標を達成するためには多くのリファクタリング操作を組み合わせる必要がある。2つ目は、リファクタリング操作の適用結果の評価が困難である。

2.4.1 操作の組み合わせ

リファクタリングをする場合は、機能を変えないために変更操作を小さなステップに分けて行う必要があるため、目標を達成するためには 2.3 節で示したように多くのリファクタリング操作を組み合わせる必要がある。しかしながら適用する操作の種類や順番によって結果が異なり、さらに操作を適用するごとにソースコードが書き換わってしまうため操作結果の予測がたてにくい。そのため適切な操作の組み合わせを見つけるために試行錯誤が必要になる。

2.4.2 適用結果の評価

リファクタリング操作を適用した結果が目標を達成する方向に進んでいるか評価する必要がある。しかしながらリファクタリングの経験が少ない場合は適切な評価をすることが困難なためソフトウェアを十分に改善することができない場合が多い。

第3章　既存のリファクタリング支援ツール

現在ではリファクタリングの有効性が認められ多くのソフトウェア開発環境でリファクタリングをサポートしている。そこで、本研究のアプローチを説明する前に既存のリファクタリングツールについて調査した結果を示す。調査対象は、RefactorIT[2]、Eclipse[3]、Java Refactoring Browser[4](以下、JRB とする) の 3 つで、以下の項目を調査した。

- 実装するリファクタリング操作 (一覧を表 3.1 に示す)
- 特徴的な操作
- 特徴的な機能

3.1　RefactorIT

RefactorIT は、単独で動作するものと他の開発環境のプラグイン版がある。その中から Eclipse プラグイン版について調べた。

開発元　Aqris Software

特徴　多くのリファクタリング操作を実装している。特徴的な機能の一つに Move 操作がある。この操作は、メソッドやフィールドをあるクラスから別のクラスへ移動する。この操作をカプセル化されているフィールドに対して適用する場合は、そのアクセサも一緒に移動することを検討しなければならない。Eclipse や JRB では、フィールドとアクセサに対して個々に操作を適用する必要がある。RefactorIT では、ツールの利用者がアクセサも移動する必要があると判断した場合は、一回の操作でフィールドとアクセサをまとめて移動することができる。リファクタリング操作以外の特徴的な機能として、図 3.1 のようにクラスやメソッドの依存関係を調べる機能と図 3.2 のようにソフトウェアメトリクスを調べる機能がある。

表 3.1: 各ツールが実装するリファクタリング操作一覧

操作名	RefactorIT	Eclipse	JRB
Rename	○	○	○
Move	○	○	○
Add Delegate Method	○	○	
Change Method Signature	○	○	
Clean Imports	○	○	
Convert Anonymous Class to Nested		○	
Convert Nested Type to Field		○	
Convert Temp To Field	○	○	
Create Constructor	○	○	
Create Factory Method	○	○	
Encapsulate Field	○	○	○
Extract Constant		○	
Extract Method	○	○	
Extract Superclass/Interface	○	○	○
Inline	○	○	
Introduce Explaining Variable	○	○	
Merge Class			○
Minimize Access Rights	○		
Pull Up/Push Down	○	○	○
Use Supertype Where Possible	○	○	

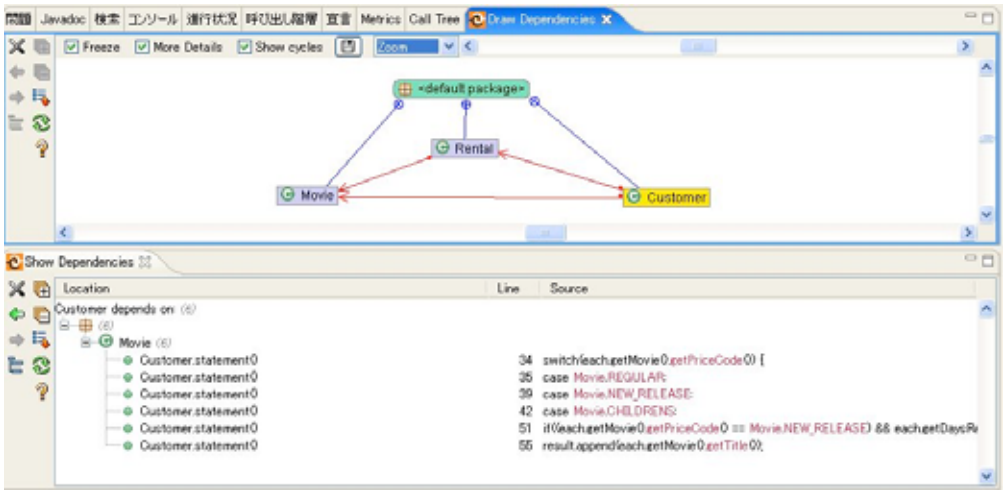

図 3.1: 依存関係の表示例

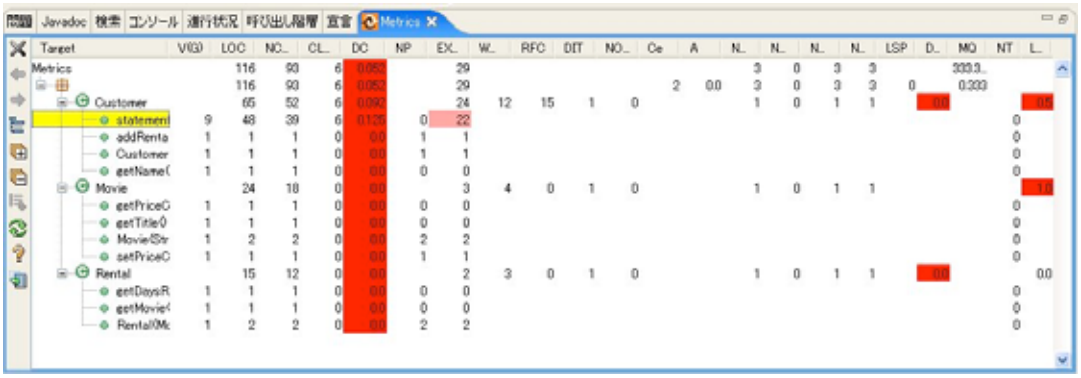

図 3.2: メトリクスの計算結果の例

3.2 Eclipse

Java の統合開発環境として広く使われている Eclipse に標準で付属するリファクタリング機能について調べた。

開発元 Eclipse.org

特徴 多くのリファクタリング操作を実装している。特徴的な機能の一つに Extract Method 操作がある。Extract Method は、メソッドの一部分を抽出して新しいメソッドを作成する操作である。Eclipse の Extract Method は、図 3.3 に示すように抽出した部分と重複するコードをまとめて抽出したメソッドの呼び出しに置き換えることができる。

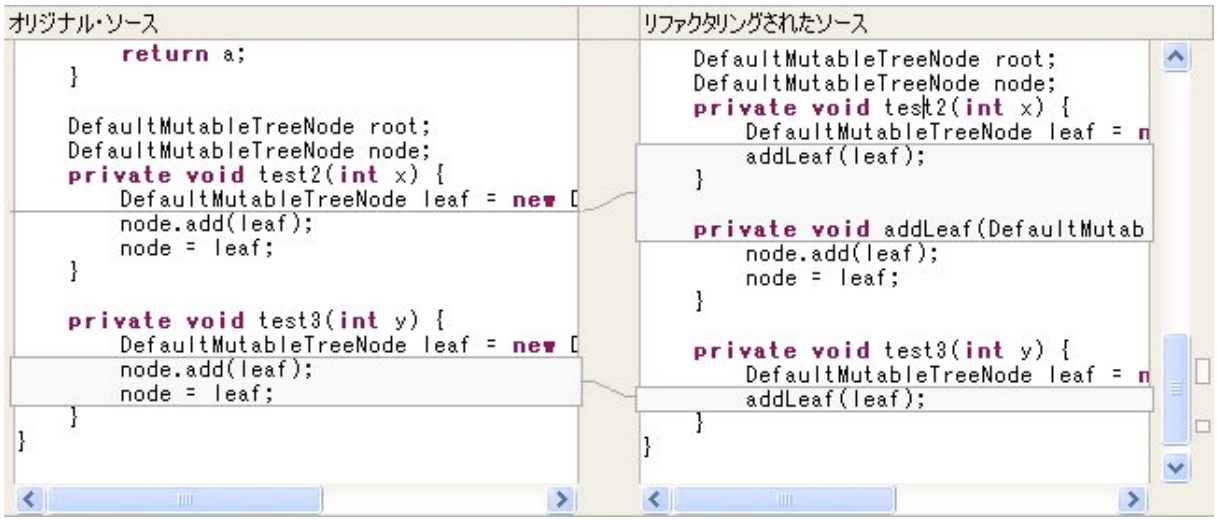

図 3.3: Eclipse の ExtractMethod の例

3.3 Java Refactoring Browser

リファクタリング操作を履歴として保存することで、次に行う操作を提示することができる JRB について調べた。

開発元 立命館大学理工学部情報科学ソフトウェア基礎技術研究室

特徴 実装している操作が RefactorIT や Eclipse に比べて少ない。JRB の特徴は実行したリファクタリング操作を履歴として保存し、これを使って利用者に次の操作を提案することができる。この履歴には適用した操作名と操作の対象となったファイル名が保存されていて、一つのファイルに対して連続して行った操作が一つの操作列と

して管理される。図 3.4 の一行一行が操作列になる。JRB の履歴の利用方法は、利用者がリファクタリング操作を行ったときその操作名で履歴を検索し、その操作の次に行った操作を次の操作として提案する。この履歴の利用例を図 3.5 に示した操作列を使って説明する。この図に示した 2 つの操作列が履歴に保存されている状態で、利用者が EncapsulateField を実行した場合、波線が引かれている 2 つの Encapsulate が一致するので次の操作として RenameMethod と SelfEncapsulateField を提案する。利用者が続けて SelfEncapsulateField を実行した場合、二重線を引いた部分が一致するので次の操作として RenameField を提案する。JRB では 1 つ前または 2 つ前の操作を使って履歴を検索するため、このまま続けて RenameField を実行した場合は、RenameField もしくは SelfEncapsulateField と RenameField の組み合わせを使って検索することになる。

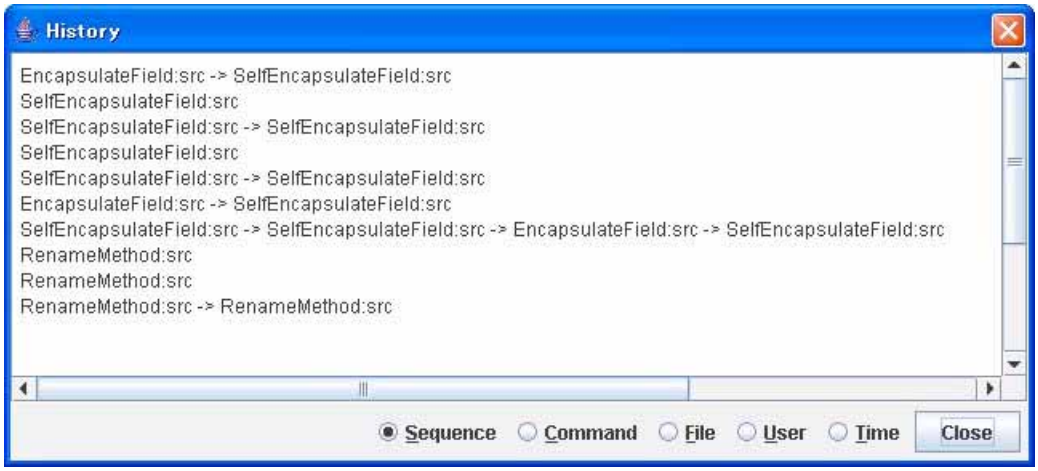

図 3.4: JRB の履歴

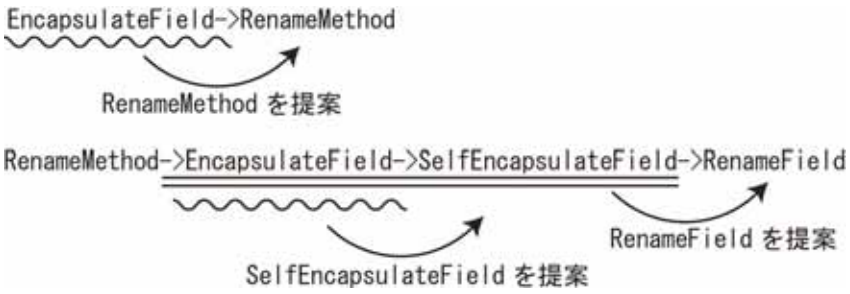

図 3.5: JRB の履歴の利用例

3.4 まとめ

それぞれのツールを使ってみて感じたことを以下にまとめる。

3.4.1 RefactorIT

RefactorIT は、実用に耐えるだけの多くのリファクタリング操作をサポートしているため、適用する操作が決まっているときは有効なツールである。また、Move 操作が必要なときは、関連する複数のフィールドやメソッドを移動する場合が多いので、一度の操作でまとめて移動できる機能は非常に使いやすかった。

依存関係やメトリクス値を計算する機能は、問題のありそうな部分を見つけ出す手がかりとして使えた。また、リファクタリング操作の適用後のメトリクス値を見ることで、客観的な評価が可能であった。

3.4.2 Eclipse

Eclipse は、RefactorIT と同様に実用に耐えるだけの多くのリファクタリング操作をサポートしている。Extract Method は、コピー&ペーストを多用しているソースコードや、ソースコード中で慣用句的に使われる処理を一つのメソッド呼び出しにまとめるときに有効である。

Eclipse のリファクタリング機能は、一つ一つの操作の完成度が高いのでソースコードを書いているときに気になった部分を修正する場合に有効である。

3.4.3 Java Refactoring Browser

JRB は、実装しているリファクタリング操作の数が少ないため、実際の開発に使うには不向きであると感じた。特に、Extract Method のようなよく使われる操作をサポートしていないため利用範囲が限定される。

JRB のリファクタリング操作履歴を使った、次の操作の提案機能はあまり有効であると感じなかった。なぜなら履歴に残されている操作列には、その順番で操作が行われたという情報しか含まれていないため、なぜその順番で操作をする必要があるのか理由が分からないからだ。

第4章　アプローチ

リファクタリングするときは次のような問題がある。リファクタリングによって目標を達成するためには多くのリファクタリング操作を組み合わせる必要があり試行錯誤が避けられない。リファクタリング操作の適用結果の評価が困難である。本研究ではこれらの問題を改善するために次のようなアプローチをとる。

- リファクタリングするときに操作をすべて人手で行うのは利用者の負担が大きいので Fowler のカタログからよく使われる操作を自動化する
- リファクタリングによって目標を達成するときに一度に多くの変更をソースコードに加えてしまうとバグの混入による無駄な手間が増えてしまうので少しずつリファクタリングを行うように部分目標を設定する
- リファクタリング操作の履歴を保存しこれを利用してリファクタリングするときに必要になる試行錯誤を軽減する
- リファクタリング操作の適用前と適用後のメトリクス値の計算結果を利用者に提示し操作の妥当性に関する評価を客観的に行えるようにする

第5章　ツールの実現方法

4 章で示したアプローチをツールで実現する方法について説明する。

5.1　リファクタリング操作の自動化

リファクタリングする場合は多くの操作を行う必要があるがすべてのリファクタリング操作を人手で行うのは負担が大きいので操作を自動化して容易に操作を適用できるようにする。自動化したリファクタリング操作を基本操作と呼ぶ。

5.1.1　基本操作

基本操作として Fowler のカタログからよく使われる操作を 19 個抽出して自動化した。複雑なリファクタリングは、この基本操作を組み合わせることで行う。基本操作の一覧を表 5.1 に示す。

5.1.2　基本命令

基本操作を計算機で実行するために形式化したものを基本命令と呼ぶ。基本命令は、基本操作を識別する識別子と適用箇所と適用実体からなるオペランドを持ち、次のように記述する。

識別子 (適用箇所, 適用実体)

適用箇所は、基本操作を適用する場所を示すもので、パッケージ名、クラス名、メソッド名、領域の順に指定する。領域は、空行などを読み飛ばした論理的な構造によって決まる開始行と終了行で指定する。適用箇所は次のように記述する。

パッケージ名::クラス名::メソッド名::[開始行, 終了行]

適用実体は、各基本操作を実行するために必要な情報で基本操作毎に異なる情報を持ち、Move Method の場合は次のように定義される。

識別子 MM

表 5.1: 基本操作一覧

名称	機能
Move Class	別のパッケージへクラスを移動する
Move Method	メソッドを移動する
Move Field	フィールドを移動する
Extract Class	クラスを抽出する
Extract Interface	メソッドの抽出する
Extract Method	メソッドを抽出する
Encapsulate Field	フィールドをカプセル化する
Self Encapsulate Field	フィールドを自己カプセル化する
Pull Up Method	メソッドを引き上げる
Push Down Method	メソッドを引き下げる
Pull Up Field	フィールドを引き上げる
Push Down Field	フィールドを引き下げる
Replace Data Value with Object	指定したフィールドを持つオブジェクトを作る
Replace Type Code with Subclass	指定したタイプコード毎にサブクラスを作る
Replace Conditional with Polymorphism	指定したメソッドの条件分をポリモーフィーズムで置き換える
Replace Type Code with State/Strategy	タイプコード毎にサブクラスを作りState/Strategy の雛形を作る
Replace Temp with Query	一時変数をメソッドの呼び出しに置き換える
Rename	クラスやメソッドなどの名前を変更する
Split Loop	ループ処理を分割する

形式 MM(p::c::m, MMop(”p::c2”, ”m2”))

適用箇所 メソッド

適用実体 第一引数=移動先のクラス、第二引数=移動先で使う新しいメソッド名

5.1.3 基本命令の例

基本命令の例として Move Method を示す。Move Method はあるクラスから別のクラスへメソッドを移動する基本操作である。図5.1で示す例では、パッケージ p の Customer クラスの amountFor メソッドをパッケージ p の Rental クラスへ移動しメソッド名を getCharge に変更している。

基本命令：MM(p::Customer::amountFor, MMop(”p::Rental”, ”getCharge”))

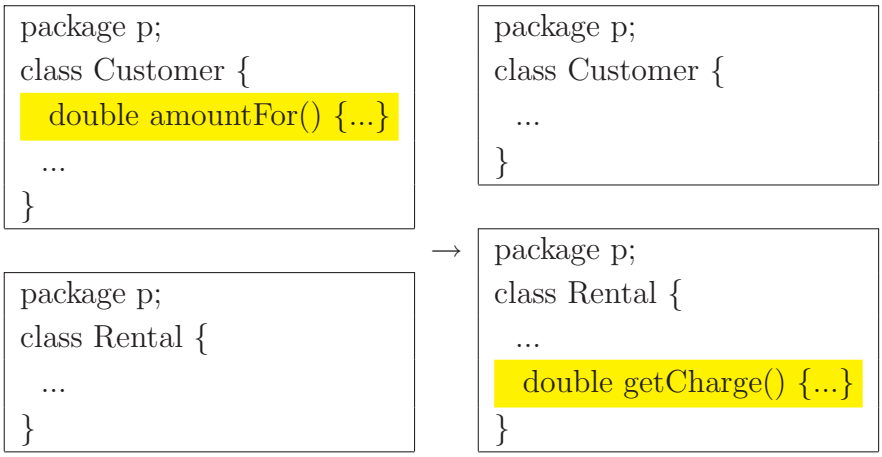

図 5.1: 基本命令の例

5.2 変更方針

本研究では、リファクタリングするときに部分目標を設定して少しずつリファクタリングすることでバグの混入を少なくする。少しずつリファクタリングするために部分目標とその部分目標を達成する基本命令の列の組を変更方針と定義する。変更方針は次に示す情報で構成される。

変更方針= 部分目標+基本命令の列+変更方針の状態

変更方針には変更方針の状態を表す情報がありその状態を以下に示す。

未達成 部分目標を達成していない状態

達成 部分目標を達成している状態

失敗 部分目標を達成できない状態

5.2.1 変更方針の例

変更方針の例として 2.3 節の 5 回目の操作を用いて説明する。まず目標を達成するためのサブゴールである部分目標「statement メソッドを 3 つの処理に分割する」を設定する。このとき操作適用前のプログラム (図 5.2 の P0) は部分目標は達成されていないので変更方針は未達成状態である。次に部分目標を達成するために基本操作を適用する。1 回目の操作適用結果は図 5.2 の P1 になるが、まだ部分目標は達成されていないので操作を続ける。最終的に 3 回の操作を適用することで部分目標が達成 (図 5.2 の P3) される。

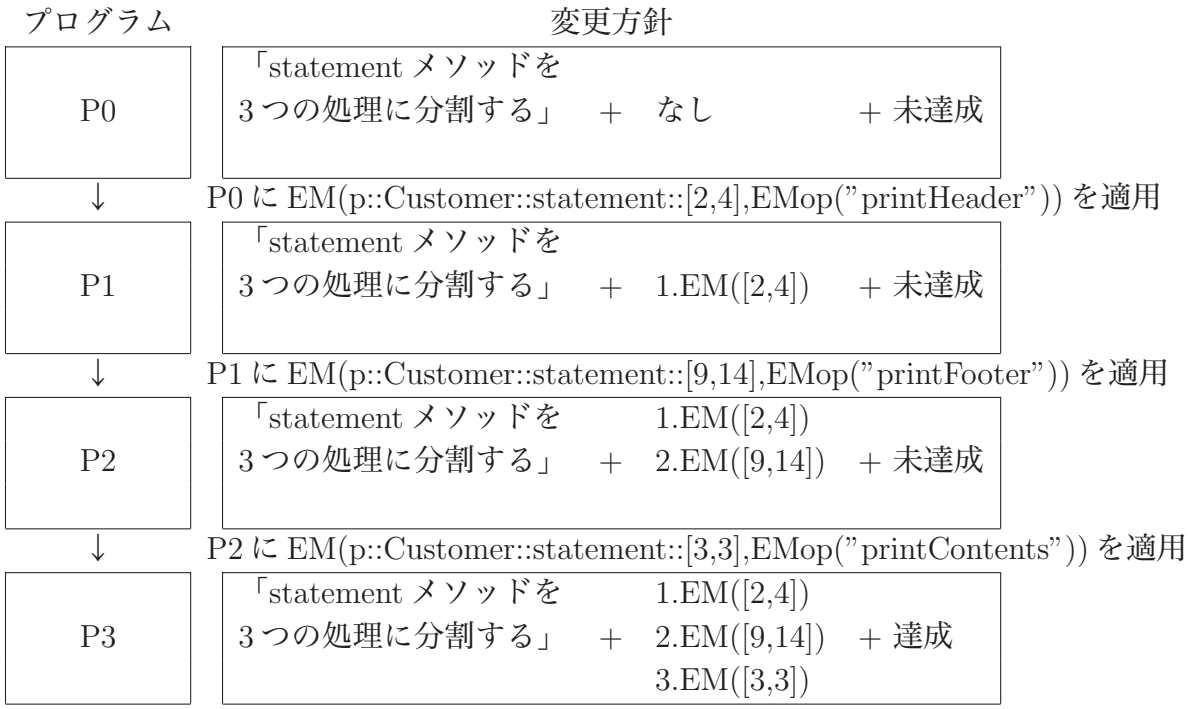

図 5.2: 変更方針の例

5.3 操作履歴

リファクタリングするときの試行錯誤を軽減するために利用者の行った基本操作を履歴として保存する。この履歴を操作履歴と呼ぶ。

5.3.1 履歴の構成

操作履歴にはリファクタリングによって達成すべき目標と目標を達成するために行った操作が記録される。操作履歴は利用者が試行錯誤している過程を記録するために利用者が行った操作を木構造で管理しそのノードは変更方針からなる。2.3 節の例で示したリファクタリングを行った場合、操作履歴は図 5.3 のようになる。

図 5.3: 操作履歴の例

5.4　メトリクス

リファクタリング操作を適用した場合は適用した操作が妥当であるか評価する必要がある。このときソフトウェア開発やリファクタリングの経験が豊富であれば適用結果を容易に判定することができるが経験が少ない場合は適切な判定をすることが困難である。そこで本研究ではリファクタリング操作を適用する前と適用した後のメトリクス値を利用者に提供することで客観的な評価を行えるようにする。

5.4.1　計算可能なメトリクス値

本研究では Fowler によって定義された「不吉な匂い」をメトリクス値を用いて客観的に判定できるようにする。これにより操作適用前と適用後でどの程度ソフトウェアが改善されたのかを客観的に評価できるようにする。計算可能なメトリクス値と対応する不吉な匂いの一覧を表 5.4.1 に示す。

表 5.2: 計算可能なメトリクス値一覧

名称 (略記)	説明	不吉な匂い
Line of Code in Class(LOCc)	クラスの行数	巨大なクラス
Line of Code in Method(LOCm)	メソッドの行数	長すぎるメソッド
Number of Method(NOM)	メソッドの数	巨大なクラス
Number of Field(NOF)	フィールドの数	巨大なクラス
Number of Parameter(NOP)	引数の数	多すぎる引数
Number of Switch Statement(NOS)	switch 文の数	switch 文
Comment Line of Code(CLOC)	コメントの数	コメント

5.4.2　メトリクス値の利用例

メトリクス値の利用例として 5.2 節で示した変更方針の例を用いる。5.2 節では部分目標を「statement メソッドを 3 つの処理に分割する」に設定していた。この部分目標は statement メソッドには「長すぎるメソッド」という不吉な匂いがあり、かつメソッドが「3 つの処理」からなることをソースコードから簡単に読み取れるので設定できた。一般的に「長すぎるメソッド」という不吉な匂いを改善するときはメソッドを何個に分割するかではなくメソッドの長さを短くすることが目標になる。そこでメソッドの長さの目安としてメトリクス値を利用しリファクタリング操作の適用結果を客観的に評価できるようにする。その例を図 5.4 に示す。この例では部分目標を「statement メソッドを分割する」

に設定しメソッドの長さの目安として 10 行未満（LOCm(statement) < 10）を設定している。操作適用前のプログラム（図 5.4 の P0）の LOCm(statement)=17 であったのが 3 回目の操作適用後 (図 5.4 の P3) では LOCm=5 になっていることが分かる。このように操作適用前と適用後のメトリクス値を提供することでプログラムがどの程度改善されたか客観的な評価を行えるようになる。

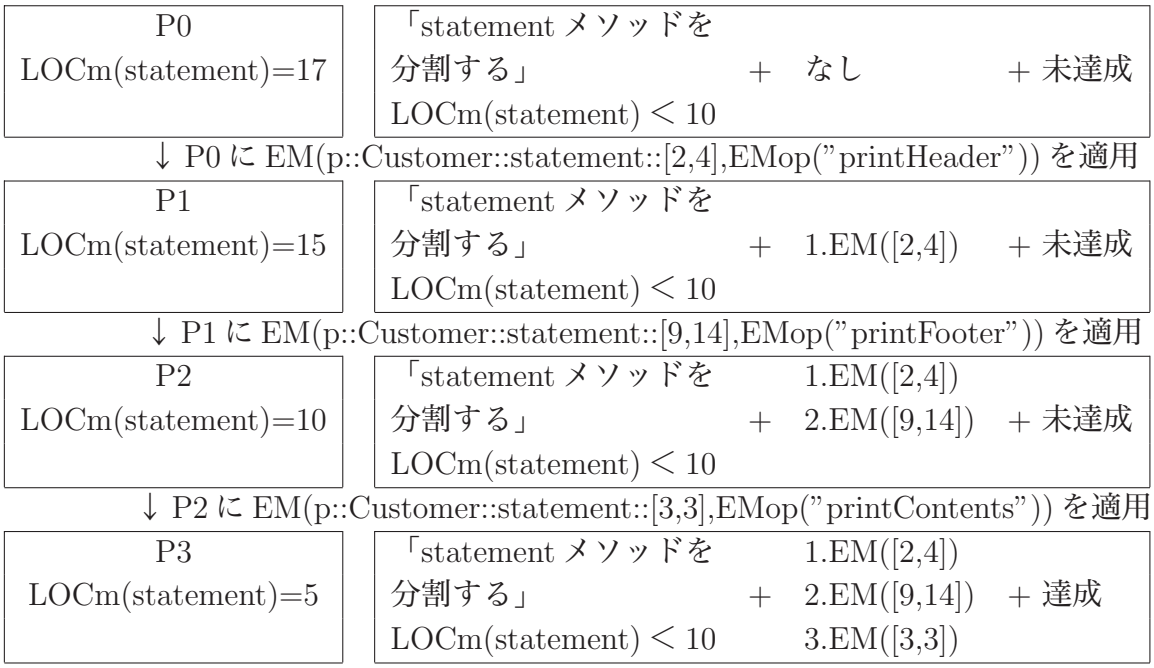

図 5.4: メトリクス値の例

第6章　操作履歴の利用方法

操作履歴の利用方法として後戻りの支援と操作結果の比較について説明する。

6.1　後戻りの支援

リファクタリングによって目標を達成するためには試行錯誤が避けられず、このときリファクタリング操作を適用した結果に納得がいかなければ適用前の状態に戻すといった後戻りが何度も発生する。本研究では操作履歴を利用し後戻りを支援することで利用者の負担を軽減する。例えば、あるプログラム P0 に対してリファクタリングを行う。目標を達成するために 2 つの変更方針を設定しそれに従って操作を適用した結果を PA とする。このときの 2 つの変更方針は図 6.1 の左側のように変更方針 A1 と変更方針 A2 として保存される。利用者は、PA が目標を達成できていない、またはこのままリファクタリングを続けても目標を達成することが困難であると判断した場合は P0 に戻して変更方針 B を採用する。この変更方針 B を適用した結果を PB とする。もしこのとき利用者が PB より PA の方が目標に近いと判断した場合は、操作履歴から変更方針 A2 を選択することで図 6.1 のように P0 に変更方針 A1 と変更方針 A2 の操作を適用して PA を得る。このように操作履歴に木構造を採用したことで利用者の行った操作をすべて記録できるようになり、利用者の要求に応じて過去に行った操作結果まで容易に戻れるようになる。

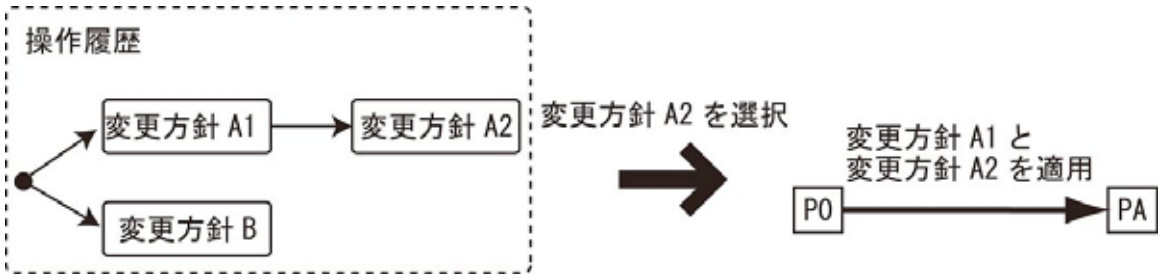

図 6.1: 後戻りの支援の例

6.2　操作結果の比較

操作履歴を利用して複数の変更方針を適用した結果を同時に比較できるようにする。適用結果同士を比較することでより目標に近い結果を選択できるようにする。具体的には、あるプログラム P0 をリファクタリングするときに 3 つの変更方針が考えられる場合、利用者はそれぞれを P0 に適用してその結果同士を比較することでより目標に近い変更方針を選択できるようにする。図 6.2 の場合は P0 に対して 3 つの変更方針（変更方針 A、変更方針 B、変更方針 C）を適用してそれぞれの適用結果（PA、PB、PC）同士を比較することでより目標に近い結果を容易に得られるようにする。

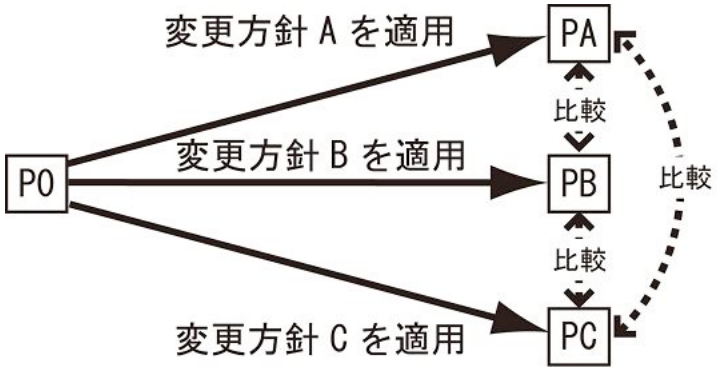

図 6.2: 操作結果の比較の例

第7章　ツールの実装

7.1　仕様

7.1.1　実装環境

本研究で作成したツールの実装環境を以下に示す。

- OS : WindowsXP
- プログラミング言語 : Java (J2SE 5.0)
- 使用ライブラリ : eclipse-JDT-3.0.2
- リファクタリング対象：Java(J2SE 1.4)

7.1.2　ユースケース

本研究で作成したツールのユースケース図を図 7.1 に示す。

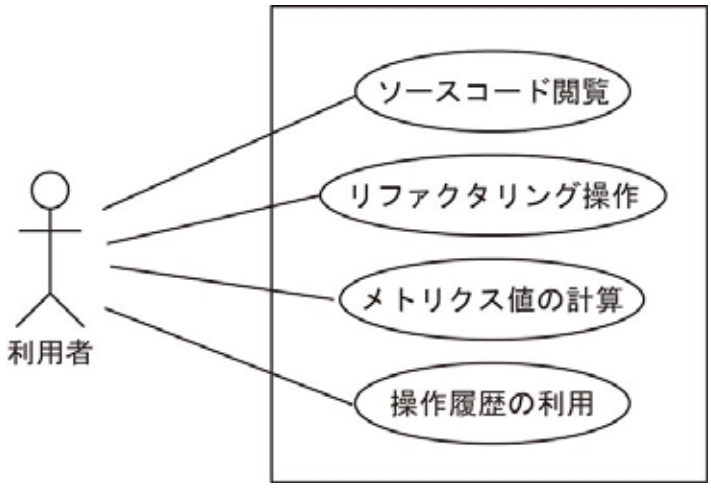

図 7.1: ユースケース図

ユースケース名 ソースコード閲覧

アクター 利用者

概要 利用者がソースコードを閲覧する

メインシナリオ

1. 利用者は閲覧したいソースコードがあるディレクトリを指定する
2. システムは指定されたディレクトリから Java ソースコードの一覧を表示する
3. 利用者は一覧から閲覧したいソースコードを選択する
4. システムは選択されたソースコードの内容を表示する

ユースケース名 リファクタリング操作

アクター 利用者

概要 利用者はリファクタリングを行う

メインシナリオ

1. 利用者はリファクタリングの目標を設定する
2. 利用者は部分目標を設定する
3. 利用者は基本操作を選択する
4. システムは基本操作を適用した結果を表示する
5. 利用者は部分目標を達成しているか確認する
6. 利用者は目標を達成しているか確認する

サブシナリオ

3.a 部分目標にメトリクス値が設定されている場合
.1 システムはメトリクス値を計算する
.2 利用者は基本操作を選択する
.3 システムは基本操作を適用した結果を表示する
.4 システムはメトリクス値の計算結果を表示する
5 へ進む

5.a 行き詰まり状態に陥った場合
2 へ戻り異なる変更方針を設定する

5.b まだ部分目標が達成されていない場合
3 へ戻り次の操作を適用する

6.a 目標が達成していない場合
2 へ戻り次の目標を設定する

ユースケース名 メトリクス値の計算

アクター 利用者

概要 ソースコードのメトリクス値を計算する

メインシナリオ
1. 利用者は計算したいメトリクス値を選択する
2. システムはメトリクス値の計算結果を表示する

ユースケース名 操作履歴の利用

アクター 利用者

概要 利用者は操作履歴を閲覧し過去に行った操作の結果を得る

メインシナリオ
1. 利用者は操作履歴から変更方針を選択する
2. システムは選択された変更方針の情報を表示する
3. 利用者は変更方針の適用結果を表示するように要求する
4. システムは変更方針をソースコードに適用した結果を表示する

7.2 全体像

本研究では、利用者が行ったリファクタリング操作の履歴を用いてリファクタリングの支援を行うために、以下の機能を持つツールを作成する。ツールの全体像を図 7.2 に示す。

- リソース管理機能
- リファクタリング機能
- 履歴管理機能
- メトリクス計算機能

7.2.1 リソース管理機能

リソース管理機能では、ソースコードや操作履歴を管理する。本研究で作成するツールでは、リファクタリングの対象となるプログラムを一つのプロジェクトとして管理する。プロジェクトは指定されたディレクトリをルートとして、指定されたルート以下にあるすべての Java ファイルが、そのプロジェクトのリソースファイルとなる。リソース管理機能は、新規プロジェクトを作成したときにルート以下のすべての Java ファイルの情報を取得し空の操作履歴を作成する。

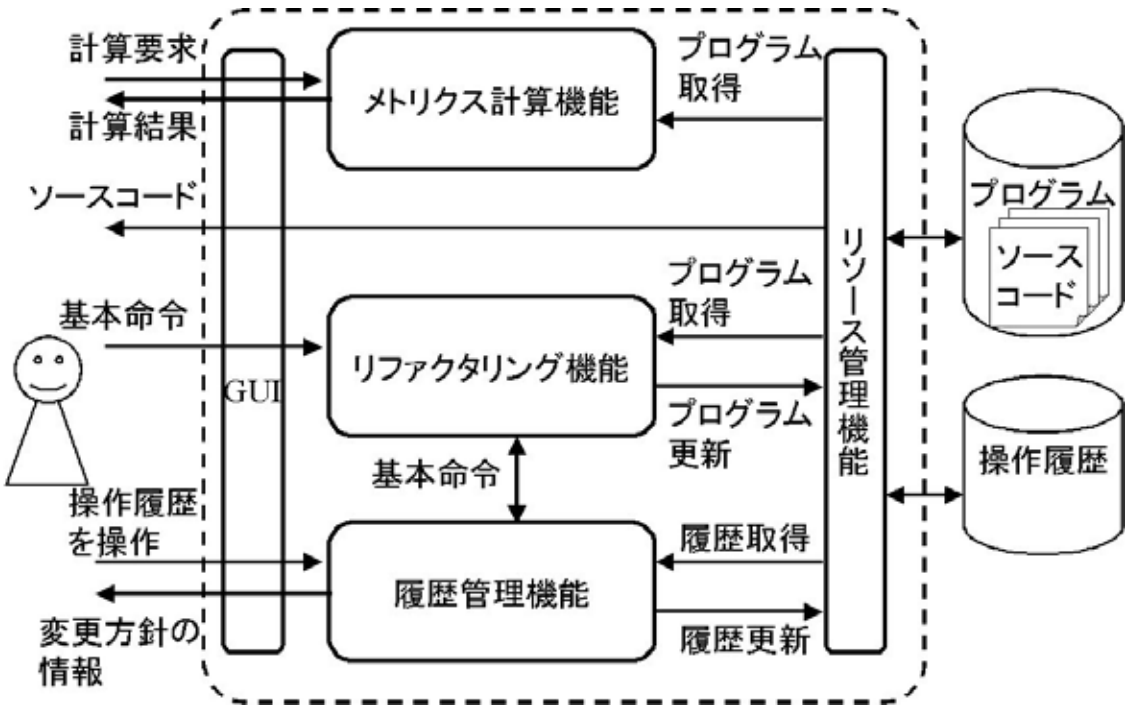

図 7.2: ツールの全体像

7.2.2 リファクタリング機能

この機能は入力された基本命令を判別して、リファクタリング操作をプログラムに対して適用する。基本命令は利用者から入力される場合と履歴管理機能から入力される場合がある。利用者から基本命令が入力された場合はリソース管理機能からソースコードを取得してリファクタリングを適用し、その結果をリソース管理機能に返す。これと同時に利用者から入力された基本命令は、履歴管理機能へ渡される。

7.2.3 履歴管理機能

履歴管理機能は、目標の設定、変更方針の追加と削除、変更方針へ基本命令の追加を行う。新しいプロジェクトが作成された場合は、利用者に作成した操作履歴に目標を設定させる。利用者にリファクタリング操作を適用する前に新しい変更方針を作成させ操作履歴に追加する。リファクタリング機能から基本命令を受け取ると変更方針にその基本命令を追加する。目標が達成された段階で利用者の要求に応じて操作履歴の余分な変更方針を削除する。

7.2.4　メトリクス計算機能

メトリクス計算機能は、リソース管理機能からソースコードを取得してメトリクスを計算する。利用者の要求によってメトリクス値を計算してその結果を返す。

第8章　評価

本研究で作成したツールではリファクタリング操作を自動化することで利用者が手動でリファクタリングを行う手間を取り除くことができた。また自動化したリファクタリング操作を形式化することで複数の操作を一つのまとまりとして管理できるようになった。

目標を達成するために部分目標を設定して少しずつリファクタリングを行えるようにした。部分目標とそれを達成するための基本命令の列を一つの組にした変更方針を定義した。利用者は設定した変更方針に従って基本操作をを適用することで少しずつリファクタリングを行えるようにした。利用者に部分目標を設定させることでプログラム中の問題を切り分けることができ、リファクタリングの経験が少なくても次に行う操作の予測がたてやすくなった。

利用者が行ったリファクタリング操作を操作履歴として保存しこの履歴を利用することで試行錯誤を軽減することができた。操作履歴を利用することで操作の後戻りの支援と複数の操作結果の比較がおこなえるようになった。後戻りが容易になったため行き詰まりになった場合でも回復が容易になった。行き詰まりとはそれ以上リファクタリングを適用できない、または適用することで以前に行った操作の効果が失われる状態のことを言う。利用者は複数の変更方針の結果同士を比較することでより目標に近い変更方針を選択できるようになった。また履歴のノードに変更方針を採用したことでこれまでの作業内容の確認が容易に行えるようになり、次の変更方針を立てる手がかりとすることができるようになった。現時点では利用者のリファクタリングを十分支援できているとはいえないが操作履歴を利用することで試行錯誤の過程を軽減できることが確認できた。今後の課題としては履歴に保存されている基本命令を再利用するためにパターン化する方法について調べる必要がある。

リファクタリング操作の適用前と適用後のメトリクス値の計算結果を利用者に提示することで操作の妥当性に関する評価を客観的に行えるようにした。リファクタリング操作の適用結果の評価を支援するためにメトリクス値を利用することの有効性を確認できた。しかしながら現状では簡単なメトリクス値しか計算できないため利用範囲が限定されている。リファクタリング操作の適用結果の評価に使えるメトリクス値を追加することで利用範囲を拡大することが今後の課題としてあげられる。

第9章　おわりに

本研究では、リファクタリングの履歴を用いることでより少ない試行でリファクタリングする手法を提案し支援ツールを作成した。提案した手法では、利用者の行ったリファクタリング操作の過程と結果をその目的と共に操作履歴として保存できるようにした。また操作の自動化のためリファクタリング操作を形式化し、複数の操作の組み合わせを一つの変更方針として管理できるようにした。この操作履歴を利用することで行き詰まりになった変更方針からの回復が容易に行えるようになった。さらに複数の変更方針を同時に実行し、その結果を比較できるようにすることで、目的を早く達成できるようになった。本研究で提案した手法により開発現場にリファクタリングを導入し目標を達成することが容易になり、開発コストの低下が期待できる。

9.1　今後の課題

今後の課題としてはリファクタリングするときに必要になる試行錯誤を軽減するために変更方針をパターン化し再利用できるようにする方法について調べることが課題としてあげられる。またリファクタリング操作の適用結果の評価に使えるメトリクス値を追加することで評価できる範囲を拡大することも課題としてあげられる。

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助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。

最後に、大学院での生活を援助を行ってくれた両親ならびに生活面でお世話になった友人に感謝いたします。

付 録A　リファクタリング操作

A.1　基本操作

- Move Method

  説明 別のクラスにメソッドを移動する。

  操作手順

  1. 移動先でメソッドを作る。
  2. 移動先に移動元のメソッドのコードをコピーする。
  3. 元のメソッドを委譲メソッドにする。

- Move Field

  説明 別のクラスへフィールドを移動する。

  操作手順

  1. 移動先にフィールドを作る。
  2. 移動先に getter/setter を作る。
  3. 移動元のフィールドを移動先の getter/setter に置き換える。

- Move Class

  説明 別のパッケージへクラスを移動する。

  操作手順

  1. クラスを作成する
  2. パッケージを作成する
  3. 元のクラスのメンバをすべてコピーする
  4. 元のクラスを使っているクラスのインポート宣言を置き換える
  5. 元のクラスを削除する

- Extract Method

  説明 メソッドから指定した範囲を新しいメソッドとして抽出する。

操作手順

1. メソッドを作成する。
2. 抽出元するコードをメソッドにコピーする。
3. 抽出元を作成したメソッドに置き換える。

- Extract Class

  説明 指定したクラスから指定したフィールドやメソッドを抽出する。

  操作手順

  1. クラスを作成する。
  2. 元のクラスから新しいクラスへリンクを作る。
  3. フィールドを移動する。
  4. メソッドを移動する。

- Extract Interface

  説明 指定したクラスからインタフェースを抽出する。

  操作手順

  1. 空のインタフェースを作る。
  2. 作成したインタフェースにメソッドを定義する。
  3. 抽出元のクラスでインタフェースを実装する。

- Encapsulate Field

  説明 フィールドをカプセル化する。

  操作手順

  1. getter/setter を作る。
  2. 他のクラスから、このフィールドを使用している部分を getter/setter に置き換える。
  3. フィールドを private にする。

- Self Encapsulate Field

  説明 フィールドを自己カプセル化する。

  操作手順

  1. getter/setter を作る。
  2. このフィールドを使用している部分を getter/setter に置き換える。

3. フィールドを private にする。

- Push Down Method

  説明 指定したクラスにあるメソッドをサブクラスに引き下げる。

  操作手順

  1. サブクラスに指定したメソッドを作成する。
  2. メソッドをコピーする。
  3. 必要に応じてスーパークラスのメソッドはそのままにしておくか、削除するか、abstract にする。

- Pull Up Method

  説明 指定したクラスにあるメソッドをスーパークラスに引き上げる。

  操作手順

  1. スーパークラスに新しいメソッドを作る。
  2. サブクラスのコードをコピーする。
  3. サブクラスのメソッドを削除する。

- Push Down Field

  説明 指定したクラスにあるフィールドをサブクラスに引き下げる。

  操作手順

  1. フィールドをサブクラスに作る。
  2. スーパークラスからそのフィールドを削除する。

- Pull Up Field

  説明 指定したクラスにあるフィールドをスーパークラスに引き上げる

  操作手順

  1. フィールドをスーパークラスに作る。
  2. サブクラスからそのフィールドを削除する。

- Replace Data Value with Object

  説明 指定したフィールドを持つオブジェクトを作る。

  操作手順

  1. 指定されたフィールドを持つクラスを作る。

2. getter/setter を作る。
3. フィールドを初期化するコンストラクタを作る。
4. 元のクラスに作成したクラスを参照するためのフィールドを追加する。
5. 元のクラスの getter/setter を作成したクラスの getter/setter に委譲する
6. 元のクラスのフィールドを作成したクラスに置き換える。

- Replace Type Code with Subclass

  説明 指定したタイプコード毎にサブクラスを作る

  操作手順

  1. タイプコード毎にサブクラスを作る
  2. getter をオーバライドして適切なタイプコードを返すようにする。
  3. ファクトリメソッドを作る (create スーパークラス名 (タイプコード))

- Replace Conditional with Polymorphism

  説明 指定したメソッドの条件文をポリモーフィズムで置き換える

  操作手順

  1. 指定したメソッド (条件文のみ持つ) を、サブクラスでオーバーライドする
  2. 条件文の各アクション部を作成したメソッドにコピーする
  3. コピー元を abstract にする

- Replace Type Code with State/Strategy

  説明 指定したタイプコードを保持する抽象クラスを作り、タイプコード毎にサブクラスを作る。

  操作手順

  1. タイプコードの情報を保持する抽象クラスを作成する。
  2. 作成したクラスにタイプコードをコピーする。
  3. 作成したクラスにタイプコード用の getter(abstract) を作成する。
  4. タイプコード毎にサブクラスを作成する。
  5. サブクラスで getter を実装して適切なタイプコードを返すようにする。
  6. ファクトリメソッド (create スーパークラス名 (タイプコード)) を作成する。
  7. 元のクラスに抽象クラスを参照するためのフィールド作成する。
  8. 元のクラスの getter を作成した抽象クラスの getter に委譲する。
  9. 元のクラスの setter から抽象クラスのファクトリメソッドを呼び出すようにする。

10. 元のクラスからタイプコードを削除する。

- Replace Temp with Query

  説明 一時変数をメソッド呼び出しに置き換える。

  操作手順

  1. 一時変数を使っている部分をメソッド呼び出しに置き換える。
  2. 一時変数を削除する。

- Rename

  説明 パッケージ、クラス、インタフェース、フィールド、変数の名を変更する。

  操作手順

  1. 新しい名前を持つ要素を作成を作成する。
  2. 古い要素を新しい要素に置き換える。
  3. 古い要素を削除する

- Split Loop

  説明 ループを分割する。

  操作手順

  1. ループを分割する。

A.2 基本命令

- Move Method

  識別子 MM

  形式 MM(p1::c1::m1, MMop(”p2::c2”, ”m2”))

  適用箇所 メソッド

  適用実体 第一引数=移動先のクラス、第二引数=移動先で使う新しいメソッド名

- Move Field

  識別子 MF

  形式 MF(p1::c1::f1, MFop(”p2::c2”, ”f2”))

  適用箇所 フィールド

  適用実体 第一引数=移動先のクラス、第二引数=移動先で使う新しいフィールド名

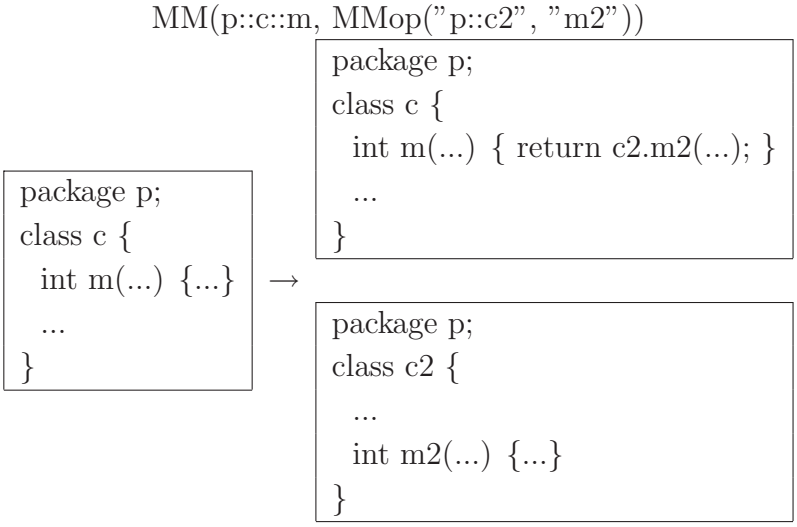

図 A.1: Move Method の例

- Move Class

  識別子 MC

  形式 MC(p1::c1, MCop(”p2”))

  適用箇所 クラス

  適用実体 第一引数=移動先のパッケージ

- Extract Method

  識別子 EM

  形式 EM(p1::c1::m1::[?, ?], EMop(”m2”, [”arg1”, ... ,”argN”], ”returnType”))

  適用箇所 メソッド内の領域

  適用実体 第一引数=新メソッド名、第二引数=引数の列、第三引数=戻り値

- Extract Class

  識別子 EC

  形式 EC(p1::c1, ECop(”c2”, [”f1”, ..., ”fN”, ”m1”, ...,”mM”]))

  適用箇所 クラス

  適用実体 第一引数=新クラス名、第二引数=抽出するメンバの列

- Extract Interface

  識別子 EI

MF(p::c::f, MFop(”c2”, ”f”))

```
package p;
class c {
  int f;
  void setF(int f) {
    this.f = f;
  }
  int getF() {
    return f;
  ...
}
```

→

```
package p;
class c {
  void setF(int f) { c2.setF(f);
  int getF() { return c2.getF(f);
  ...
}
```

```
package p;
class c2 {
  int f;
  public void setF(int f) {
    this.f = f;
  }
  public int getF() {
    return f;
  }
  ...
}
```

図 A.2: Move Field の例

形式 EI(p1::c1, EIop(”i1”, [”m1”, ..., ”mN”]))

適用箇所 クラス

適用実体 第一引数=新インタフェース名、第二引数=抽出するインタフェースの列

- Encapsulate Field

  識別子 EF

  形式 EF(p1::c1::f1, EFop(”getF1”, ”setF1”))

  適用箇所 フィールド

  適用実体 第一引数=getter 名、第二引数=setter 名

- Self Encapsulate Field

  識別子 SEF

  形式 SEF(p1::c1::f1, SEFop(”getF1”, ”setF1”))

  適用箇所 フィールド

MC(p1::c1, MCop(”p2”))

```
package p1;
class c1 {
  ...
}
```

→

```
package p2;
class c1 {
  ...
}
```

図 A.3: Move Class の例

EM(p:c:m:[3, 3]), emop(”m2”, [”int y”], ”int”))

```
package p;
class c {
  int m(int x) {
    int y;
    y = f(x);
    while(y¡10) {
      y += z(x);
    }
  ...
  }
  ...
}
```

→

```
package p;
class c {
  int m(int x) {
    int y;
    y = f(x);
    y = m2(y);
    ...
  }
  int m2(int y) {
    while(y¡10) {
      y += z(x);
    }
    return y;
  }
  ...
}
```

図 A.4: Extract Method の例

適用実体 第一引数=getter 名、第二引数=setter 名

- Push Down Method

  識別子 PDM

  形式 PDM(p1::c1::m1, PDMop([”p2::c2”, ..., ”pN::cM”]))

  適用箇所 メソッド

  適用実体 第一引数=引き下げ先のクラスの列

- Pull Up Method

  識別子 PUM

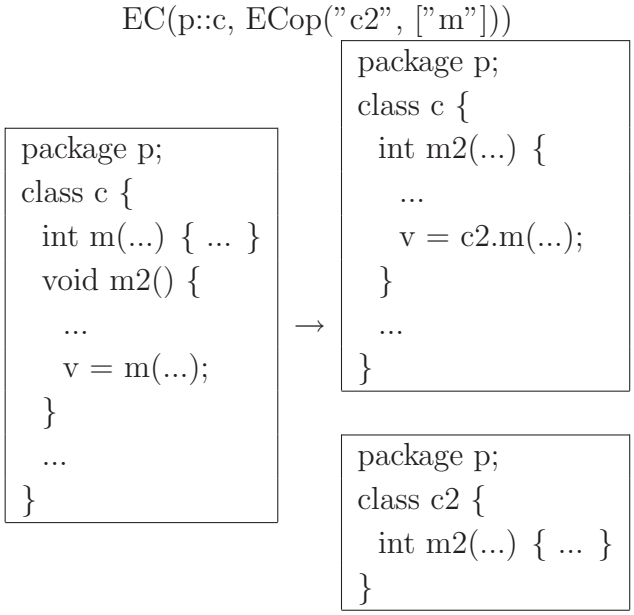

図 A.5: Extract Class の例

形式 PUM(p1::c1::m1, PUMop(”p2::c2”))

適用箇所 メソッド

適用実体 第一引数=引き上げ先のクラス

- Push Down Field

  識別子 PDF

  形式 PDF(p1::c1::f1, PDFop([”p2::c2”, ..., ”pN::cM”]))

  適用箇所 フィールド

  適用実体 第一引数=引き下げ先のクラス

- Pull Up Field

  識別子 PUF

  形式 PUF(p1::c1::f1, PUFop(”p2::c2”))

  適用箇所 フィールド

  適用実体 第一引数=引き上げ先のクラス

- Replace Data Value with Object

  識別子 RDO

EI(p::c, EIop(”i1”, [”m1”, ”m3”]))

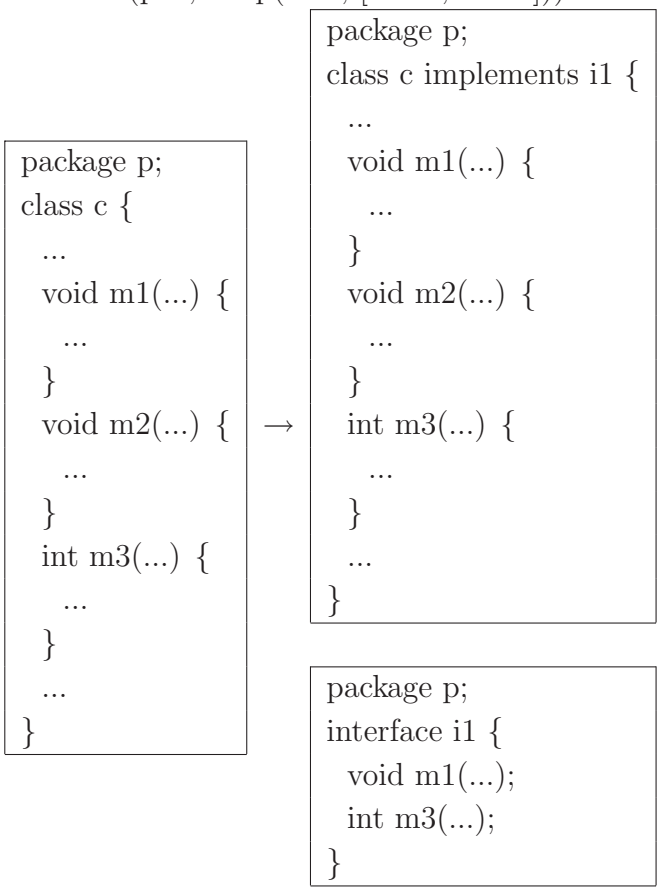

図 A.6: Extract Interface の例

形式 RDO(p1::c1, RDOop(”c2”, [”type1 name1”, ..., ”typeN nameN”], [”initData1”, ..., ”initDataN”]))

適用箇所 クラス

適用実体 第一引数=新クラス名、第二引数=フィールドの列、第三引数=初期値の列

- Replace Type Code with Subclass

  識別子 RTS

  形式 RTS(p1::c1, RTSop([RTSrec(”c2”, ”type1”), ..., RTSrec(”cN”,”typeM”)]))

  適用箇所 クラス

  適用実体 第一引数=クラスとタイプコードを組にしたレコードの列

- Replace Conditional with Polymorphism

  識別子 RCP

EF(p::C1::f, EFop(”getF”, ”setF”))

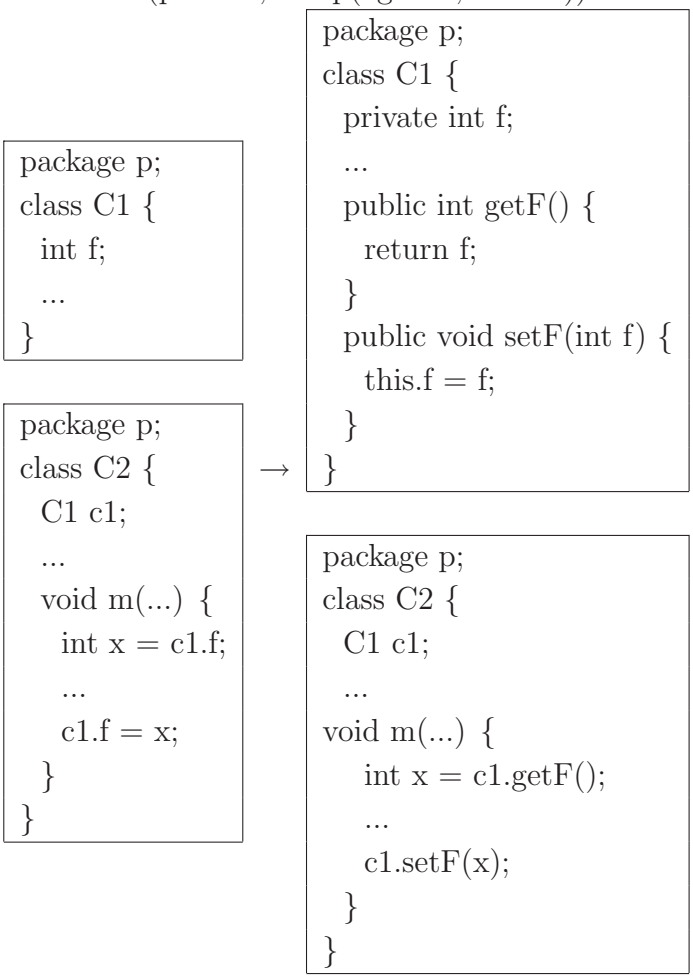

図 A.7: Encapsulate Field の例

形式 RCP(p1::c1::m1, RCPop([RCPrec(”c2”, ”con1”), ..., RCPrec(”cN”, ”conM”)]))

適用箇所 メソッド

適用実体 第一引数=クラスと条件式を組にしたレコードの列

- Replace Type Code with State/Strategy

  識別子 RTSS

  形式 RTSS(p1::c1, RTSSop(”p2::c2”, [RTSSrec(”c3”, ”type1”), ..., RTSSrec(”cN”, ”typeM”)]))

  適用箇所 クラス

  適用実体 第一引数=スーパークラス名、第二引数=クラスとタイプコードを組にしたレコードの列

SEF(p::C::f, SEFop(”getF”, ”setF”))

```
package p;
class C {
  int f;
  ...
  void m1(...) {
    m2(f);
    ...
    f = m3();
  }
}
```

→

```
package p;
class C {
  private int f;
  ...
  void m1(...) {
    m2(getF());
    ...
    setF(m3());
  }
  public int getF() {
    return f;
  }
  public void setF(int f) {
    this.f = f;
  }
}
```

図 A.8: Self Encapsulate Field の例

- Replace Temp with Query

  識別子 RTQ

  PI 形式 RTQ(p1::c1::m1::v1, RTQop())

  適用箇所 一時変数

  適用実体 なし

- Rename

  識別子 RN(仮)

  PI 形式 **1** RN(p1, RNop(”p2”))

  PI 形式 **2** RN(p1::c1, RNop(”c2”))

  PI 形式 **3** RN(p1::i1, RNop(”i2”))

  PI 形式 **4** RN(p1::c1::m1, RNop(”m2”))

  PI 形式 **5** RN(p1::c1::f1, RNop(”f2”))

  PI 形式 **6** RN(p1::c1::m1::v1, RNop(”v2”))

  PI 形式 **6** RN(p1::c1::m1::a1, RNop(”a2”))

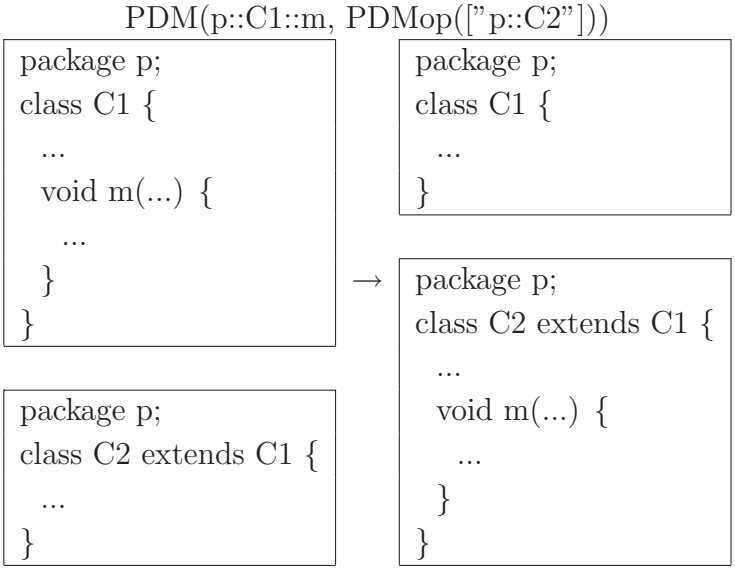

図 A.9: Push Down Method の例

適用箇所 パッケージ、クラス、メソッド、フィールド、変数、引数

適用実体 第一引数=新しい名前

- Split Loop

  識別子 SL

  PI 形式 SL(p1::c1::m1::[?,?], SLop())

  適用箇所 ループ内の領域

  適用実体 なし

PUM(p::C1::m, PUMop(”p::C2”))

```
package p;
class C1 extends C2 {
  ...
  void m(...) {
    ...
  }
}
```

```
package p;
class C2 {
  ...
}
```

→

```
package p;
class C1 extends C2 {
  ...
}
```

```
package p;
class C2 {
  ...
  void m(...) {
    ...
  }
}
```

図 A.10: Pull Up Method の例

PDF(p::C1::f, PDFop([”p::C2”]))

```
package p;
class C1 {
  int f;
  ...
}
```

```
package p;
class C2 extends C1 {
...
}
```

→

```
package p;
class C1 {
  ...
}
```

```
package p;
class C2 extends C1 {
  ...
  int f;
}
```

図 A.11: Push Down Field の例

PUF(p::C1::f, PUFop("p::C2"))

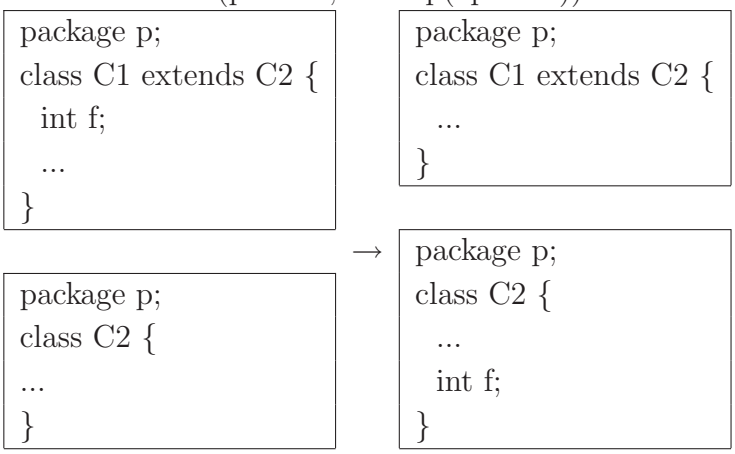

図 A.12: Push Down Field の例

RD(p::c, rdop("R", [int x, int y], [f1(a), f2(b)]))

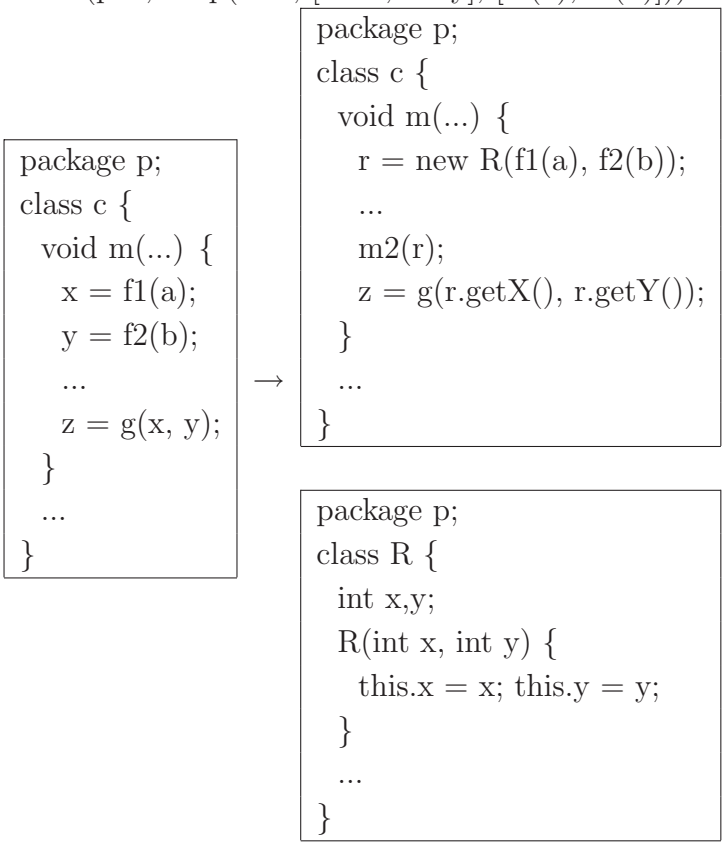

図 A.13: Replace Data Value with Object の例

RTS(p::C0, RTSop([RTSrec(”C1”, ”TYPE1”), RTSrec(”C2”,”TYPE2”)]))

```
package p;
class C0 {
  private int type;
  static final int TYPE1 = 1;
  static final int TYPE2 = 2;
  ...
  public int getType() {
    return type;
  }
}
```

→

```
package p;
class C0 {
  ...
  public abstract int getType();
  public static C0 create(int type) {
    if(type==C1.TYPE1) return new C1();
    else if(type==C2.TYPE2) return new C2();
    else throw new IllegalArgumentException(”...”)
  }
}
```

```
package p;
class C1 extends C0 {
  static final int TYPE1 = 1;
  public int getType() {
    return TYPE1;
  }
}
```

```
package p;
class C2 extends C0 {
  static final int TYPE2 = 2;
  public int getType() {
    return TYPE2;
  }
}
```

図 A.14: Replace Type Code with Subclass の例

RCP(p::C0::m, RCPop([RCPrec(”C1”, ”type==C1.TYPE1”), RCPrec(”C2”, ”type==C2.TYPE2”)]))

```
package p;
class C0 {
  ...
  void m (int type) {
    if(type==C1.TYPE1) {
      処理 1
    } else if(type==C2.TYPE2) {
      処理 2
    } else
    throw new IllegalArgumentException(”...”);
  }
}
```

→

```
package p;
class C0 {
  ...
  abstract void m (int type);
}
```

```
package p;
class C1 extends C0 {
  ...
  void m(int type) {
    処理 1
  }
}
```

```
package p;
class C1 extends C0 {
  ...
}
```

```
package p;
class C2 extends C0 {
  ...
  void m(int type) {
    処理 2
  }
}
```

```
package p;
class C2 extends C0 {
  ...
}
```

図 A.15: Replace Conditional with Polymorphism の例

RTSS(p::C, RTSSop(”p::C0”, [RTSSrec(”C1”, ”TYPE1”), RTSSrec(”C2”, ”TYPE2”)]))

```
package p;
class C {
  static final int TYPE1 = 1;
  static final int TYPE2 = 2;
  private int type;
  ...
  public int getType() {
    return type;
  }
  public void setType(int type) {
    this.type = type;
  }
}
```

→

```
package p;
class C {
  ...
  C0 c0;
  public int getType() {
    return c0.getType();
  }
  public void setType(int type) {
    c0 = C0.create(type);
  }
}
```

```
package p;
abstract class C0 {
  abstract int getType();
  static C0 create(int type) {
    if(type==C1.TYPE1) return new C1();
    else if(type==C2.TYPE2) return new C2();
    else throw new IllegalArgumentException(”...”);
  }
}
```

```
package p;
class C1 extends C0 {
  static final int TYPE1 = 1;
  int getType() {
    return TYPE1;
  }
}
```

```
package p;
class C1 extends C0 {
  static final int TYPE1 = 1;
  int getType() {
    return TYPE1;
  }
}
```

図 A.16: Replace Type Code with State/Strategy の例

RN(p::C::m1, RNop(”m2”))

```
package p;
class C {
  ...
  void m1(...) {
    ...
  }
}
```

→

```
package p;
class C {
  ...
  void m2(...) {
    ...
  }
}
```

図 A.17: Rename の例 (メソッド名の変更)

SL(p::C::m::[8,8], SLop())

```
package p;
class C {
  void m(...) {
    while(iterator.hasNext()) {
      ...
    }
  }
}
```

→

```
package p;
class C {
  void m(...) {
    while(iterator.hasNext()) {
      ...
    }
    while(iterator.hasNext()) {
      ...
    }
  }
}
```

図 A.18: Split Loop の例

参考文献

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

[2] Aqris Software AS, RefactorIT, http://www.refactorit.com/

[3] Eclipse org, Eclipse, http://www.eclipse.org/

[4] 立命館大学理工学部情報科学ソフトウェア基礎技術研究室, Java Refactoring Browser, http://www.fse.cs.ritsumei.ac.jp/refactoring/rise/

[5] Wake, W. C., (長瀬嘉秀, 株式会社テクノロジックアート 訳), リファクタリングワークブック, 株式会社アスキー, 2004

[6] Kerievsky, J., (小黒直樹, 村上歴, 高橋一成, 越智典子訳), パターン指向リファクタリング入門, 日経 BP 社, 2005