

---

# オブジェクト指向ソフトウェア向け リファクタリングツールの開発

丸山 勝久

立命館大学 理工学部 情報学科

2002年2月27日

maru@cs.ritsumeai.ac.jp

# リファクタリングとは

---

既存ソフトウェアの設計の理解性や変更容易性を向上させることを目的とした上で、そのソフトウェアの外部からみた挙動(振る舞い)を変えずに、内部構造を再構成すること

## リファクタリングの必要性

- 将来のクラスやクラス間の関係を完全には予測不可能  
⇒ 再利用率の低下 , 設計作業の高コスト化
- 保守や改良により設計が劣化  
⇒ 理解性や変更(カスタマイズ)容易性の低下



リファクタリングによる再利用率 , 保守性の維持・向上

# リファクタリングの例(メソッドの移動)

```
class Customer ...
public statement() {
    ...
    Iterator it = _rentals.iterator();
    while (rentals.hasNext()) {
        Rental each = (Rental)it.next();
        double thisAmount = amountFor(each);
        ...
    }

    private double amountFor(Rental aRental) {
        ...
        switch (aRental.getMovie().getPriceCode()) {
            ...
            case Movie.NEW_RELEASE:
                result += aRental.getDaysRented() * 3;
            ...
        }
    }
}
```



```
class Customer ...
public statement() {
    ...
    Iterator it = _rentals.iterator();
    while (rentals.hasNext()) {
        Rental each = (Rental)it.next();
        double thisAmount = each.amountFor();
        ...
    }
}
```

amountFor  
の移動

```
class Rental ...
public double amountFor() {
    ...
    switch (getMovie().getPriceCode()) {
        ...
        case Movie.NEW_RELEASE:
            result += getDaysRented() * 3;
        ...
    }
}
```

# リファクタリングの実現

---

大きな(複雑な)設計変更を一連の小さな変換により実現

小さな変換が変換前後でソフトウェアの挙動を保存

⇒ 大きな設計変更がソフトウェアの挙動を保存

リファクタリングの安全性

Opdyke:

- 26の基本変換と3つの複合変換(基本変換の組合せ)に対して, 変換前に成立する前提条件(precondition)と変換操作を定義
- 変換前後で挙動が保存されることを保証(安全)

Fowler:

- 72の変換に対して, 名前・動機・手順・例などのカタログ化
- 変換前後で挙動が保存されることを非保証(安全でない)  
(プログラマが変換後にテストを行うことで保証)

## メソッド移動の操作例

```
class Customer ...
```

同一メソッドが移動先クラス(含祖先クラス)に存在するか?

```

Iterator it = _rentals.iterator();
while (rentals.hasNext()) {
    Rental each = (Rental)it.next();
    double thisAmount = amountFor(each);
}

```

## 移動メソッドの呼び出しか？

```
class Customer ...
```

Iterator it = \_rentals.iterator();  
 while (it.hasNext()) {  
 Rental each = (Rental)it.next();  
 double thisAmount = each.amountFor();  
 }

```
private double amountFor(Rental aRental) {
    ...
    switch (aRental.getMovie().getPriceCode()) {
        ...
        case Movie.NEW_RELEASE:
            result += aRental.getDaysRented() * 3;
        ...
    }
}
```

## 委譲/転送メソッドが作成可能か？

## 移動先クラスのオブジェクトか？

class Rental ...

オブジェクトか?

## 移動元クラスのメソッド/フィールドを利用していないか?

# リファクタリングの自動化

---

- 手動リファクタリングは安全か? 簡単か?
  - 誤りが混在しやすい
  - プログラマに多くの検査を強要
- いつ,どのコードに,どの変換を適用するのか?
  - 適用に関する指針が不十分
  - 結果や効果は変換の適用順序に大きく依存



## リファクタリングを自動化する手法・ツールが必須

- 抽象構文木におけるノードの追加,削除,書き換え
- プログラム解析 (フロー解析,依存解析)に基づく変換
- 履歴を用いたリファクタリング計画の提示

# JRB (Java Refactoring Browser)

JRB

改善前の  
Java ソースコード

```
public class Tile {
    public int posX, posY;

    Tile(int x, int y, Color c) {
        ...
    }

    public void paint(Graphics g, int offsetX, int offsetY) {
        int left = posX * size + offsetX;
        g.setColor(color);
        g.fillRect(left + border, top + ... );

        g.setColor(Color.white);
        for (int i = 0; i < border; i++) {
            g.drawLine(left + i, top + i, right - i, top + i);
            g.drawLine(left + i, top + i, left + i, bottom - i);
        }

        g.setColor(Color.black);
        g.drawRect(left, top, size - 1, size - 1);
    }
}
```

リファクタリング機能  
(メニュー)



```
public class Tile {
    public int posX, posY;

    Tile(int x, int y, Color c) {
        ...
    }

    public void paint(Graphics g, int offsetX, int offsetY) {
        paintBackgroundGraphics g);
        eraseTile();
        paintTitle();
    }

    public void paintBackground(Graphics g, int x, int y) {
        ...
    }

    public void paintTile(Graphics g, int x, int y) {
        ...
    }

    public void eraseTile(Graphics g, int x, int y) {
        ...
    }
}
```

改善後の  
Java ソースコード

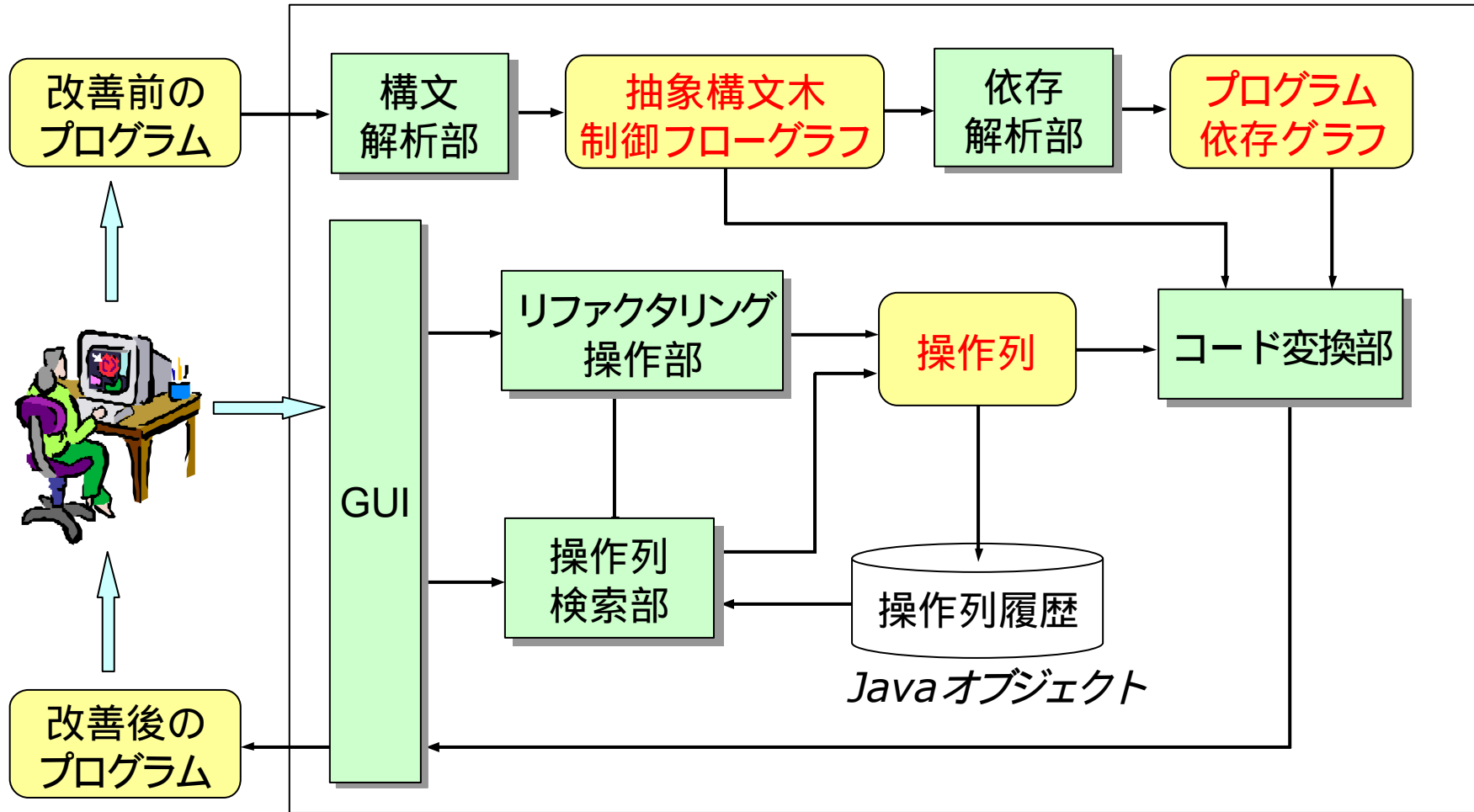
プログラム編集機能  
(エディタ)



プログラマ

# JRBの構成

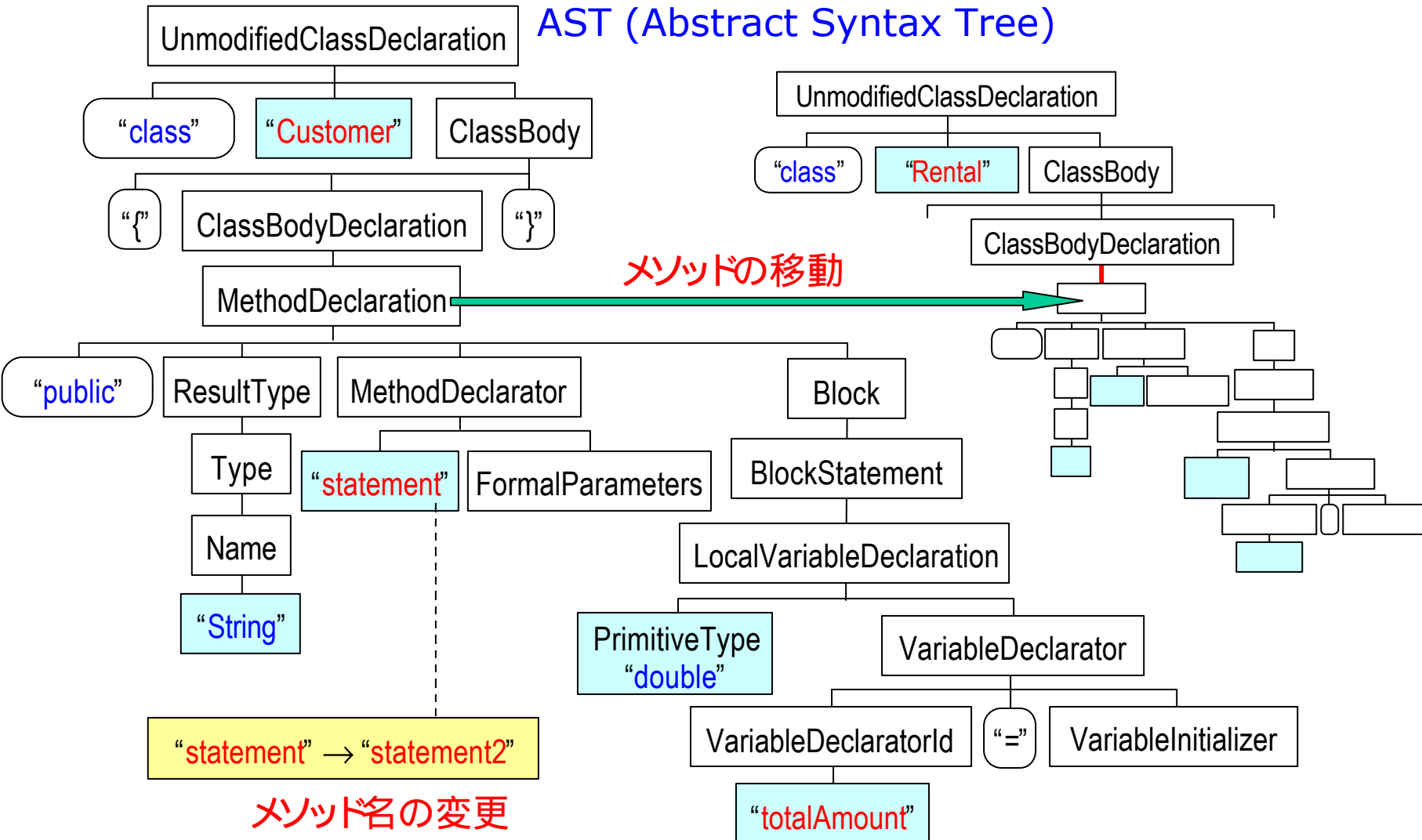
JRB





## 抽象構文木(AST)

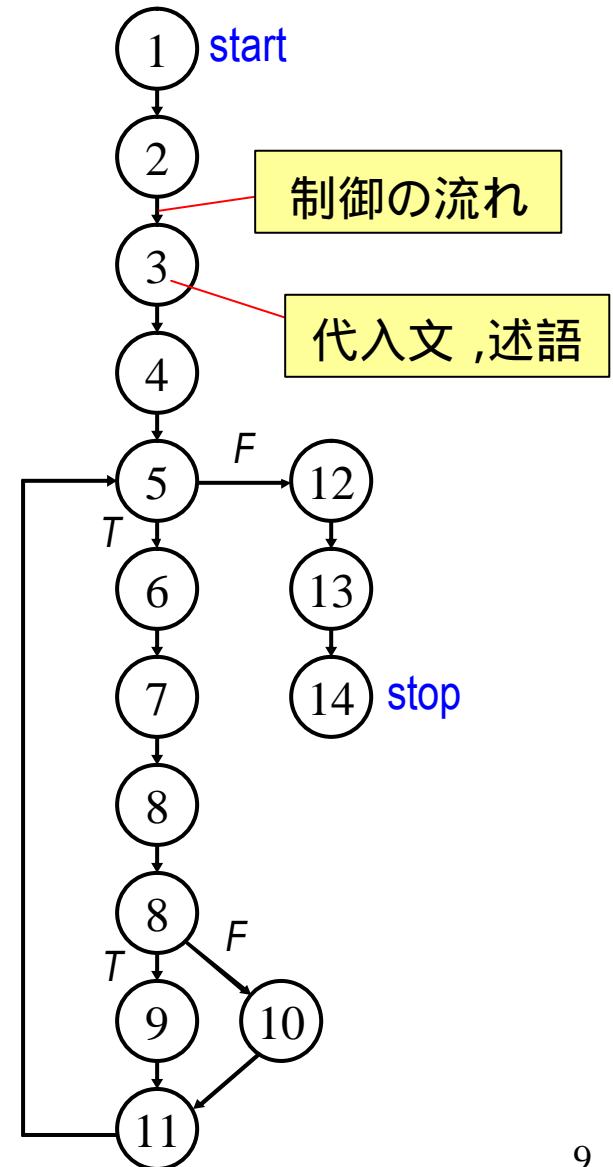
## AST (Abstract Syntax Tree)



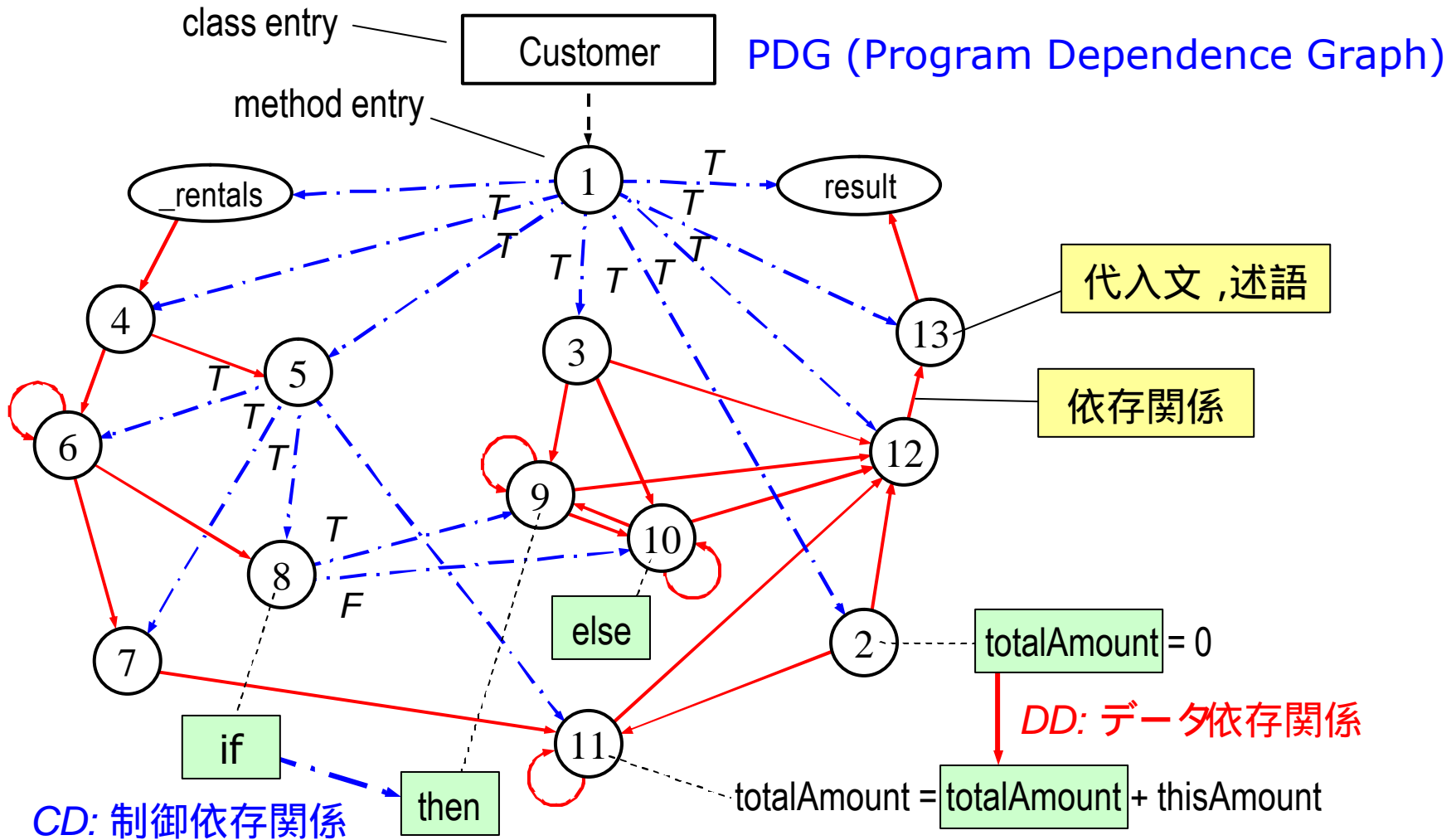
# 制御フローグラフ(CFG)

```
class Customer {  
    ...  
[1]   public String statement() {  
[2]       double totalAmount = 0;  
[3]       int renterPoints = 0;  
[4]       Iterator rentals = _rentalsiterator();  
[5]       while (rentals.hasNext()) {  
[6]           Rental each = rentals.next ();  
[7]           double thisAmount = each.getCherge();  
[8]           if (each.getMovie().getPriceCode() == ... )  
[9]               renterPoints = renterPoints + 2;  
[10]          else  
[11]              renterPoints++;  
[11]          totalAmount = totalAmount + thisAmount;  
[12]      }  
[12]      String result = "Amount: " + String.valueOf(totalAmount)  
[12]          + "Points: " + String.valueOf(renterPoints);  
[13]      return result;  
[13]  }  
}
```

## CFG (Control Flow Graph)



# プログラム依存グラフ(PDG)

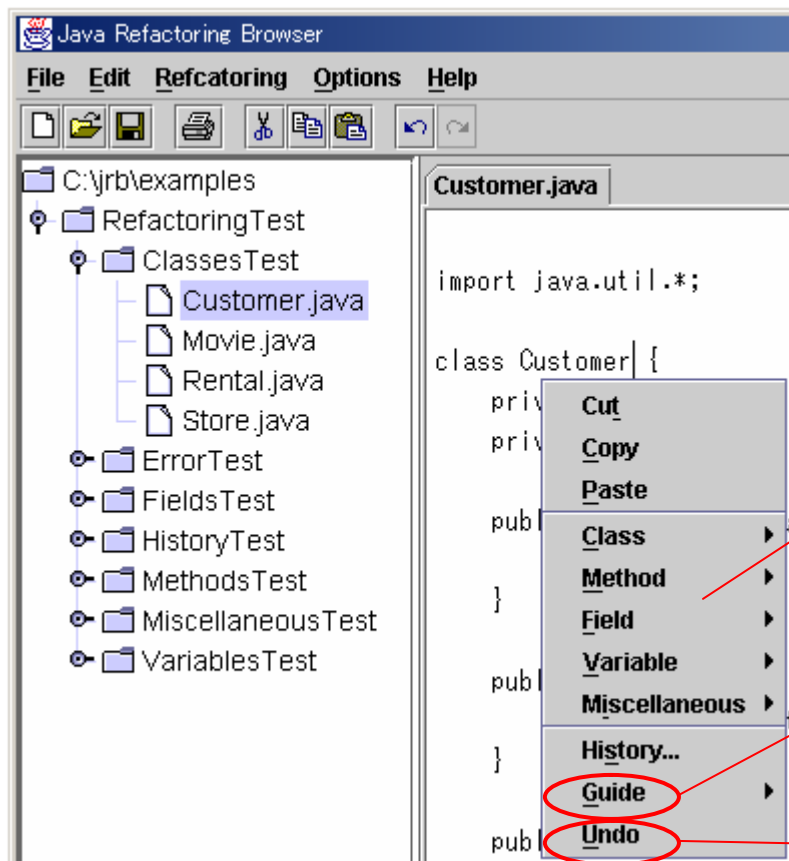


# 実装 (Java program)

| Package                           | # of files |
|-----------------------------------|------------|
| parser (including “.jjt”)         | 7          |
| parser.summary (including “.jjt”) | 7          |
| model                             | 8          |
| graphs                            | 1          |
| graphs.cfg                        | 18         |
| graphs.pdg                        | 13         |
| graphs.slice                      | 1          |
| graphs.util                       | 7          |
| refactor                          | 8          |
| refactor.classes                  | 18         |
| refactor.methods                  | 10         |
| refactor.fields                   | 14         |
| refactor.variables                | 6          |
| refactor.miscellaneous            | 2          |
| refactor.dialog                   | 14         |
| refactor.util                     | 5          |
| gui                               | 29         |
| total                             | 168        |

“.jjt” is a JavaCC file.

# リファクタリングメニュー



## (1) リファクタリング操作

- Classメニュー (8種類)
- Methodメニュー (5種類)
- Fieldメニュー (7種類)
- Variableメニュー (3種類)
- Miscellaneousメニュー (1種類)

## (2) 次操作の提案(Guide)

## (3) 取り消し(Undo)

# リファクタリング操作

---

- Classメニュー

- (1) Rename Class
- (2) Move Class
- (3) Merge Class
- (4) Delete Class
- (5) Extract Subclass
- (6) Extract Superclass
- (7) Extract Super Interface
- (8) Extract Interface

- Methodメニュー

- (1) Rename Method
- (2) Move Method
- (3) Delete Method
- (4) Pull Up Method
- (5) Push Down Method

- Fieldメニュー

- (1) Rename Field
- (2) Move Field
- (3) Delete Field
- (4) Pull Up Field
- (5) Push Down Field
- (6) Encapsulate Field
- (7) Self Encapsulate Field

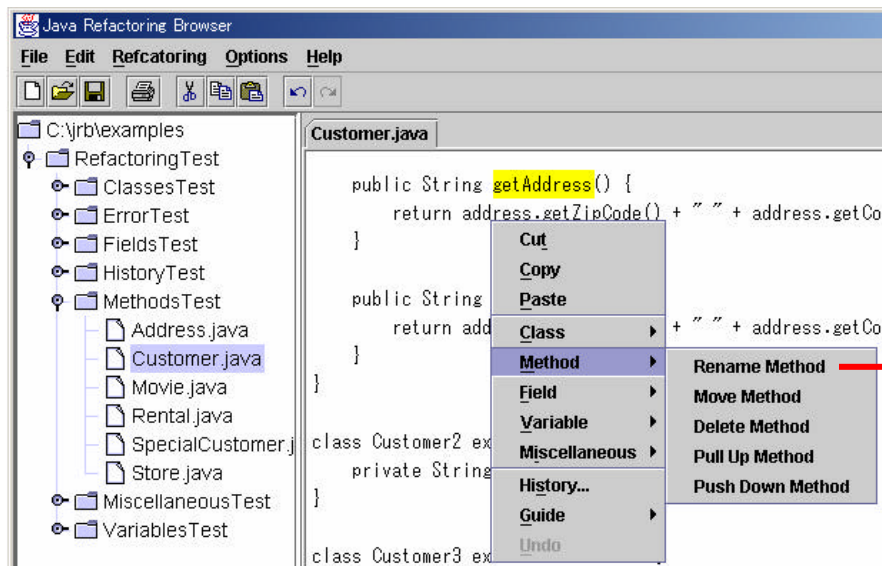
- Variableメニュー

- (1) Rename Variable
- (2) Delete Variable
- (3) Slice on Variable

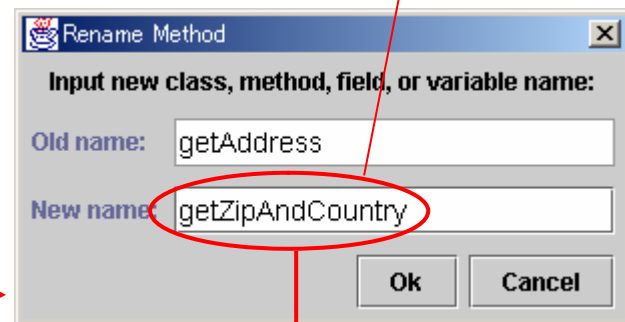
- Miscellaneousメニュー

- (1) Switch to Polymorphism

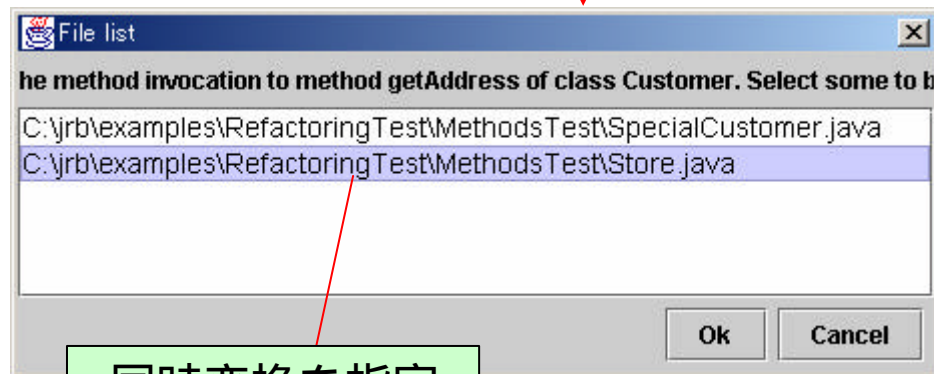
# 実行例(Rename Method)



変更後の名前の入力



変換メソッドへの呼出しを含むファイルの一覧



同時変換を指定

# 実行例(Cont.)

Confirm Changes

Confirm changes made to the following texts:

Customer.java SpecialCustomer.java

Before

```
protected Address address;

public String getAddress() {
    return address.getZipCode() + " " + address.getCountry();
}

public String getAddress2() {
    return address.getZipCode() + " " + address.getCountry();
}

class Customer2 extends Customer {
    private String _name;
}

class Customer3 extends Customer2 {

    public String getName() {
        return "***" + getName() + getAddress();
    }
}
```

After

```
public String getZipAndCountry() {
    return address.getZipCode() + " " + address.getCountry();
}

public String getAddress() {
    return getZipAndCountry();
}

public String getAddress2() {
    return address.getZipCode() + " " + address.getCountry();
}

class Customer2 extends Customer {
    private String _name;
}

class Customer3 extends Customer2 {

    public String getName() {
        return "***" + getName() + getZipAndCountry();
    }
}
```

名前の変更

転送メソッド

変換メソッドの呼出し

Customer.java

Store.java

Before

```
Iterator it = customers.iterator();
while (it.hasNext()) {
    Customer cus = (Customer)it.next();
    System.out.print(cus.getName() + " : ");
    System.out.print(cus.getAddress());
}
```

After

```
Iterator it = customers.iterator();
while (it.hasNext()) {
    Customer cus = (Customer)it.next();
    System.out.print(cus.getName() + " : ");
    System.out.print(cus.getZipAndCountry());
}
```

Ok Cancel

Ok Cancel





## 移動先クラスの入力



## クラスの 一覽

# 実行例(Cont.)

Confirm changes made to the following texts:

Customer.java Rental.java

**Before**

```
private double amountFor(Rental aRental) {
    double result = 0;
    switch (aRental.getMovie().getPriceCode()) {
        case Movie.REGULAR:
            result += 2;
            if (aRental.getDaysRented() > 2)
                result += (aRental.getDaysRented() - 2) * 1.5;
            break;
        case Movie.NEW_RELEASE:
            result += aRental.getDaysRented() * 3;
            break;
        case Movie.CHILDRENS:
            result += 1.5;
            if (aRental.getDaysRented() > 3)
                result += (aRental.getDaysRented() - 3) * 1.5;
            break;
    }
    return result;
}
```

**After**

```
return result;

public double amountFor(Rental aRental) {
    return aRental.amountFor(aRental);
}
```

protected Address address;

```
public String getAddress() {
    return address.getZipCode() + " " + address.getCountry();
}

public String getAddress2() {
    return address.getZipCode() + " " + address.getCountry();
}

class Customer2 extends Customer {
    private String _name;
}
```

転送メソッド

移動先クラスのオブジェクト

メソッドの移動

Customer.java

Rental.java

Confirm changes made to the following texts:

Customer.java Rental.java

**Before**

```
public Movie getMovie() {
    return _movie;
}
```

**After**

```
public Movie getMovie() {
    return _movie;
}

public double amountFor(Rental aRental) {
    double result = 0;
    switch (this.getMovie().getPriceCode()) {
        case Movie.REGULAR:
            result += 2;
    }
}
```

C:\jrb\examples\RefactoringTestMethodsTest\Customer.java

C:\jrb\examples\RefactoringTestMethodsTest\Rental.java

Ok Cancel

# 次操作の提案(Guide)

The screenshot illustrates the Eclipse IDE's refactoring tools. The **History** window on the left lists a sequence of operations: `RenameMethod:src -> MoveMethod:src -> DeleteMethod:src`, `MoveMethod:dst`, `RenameVariable:src -> MoveMethod:src -> DeleteMethod:src -> RenameMethod:dst`, `MoveMethod:dst -> DeleteVariable:src -> RenameMethod:src -> DeleteMethod:src`, and `RenameMethod:src`. Red circles highlight `RenameMethod:src -> DeleteMethod:src` and `RenameMethod:dst`. A box labeled **一致** (Consistent) points to the `RenameMethod:src -> DeleteMethod:src` entry. A yellow box labeled **MoveMethod DeleteMethod ?** points to the `MoveMethod:src -> DeleteMethod:src` entry. A green box labeled **直前(1つ前)の操作** (Previous operation) points to the `MoveMethod:src` entry. Another green box labeled **2つ前の操作** (Operation 2 steps back) points to the `MoveMethod:src` entry. The **Java Refactoring Browser** window on the right shows a project tree with `Customer.java`, `Movie.java`, and `Rental.java` selected. A context menu is open over the code, with the **RenameMethod:dst** option circled in red. A box labeled **リファクタリング履歴** (Refactoring History) is positioned above the browser window.

History

RenameMethod:src -> MoveMethod:src -> DeleteMethod:src  
MoveMethod:dst  
RenameVariable:src -> MoveMethod:src -> DeleteMethod:src -> RenameMethod:dst  
MoveMethod:dst -> DeleteVariable:src -> RenameMethod:src -> DeleteMethod:src  
RenameMethod:src

一致

MoveMethod  
DeleteMethod  
?

直前(1つ前)の操作

2つ前の操作

リファクタリング履歴

Java Refactoring Browser

File Edit Refactoring Options Help

C:\jrbjrt\testcases\Refactori

Customer.java Movie.java Rental.java

import java.util.\*;

cl

Cut  
Copy  
Paste  
Class  
Method  
Field  
Variable  
Miscellaneous  
History...  
Guide  
Undo

RenameMethod:dst

Empty

# 取り消し(Undo)

取り消し前

RenameMethod:src    MoveMethod:src

同時に行った  
メソッド移動操作

MoveMethod:dst

Undo

取り消し後

RenameMethod:src

無矛盾

(Empty)

RM    MoveMethod:src

同時に行った  
メソッド移動操作

MoveMethod:dst    MoveField:src

同時に行った  
フィールド移動操作

MoveField:dst

~~Undo~~

RM

無矛盾

(Empty)

矛盾

MoveField:dst

Cut  
Copy  
Paste

Class  
Method  
Field  
Variable  
Miscellaneous

History...  
Guide

Undo

選択不可

# 関連研究

---

## リファクタリング自動化ツール:

- Smalltalk Refactoring Browser [Roberts and Brant]
- JRefactory [Seguin and Li]
- jFactory [Instantiations, Inc.]
- Transmogrify [open source]
- JBuilder6 [Borland Software Corp.]

## リファクタリング計画(refactoring plan):

- “不吉な匂い”(bad smells) [Fowler]
- プログラム構造 [Opdyke]
- デザインパターン(design patterns) [Tokuda]

# まとめ

---

## Javaプログラム向けリファクタリングツールの開発 (JRB: Java Refactoring Browser)

- (1) 24種類のリファクタリング操作の自動化
  - 抽象構文木における ノードの追加 ,削除 ,書き換え
  - プログラム解析 (フロー解析 ,依存解析 )に基づく変換
- (2) 履歴を用いたリファクタリング計画(次操作)の提示
- (3) リファクタリング操作の取り消し(Undo)



リファクタリング作業の負担軽減

# *Thank you!*

Contact:

丸山 勝久

立命館大学 理工学部 情報学科

Katsuhisa Maruyama

Dept. of Computer Science

Ritsumeikan Univ.

[refactoring@fse.cs.ritsumei.ac.jp](mailto:refactoring@fse.cs.ritsumei.ac.jp)

<http://refactoring.fse.cs.ritsumei.ac.jp>