

契約番号 01第003号

**高度情報化支援ソフトウェアシーズ育成事業**

**「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」**

**A Tool Support for Refactoring Object-Oriented Software**

**論 文**

平成14年 1月

立命館大学

丸山 勝久 ( Katsuhisa Maruyama )

立命館大学 理工学部

〒525-8577 滋賀県草津市野路東 1-1-1

E-mail : maru@cs.ritsumeai.ac.jp

<製品に対する商標表示>

UNIX は、X/Open Company Limited を通じて独占的にライセンスされた登録商標です。

Java, Solaris は、米国 Sun Microsystems, Inc.の米国およびその他の国における登録商標です。

Windows は、米国 Microsoft, Corp の米国およびその他の国における登録商標です。

Pentium は、米国 Intel, Inc の米国およびその他の国における登録商標です。

JBuilder は米国 Borland Software Corp における登録商標です。

その他、本文中に現れる商品名などは、各販売元または開発メーカーの登録商標または製品です。なお、本文中ではTM マークなどは明記していません。

## 和文概要

オブジェクト指向プログラムの有するカプセル化，継承，多相性などの特徴は，ソフトウェアの再利用性や保守性を向上させるといわれている．しかしながら，オブジェクト指向に基づき，これらの特徴を十分に生かしたソフトウェアを設計することは難しい．また，たとえソフトウェアが初期段階においてうまく設計されていたとしても，ソフトウェア自体の保守や改良により設計が劣化し，再利用性や保守性が低下することがある．このような状況に対して，リファクタリングが有効である．リファクタリングとは，ソフトウェアの外部から見た挙動（振る舞い）を保存したまま，内部の構造だけを改良し，その設計を向上させる作業を指す．しかしながら，リファクタリング操作は一般的に煩雑であり，手動で行う際には変換時に誤りが混入しやすい．特に，リファクタリングの適用に関しては，対象ソースコードに対して多くの前提条件を検査すること，および，変換後のソースコードが影響を及ぼす箇所を特定することが非常に面倒であり，実際の開発現場では有用性が認められているにもかかわらず，敬遠されることが多い．

本研究開発では，オブジェクト指向プログラムに対する制御フロー解析と依存解析を導入することにより，リファクタリング操作の自動化を達成する．これにより，従来手動で行ってきた前提条件の検査や変更影響範囲の特定を含めたテストから，プログラムを解放することができる．本論文では，プログラム解析に基づくリファクタリングの自動化手法，および，その手法を実装したリファクタリング支援ツールの機能および適用例を示す．実装したツールは，ソースコードに対する編集作業と並行してメニューから簡単にリファクタリング操作が実行できるように設計されており，ソフトウェア開発におけるリファクタリングの効果を十分実感できるようになっている．また，本ツールには，プログラムの過去のリファクタリング操作を履歴として蓄積し，プログラムの現在の操作に応じて随時検索を行うことで，次に行うべき操作を提案する仕組みが組み込まれている．ソフトウェアの改善は一般的に難しい作業であるため，リファクタリング操作の自動化および次操作の提案は，ソフトウェア開発支援という観点から意義は大きい．本ツールを用いた評価実験の結果，手動で行うリファクタリングに比べて，利用者の負担，特に変換前の前提条件の検査およびソースコード書き換えの負担を，大幅に削減できることが確認できた．今後は，本ツールの有用性を評価するためにさまざまなソースコードを用いて数多くの実験を行う予定である．



## **ABSTRACT**

Object-oriented programming with encapsulation, inheritance, and polymorphism has potential for improving software reusability and maintainability, but designing reusable and understandable object-oriented software is hard. Even if software is well designed initially, maintenance and extension will tend to cause its design to deteriorate. Refactoring is the essential and useful process since it improves the internal structure of existing source-code without changing its external behavior. However, it is complex to do by hand. Moreover, the manual refactoring requires programmers to make many checks. For example, does the target source-code satisfy the preconditions for transformation that the programmer wants? Which fragments of source-code will be affected by the fragments that he/she modifies in refactoring? The refactoring with such checks is not easy in general; it is error-prone and time-consuming.

The tool we have developed automates the transformation in refactoring by using data- and control-flow analysis (and dependency analysis) for object-oriented software. With this tool, programmers do not have to test the changed source-code since the tool guarantees the target source-code to preserve its behavior. This paper describes a method that automates the tests and transformations in refactoring and the tool implementing the method, and gives the examples and experimental results of refactoring using the tool. The tool was designed to easily execute refactorings that programmers want in course of editing the source-code. In addition, the tool records the history of refactoring in the past, which is a sequence of refactoring names, and aggressively uses the history to propose the next refactoring to be done. Experimental results show that the automated refactoring and the guidance using refactoring history are effective in object-oriented software development. A large number of experiments with various source-codes for assessing the utility of the tool will be tackled.

## 目 次

はじめに .....	1
1 . 背景 .....	2
2 . 目的 .....	3
3 . 技術開発の内容 .....	4
3.1 技術開発項目の全体像 .....	4
3.2 リファクタリングの自動化（技術開発項目 1） .....	4
3.2.1 技術課題 .....	4
3.2.2 活用する理論・技術 .....	5
3.2.3 現状の到達度（過去の実績度） .....	9
3.3 リファクタリング計画の提示（技術開発項目 2） .....	9
3.3.1 技術課題 .....	9
3.3.2 活用する理論・技術 .....	10
3.3.3 現状の到達度（過去の実績度） .....	10
4 . 開発・実装システム内容 .....	11
4.1 研究開発システムの全体像 .....	11
4.2 JAVA 言語 .....	11
4.3 機能及びその構成概要 .....	12
4.3.1 リファクタリング部 .....	12
4.3.2 ユーザインタフェース(GUI)部 .....	14
4.4 システムの特徴 .....	14
4.5 稼動環境 .....	14
4.6 インタフェース仕様 .....	15
4.6.1 ユーザとシステム間のインタフェース .....	15
4.6.2 操作列検索部と履歴ライブラリとのインタフェース .....	19
5 . 実装と適用実験 .....	20
5.1 実装したリファクタリング操作 .....	20
5.1.1 クラスリファクタリング(Class Refactoring) .....	21
5.1.2 メソッドリファクタリング(Method Refactoring) .....	22
5.1.3 フィールドリファクタリング(Field Refactoring) .....	23
5.1.4 ローカル変数リファクタリング(Variable Refactoring) .....	25
5.1.5 その他リファクタリング(Miscellaneous) .....	25

5.1.6	リファクタリングの取り消し(Undo)	25
5.1.7	リファクタリングに関する次操作の提案	26
5.2	評価実験	27
6	評価・考察	32
6.1	リファクタリングの自動化に関して	32
6.2	リファクタリング計画の提示に関して	32
6.3	開発ツールの実装に関して	33
7	今後の課題	34
8	参考文献	35
	おわりに	36

## はじめに

オブジェクト指向プログラムの有するカプセル化，継承，多相性などの特徴は，ソフトウェアの再利用性や保守性を向上させるといわれている．しかしながら，オブジェクト指向に基づき，これらの特徴を十分に生かしたソフトウェアを設計することは難しい．

本研究開発では，再利用可能なソフトウェアをはじめから作成することは困難であるという立場から，既存のソフトウェアの再利用性や保守性を改善するリファクタリング技術に着目し，この操作の自動化手法の確立を目指す．リファクタリングとは，ソフトウェアの外部から見た挙動（振る舞い）を保存したまま，内部の構造だけを改良する作業を指す．自動化されたリファクタリングをソフトウェア開発に導入することで，次に示す利点が得られる．

- （１）自動化されたリファクタリング操作は，プログラムの変換前後の挙動を必ず保存する．  
よって，プログラマはリファクタリング後のプログラムに対するテストを回避することができる．
- （２）自動リファクタリングにより，再利用性や保守性を後で簡単に改善することができる．  
よって，ソフトウェア開発の初期段階において設計が完全でないことが許され，ソフトウェア設計に不必要に時間やコストを費やすことが避けられる．
- （３）あらかじめ再利用可能なソフトウェアを作成しておくのではなく，必要時にリファクタリングにより再利用ソフトウェア（部品）を作成することができる．よって，ソフトウェアに対する要求に柔軟に対応することが可能となる．

本研究開発では，プログラムの制御フロー解析および依存解析技術を取り入れることで，リファクタリング操作の自動化を達成し，リファクタリングツールを構築した．構築したツールは，実際のソフトウェア開発環境での利用を想定し，リファクタリング操作をソースコードの編集作業と並行して実行できる環境を提供する．本論文では，プログラム解析に基づくリファクタリングの自動化手法，および実装したリファクタリング支援ツールの機能および適用例を示す．

## 1. 背景

再利用可能なオブジェクト指向ソフトウェアを設計することは難しい。なぜなら、設計者は適切な属性（変数）と操作（メソッド）を持つオブジェクト（クラス）、および、それらの関係をそのソフトウェアが再利用される前にあらかじめ決定しておかなければならないからである。また、たとえソフトウェアが初期段階においてうまく設計されていたとしても、ソフトウェア自体の保守や改造により、その設計が劣化することがある。特に、複数のプログラマが、さまざまな要求を満たすようにソフトウェアを変更した場合、設計（あるいはソースコード）の理解性や変更容易性は悪化することが多い。既存ソフトウェアの再利用性や保守性を維持するためには、そのソフトウェアの外部から見た挙動（振る舞い）を変えずに、内部構造だけを再構成（改良）するリファクタリング[1][2]が重要である。

オブジェクト指向ソフトウェア開発におけるリファクタリング作業の位置づけを図1に示す。リファクタリングは、従来の分析？設計？実装の流れに対して、実際に移動しているプログラム（ソースコード）を積極的に変換することで、ソフトウェアの再設計を行い、再利用性や保守性を改善する作業である。

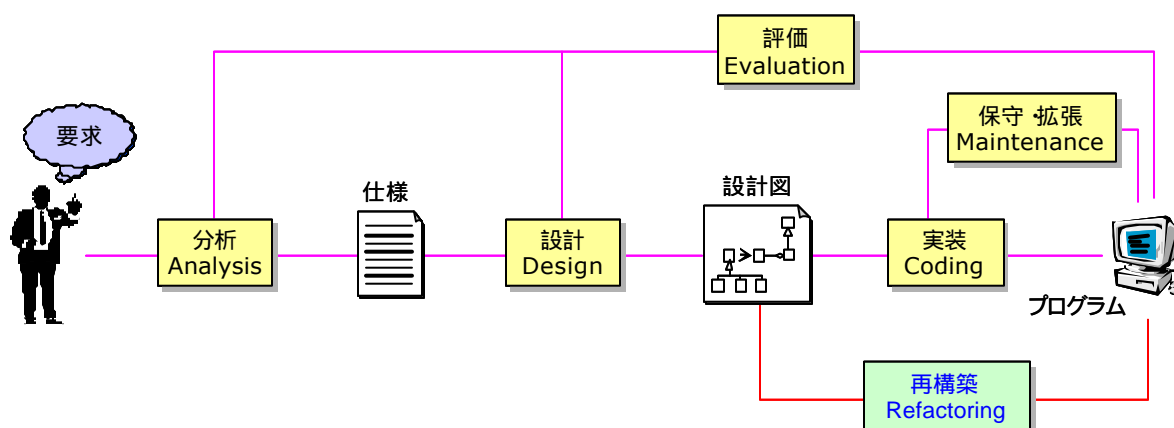


図1 ソフトウェア開発におけるリファクタリング作業の位置づけ

リファクタリングの有効性は実際のソフトウェア開発において従来から指摘されているが、その適用に関する最大の障壁は自動化ツールが存在しないことである。リファクタリングでは、ソフトウェアの挙動を保存することが必須であり、このために大きな設計変更を一連の小さな変換によって達成する。小さな変換が変換前後でソフトウェアの挙動を保存すれば、一連の大きな変更もその挙動を保存することが保証される。しかしながら、たとえリファクタリングが小さな変換の繰返しであっても、実際のソフトウェア開発において、その操作は手動（人手）で行うには煩雑すぎる。よって、適用時に誤りが混入しやすく、このため適用後に多くのテストをプログラマに強要することになる。さらに、リファクタリング自体は新規にソフトウェアを作り出す作業ではないため、その作業は敬遠されることが多い。

## 2 . 目的

本研究開発では、プログラムの負担を増加させずに、高品質なソフトウェアの開発および既存ソフトウェアの保守ができることを目標として、リファクタリング操作を自動化する手法およびその手法を実現したリファクタリング支援ツールの構築を進める。本研究開発の成果となるリファクタリングツールを用いることで、プログラマは誤りの混入や面倒なテストから解放され、従来よりも手軽にリファクタリングを適用することができる。このことはリファクタリングが頻繁に行われることにつながり、その結果としてソフトウェアの理解性が向上し、ソフトウェア保守にかかる時間と費用を飛躍的に抑えることが可能である。また、必要時（ソフトウェアの実装や保守時）にリファクタリングを行えばよいという考えは、ソフトウェアの設計がその初期段階から完全でなくてもよいことを容認する。このため、ソフトウェア設計に対して不必要に時間や費用を費やすことが避けられる。さらには、あらかじめ再利用可能なソフトウェアを作成しておくのではなく、必要時にリファクタリングにより再利用可能なソフトウェア（部品）を作成することができことになる。これにより、ソフトウェアに対するさまざま、かつ、流動的な要求に柔軟に対応することが可能となる。

リファクタリングは、XP (Extreme Programming)[3]のようなコード中心のプログラム開発方法の基盤技術であり、その自動化はソフトウェア開発が短期化する、および、保守ソフトウェアがますます増加する情報化社会において、高品質なソフトウェアを提供するために必要不可欠な技術である。

### 3. 技術開発の内容

#### 3.1 技術開発項目の全体像

オブジェクト指向プログラムに対するリファクタリング操作を支援するツールを研究開発する。本研究開発における技術開発項目は大きく2つである。

- (1) プログラムが安全にリファクタリングを適用できるように、リファクタリング操作の自動化手法を確立し、この手法に基づくツールを実現する。
- (2) プログラムが容易にリファクタリングを適用できるように、変更目標（ゴール）に応じて適切なリファクタリング計画（プラン）を提示する仕組みを確立し、開発ツールに組み込む。

これら2つの技術開発項目の関係を図2に示す。

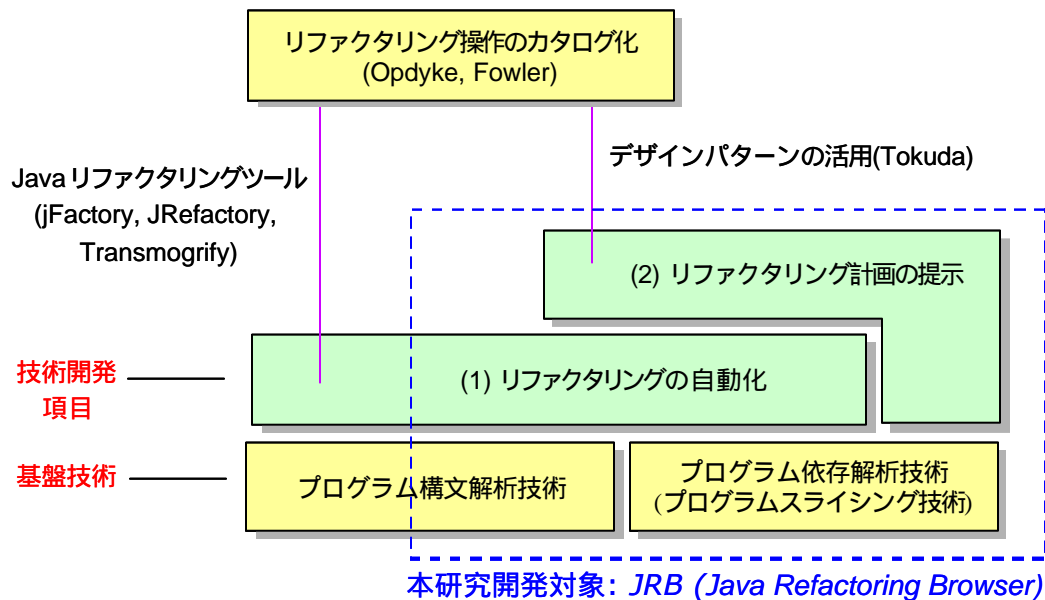


図2 本研究開発の技術開発項目と基盤技術

技術開発項目(1)は、主にオブジェクト指向プログラムの構文解析および依存解析技術を基盤とする。技術開発項目(2)は、プログラマが過去に行ったリファクタリングの履歴から、次のリファクタリング操作を提案するリファクタリング操作の蓄積・検索技術に関するものである。

#### 3.2 リファクタリングの自動化(技術開発項目1)

##### 3.2.1 技術課題

リファクタリングにおける主な変換操作は、Opdyke と Fowler の研究[1][2]に見ることができる。Opdyke は、26の基本変換と3つの複合変換(基本変換の組み合わせ)を提唱している。各変換には、変換前に成立する前提条件と操作(変換手順)が定義されており、変換前後で挙

動を保存することが保証されている。Fowler は、72 の変換に対して、その名前・動機・手順・例などのカタログ化を行っている。これらの変換は実践的に集められたものであり、変換前後における挙動の保存は特に保証していない（プログラマが各変換後にテストを行うことで挙動が保存されていることを確認する）。これらの変換操作を支援するリファクタリングツールとしては、Smalltalk プログラム対応 Refactoring Browser[4]や Java プログラム対応の jFactory[5], JRefactory[6], Transmogrify[7]などがある。これらのツールは、主にプログラムの構文解析技術に基づきリファクタリングを実現している。このため、プログラムの挙動を変えずに適用できる操作は制限され、ごく一部の操作に関してのみ自動化が行われている。

### 3.2.2 活用する理論・技術

本研究開発では、Fowler の研究[2]に見られる各種リファクタリング操作に対して、プログラム構文解析技術だけではなく、プログラム依存解析を用いることにより、それらの操作の自動化を試みる。本手法では、プログラムの挙動をソースコード内に存在するコード断片間の依存関係で捉え、依存関係を変換前後で保存することで、リファクタリング前後の挙動が等価であることを保証する。プログラム内部の依存関係を正確に把握し、その解析結果を利用することで、従来のツールに比べて数多くのリファクタリング操作の自動化が可能となる。

オブジェクト指向プログラムに対する依存解析を正確に行うためには、プログラム構文解析技術、制御/データフロー解析技術が必須である。さらに、本研究開発では、プログラム依存解析をリファクタリング操作に組み入れる際に、プログラム内部の特定の挙動に着目して関連のあるコード断片を集めるプログラムスライシング技術を活用する。本研究開発で用いるプログラム解析技術を以下にまとめる。

#### (1) 制御フローグラフ(CFG: Control Flow Graph)[8]

手続き型プログラムにおける基本ブロック間の制御の流れを表す。本研究開発における手法では、メソッド内の制御の流れを CFG により表現する。メソッド  $M$  の CFG とは、以下の条件を満たす有効グラフ  $G = (N, E, \text{start})$  である。

- (a)  $N$  は基本ブロックを表すノードの集合である。本手法では、基本ブロックにプログラムの代入式（代入文、入力文、出力文）、条件式（分岐文、ループ文）、メソッド呼出し文を割り当てる。 $N$  は、メソッドを実行するときに最初に制御が移される開始ノード  $\text{start}$  と、制御が移る先を持たない終了ノード  $\text{stop}$  を必ず 1 つずつ含む
- (b)  $E$  は矢印集合である。ノード  $p$  からノード  $q$  への矢印は、ノード  $p$  の文（命令）を実行した後、プログラム制御が直ちにノード  $q$  に移る可能性があることを指す。このとき、ノード  $p$  をノード  $q$  の直接先行(predecessor)、ノード  $q$  をノード  $p$  の直接後行(successor)と呼ぶ。
- (c)  $\text{start}$  は、メソッド  $M$  を実行するときに最初に制御が移される開始文を表すノードである。

CFG は、ソースコードに対して、プログラム制御の流れを静的に解析することで得られる。



図 3 に CFG の例を示す．左側は，オブジェクト指向プログラムのソースコードの一部（メソッド statement）を，右側はメソッド statement の CFG である．CFG において，T は条件の真方向，F は条件の偽方向への流れを表す．CFG を用いることで，プログラムは逐次実行(連接構造: sequence)，条件分岐(選択構造: alternation)，繰り返し(反復構造: iteration)の 3 つで表現される．

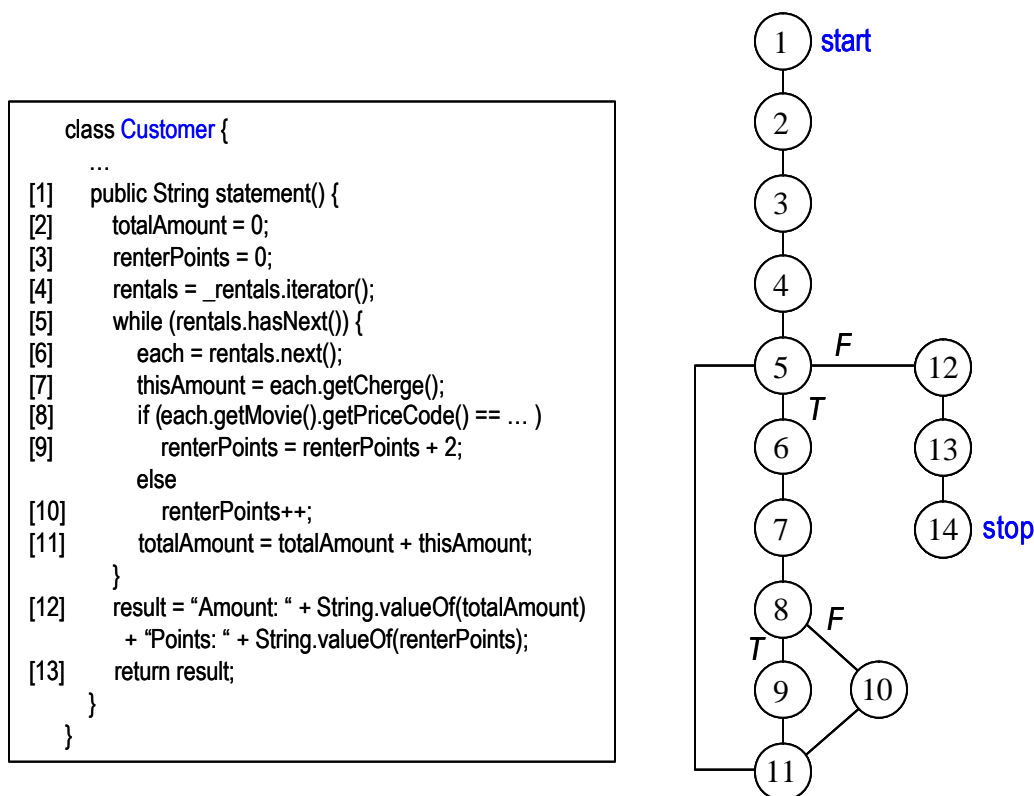


図 3 ソースコードとその制御フローグラフ(CFG)

## ( 2 ) プログラム依存グラフ(PDG: Program Dependence Graph)[9]

手続き型プログラムにおける文（命令）間の依存関係を表す．本手法では，各メソッド内に存在する依存関係を PDG により表現する．メソッド  $M$  の PDG とは，以下の条件を満たす有効グラフ  $G = (N, E, \text{entry})$  である．

- (a)  $N$  はプログラムにおける代入式あるいは条件式を割り当てたノードの集合，および，entry ノードである．PDG の  $N$  から entry ノードを取り除いた集合は，CFG における  $N$  から start ノードおよび stop ノードを取り除いた集合に等しい．
- (b)  $E$  は矢印集合である．ノード  $p$  からノード  $q$  への矢印は，ノード  $p$  からノード  $q$  に対して，制御依存関係(CD: control dependence)，あるいは，データ依存関係(DD: data dependence)があることを指す．
- (c) entry はどの節点にも依存しない入口節点である．(entry 節点を接続先とする依存関係矢印が存在しない．)

制御依存関係およびデータ依存関係の定義を以下に示す．

- ノード  $p$  における条件式の実行結果がノード  $q$  の実行を行うかどうか直接影响到を与える場合、ノード  $p$  からノード  $q$  に制御依存関係があるという。すなわち、 $p$  が条件分岐ノード (if 文に対応) であり、ノード  $q$  がその分岐内に直接含まれている場合、あるいは、 $p$  が繰り返しノード (while 文に対応) であり、ノード  $q$  がそのループ内に直接含まれている場合を指す。ノード  $p$  がノード  $q$  の分岐内あるいはループ内に直接含まれているとは、1) ノード  $q$  がノード  $p$  の分岐内あるいはループ内に含まれる、かつ、2) ノード  $q$  がノード  $p$  の分岐内あるいは繰り返し内に含まれる他の節点の分岐内あるいはループ内に含まれていないことを意味する。本手法では、条件式の実行結果の真偽に基づき制御依存関係矢印を区別し、各制御依存関係矢印に T 属性(true)あるいは F 属性(false)を付加する。
- ノード  $p$  における変数  $v$  の定義が変数  $v$  を参照するノード  $q$  に到達する場合、ノード  $p$  からノード  $q$  にデータ依存関係があるという。すなわち、1) ノード  $p$  において変数  $v$  の値を定義し、ノード  $q$  において変数  $v$  の値を参照している、かつ、2) 変数  $v$  の値を再定義しないノード  $p$  からノード  $q$  への CFG における到達可能経路が存在することを指す。ここで、変数  $x$  を定義するとは、変数  $x$  に値を設定することを指す。また、変数  $x$  を使用するとは、変数  $x$  の値を参照することを指す。また、到達可能経路とは、CFG において、ノード  $p$  からノード  $q$  までフロー矢印をたどる際に通過するノードの系列  $\langle p, \dots, q \rangle$  を指す。

PDG は、ソースコードに対して、制御フローおよびデータフローを静的に解析することで作成する。本手法では、オブジェクト指向プログラムが依存解析対象であるため、PDG を拡張したクラス依存グラフ(CIDG: Class Dependence Graph)[10]の一部の矢印 (メソッド呼出し関係を表現する矢印) を追加する。図 3 のソースコードに対する PDG (CIDG) の例を図 4 示す。一点鎖線矢印は制御依存関係、実線矢印はデータ依存関係を表す。

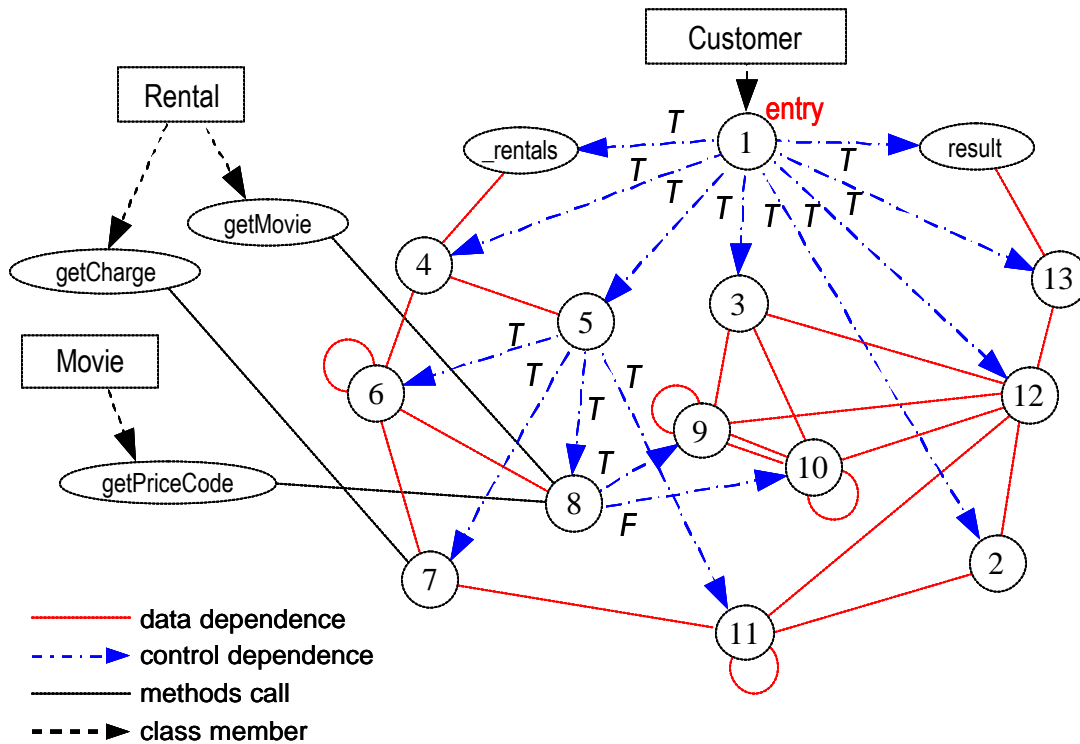


図4 プログラム依存グラフ(PDG)

### (3) プログラムスライス(Program Slice)[11]

プログラムスライシングとは、依存関係に基づき、ソースコード内の任意の文（ノード）において着目する変数に関連を持つ一部の文（ノード）集合だけを、もとのソースコードから抜き出すことである。抜き出したコードの集まりをプログラムスライス(program slice)、あるいは、単にスライスと呼ぶ。スライスには、すべての入力データに対して解析することで得られる静的スライス(static slice)[11]と、特定の入力データ（実行系列）に対して解析することで得られる動的スライス(dynamic slice)[12][13]が存在する。本手法では、スライシング技法をソースコードの変換に用いるため、静的スライスのみを扱う（よって、スライスといった場合、静的スライスを意味する）。さらに、本手法では、スライシングを着目変数の値に影響を与える（着目変数の値を計算する）ソースコードを新たなメソッドとして抽出する目的で導入するため、逆方向スライス(backward slice)を扱う。逆方向スライスとは、以下の条件を満たす有効グラフ  $G = (N, E)$  である。

- (a) 着目する文に対応するノード  $n$  において変数  $v$  が定義あるいは利用されているとする。  
 $\langle n, v \rangle$  をスライシング基準(slicing criteria)と呼ぶ。 $N$  は、PDG において、データ依存関係矢印と制御依存関係矢印を順方向にたどることでノード  $n$  に到達するノードを集めたものである。また、 $E$  は、ノード集合  $N$  を求める際に通過した矢印の集合を指す。  
 スライスは必ずもとの PDG の部分グラフとなる。

図3のソースコードにおいて、ノード11の変数のtotalAmountをスライシング基準とするス

ライスを図5示す．右側のPDGにおいて，網掛けのノードとそれらのノードを接続する矢印がスライスである．左側のソースコードは，スライスに基づき，もとのソースコードを再構成したものである．

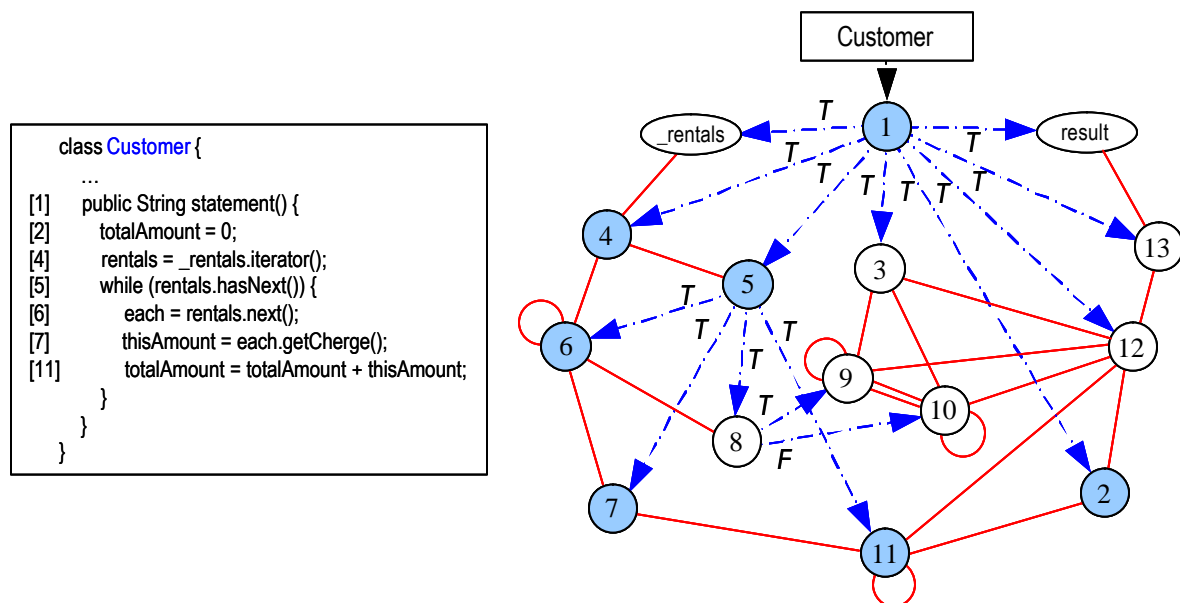


図5 プログラムスライスの例

### 3.2.3 現状の到達度（過去の実績度）

本研究代表者は，従来のプログラムスライシングに，連続する基本ブロックで構成された領域という概念を組み込むことで拡張した基本ブロックスライシングを用いて，既存メソッドから適切な規模の新規メソッドを半自動的に抽出するリファクタリング手法を提唱している[14]．また，本研究者はリファクタリング以外にも依存関係解析に基づきオブジェクト指向プログラムを自動変更する研究を行っている[15]．さらに，オブジェクト指向プログラムだけではなく，手続き型言語（C言語）を対象として既存プログラムから再利用部品を抽出する手法[16]やソースコードを再構成する手法[17]の研究も行ってきた，これらの研究と本研究開発では，プログラム解析技術に関して共通する部分が多い．

まず，現在の依存解析技術で比較的容易に実現できると考えられるリファクタリング操作の自動化を行い，自動化の効果を確認する．また，プログラムスライシングを用いたメソッド抽出リファクタリング(Method Extraction)，および，未だ自動化されていないリファクタリング操作に対する実現可能性を吟味し，順次自動化部分を増やしていく．

## 3.3 リファクタリング計画の提示（技術開発項目2）

### 3.3.1 技術課題

本研究開発では，プログラマの変更目標に応じた適切なリファクタリング計画（プラン）を提示する仕組みを確立する．一般的に，いつ，どのコードに，どのリファクタリング操作を適

用すればよいのかを，プログラマが判断することは困難である．このため，非熟練プログラマやリファクタリングに慣れていないプログラマが，実際の適用において常に大きな効果を得ることができるとは限らない．また，リファクタリングの結果や効果は，リファクタリングの適用順序に大きく依存する．このため，不適切な適用順序では十分な変換が行われず，リファクタリングにより理解性や再利用性の向上が見込めない場合が起こりうる．このような観点から，どのようにリファクタリングを適用していくのかの計画を決定する指針は必須であり，変更目標に対応させて体系化していくことが重要である．

### 3.3.2 活用する理論・技術

本研究開発では，既存ソフトウェアに対して，過去にプログラマが行った一連のリファクタリング操作を蓄積し，将来のリファクタリングを誘導する手法の確立を試みる．たとえば，同一あるいは類似の操作が適用された場合，過去あるいは別のプログラマのリファクタリング履歴を検索することで，適切なリファクタリング操作を選択し，それらを合成する（組み合わせる）ことで一連の操作列を自動生成することが考えられる．これにより，非熟練プログラマでもリファクタリングを容易に適用できるようになり，保守性の高いソフトウェアを短期間に開発できる可能性は高くなる．この技術課題に関しては，知識の表現形式や分類および検索方式など解決しなければ課題が多いため，まずリファクタリング操作の蓄積に着目し，その実現を目指す．ここでは，過去のソフトウェア開発時において行われたリファクタリング操作の系列を対象ファイルごとに蓄積する．プログラマがリファクタリングを適用する意思を示した際に，プログラマの行ったリファクタリング操作の系列と，蓄積されている過去の系列を比較することで，次に行うべき操作を提示することを考えている．また，過去のリファクタリング操作をその変更目標と対応させる方法として，デザインパターン[18]を変更目標とみなすことを考えている．

### 3.3.3 現状の到達度（過去の実績度）

本研究代表者は，過去のメソッド変更操作に基づき将来の変更を予測して，既存クラスを Template Method パターンに変換するリファクタリング操作の自動化手法を既に提唱し，その有効性を評価実験により示した[17]．また，Fowler の研究[2]では，リファクタリングを行う時期や場所を決定する基準（手がかり）を，コード内の不吉な匂い(bad smells)として列挙している．別のアプローチとして，デザインパターンをリファクタリング結果の目標とみなし，既存のソースコードをリファクタリングする操作手順（変換名，目的，条件，操作）が Tokuda によりまとめられている[19]．このように目標が明確なリファクタリングに関しては，蓄積時にその操作との対応が付けやすく，本技術課題の解決方法として活用できる可能性は高い．

まず，プログラマが行ったリファクタリング操作列を実際に蓄積し，手動での解析を試み，どのように体系化できるのかを検討する．この検討結果をふまえ，どのような形でプログラマにリファクタリング計画を提示するべきであるかを検討する．

## 4. 開発・実装システム内容

### 4.1 研究開発システムの全体像

研究開発するソフトウェアの構成を図6に示す。以下、このシステムをJRB (Java Refactoring Browser と呼ぶ。研究開発システムが対象とするオブジェクト指向プログラミング言語としては、多くのプログラマに一般的であるという理由からJava言語を採用する。四角形はツールの各構成要素を、角丸四角形はデータを表す。

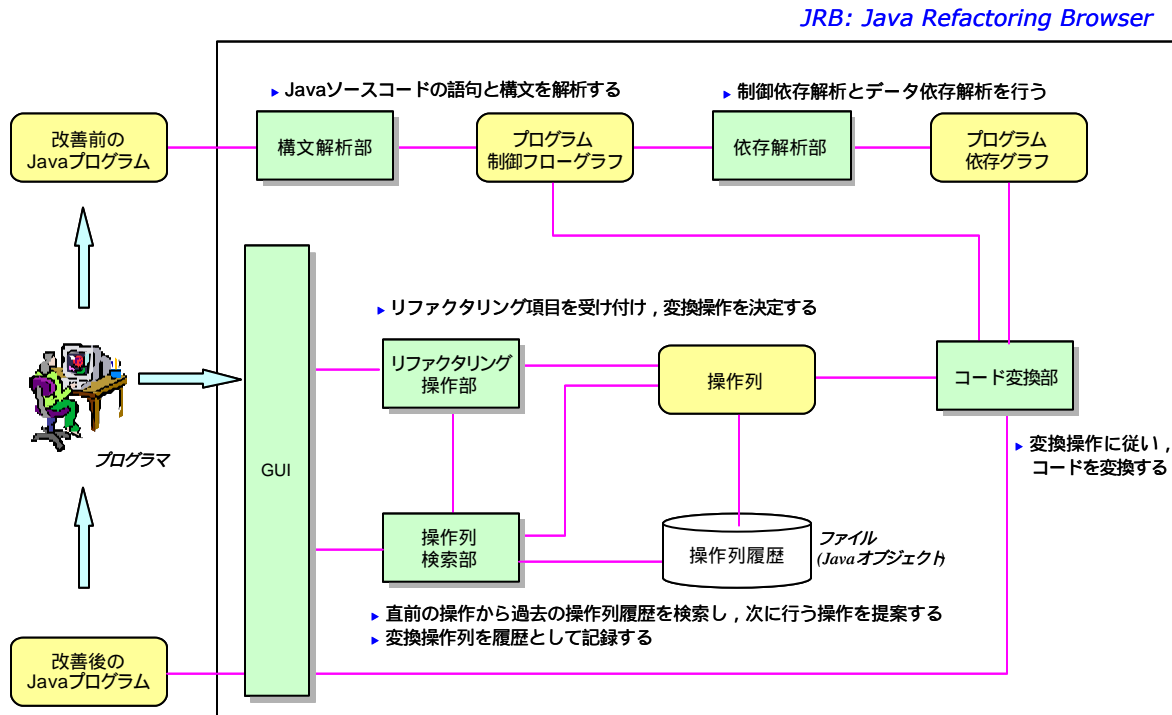


図6 本研究開発システム（ツール）の構成

JRBは、(1)リファクタリング部と(2)ユーザインタフェース(GUI)部で構成されている。まず、本システムで対象とするオブジェクト指向言語Javaおよびそのソースコードモデルについて4.2で述べる。その後、機能(1)(2)について詳細を4.3で述べる。

### 4.2 Java 言語

Java言語は、クラスに基づく強く型付けされた(strongly-typed)オブジェクト指向プログラミング言語である[20]。Javaの実行プログラムは、Javaソースコードをコンパイルすることによって得られる。JRBがリファクタリング対象とするのは、Java言語で記述されたソースコードである。JRBでは、Javaソースコードに関して、図7に示すソースコードモデルを構築する。

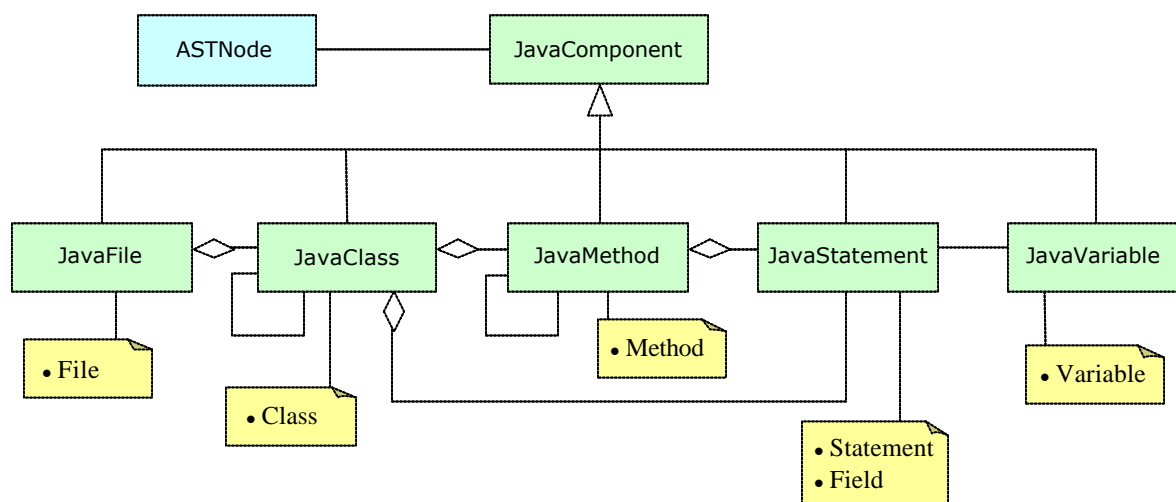


図7 JRBで構築するJavaソースコードモデルのクラス構成

JavaComponentは、Javaソースコードの構成要素を指す。また、ASTNodeは、Javaソースコードの構文木におけるノードを指す。図7においてクラスに付加されているノートは各構成要素との対応を表す。たとえば、クラス(Class)に関する情報は、JavaClassオブジェクトが保持する。

### 4.3 機能及びその構成概要

#### 4.3.1 リファクタリング部

プログラマが開発中のJavaソースコードに対して、GUIを介して指示したリファクタリング操作を適用し、変換後のJavaソースコードを出力する機能を持つ。リファクタリング部は、(1)構文解析部、(2)依存解析部、(3)コード変換部、(4)リファクタリング操作部、(5)操作列検索部で構成されている。次に、それぞれの機能について概要と入出力情報を説明する。

##### (1) 構文解析部

入力として指定されたJavaソースコードの字句と構文を解析し、各クラスに対して構文木およびソースコードモデルを構築する。さらに構文木とソースコードモデルから、Javaプログラムのメソッドごとに制御フローグラフ制御を作成する。構文木の作成には、JavaCC (Java Compiler Compiler)[21]を利用した。

入力情報：Javaソースコード(文法エラーを含まない、コンパイル可能なもの)

出力情報：構文木(AST)、制御フローグラフ(CFG)

##### (2) 依存解析部

Javaソースコードから作成したCFGに対して、データ依存解析と制御依存解析を適用し、プログラム依存グラフ(PDG: Program Dependence Graph)を作成する。本研究開発システムでは、オブジェクト指向プログラムが依存解析対象であるため、PDGに、メソッド呼出し関係を表現する

矢印を付加したクラス依存グラフ(CIDG: Class Dependence Graph)を構築する．具体的には，CFGの各ノードに対して，制御の従属関係を調査して，制御依存関係(CD)を抽出し，制御依存グラフを作成する．次に，CFGの各節点に対して，変数の定義と参照の関係（データの到達可能性）を調査して，データ依存関係(DD)を抽出し，データ依存グラフを作成する．さらに，クラス内部およびクラス間に成立するメソッド呼出しの関係を表現した依存関係矢印(Call edge)を求める．データ依存グラフと制御依存グラフを重ね合わせ，メソッド呼び出し矢印を追加することで，PDG(CIDG)を作成する．

入力情報： 制御フローグラフ(CFG)

出力情報： プログラム依存グラフ(PDG)，クラス依存グラフ(CIDG)

さらに，依存解析部では，与えられた PDG において，スライシング基準として指定された節点と変数に関する依存関係矢印をたどることでスライス进行を計算する機能を提供する．スライス作成機能の入出力情報を以下に示す．

入力情報： プログラム依存グラフ(PDG)，スライシング基準（節点と変数の組）

出力情報： スライス（入力 PDG の部分グラフ）

### （ 3 ） コード変換部

入力 Java ソースコードの構文木，制御フローグラフ，プログラム依存グラフを用いて，与えられたリファクタリング操作に従い，Java ソースコードに変換操作を適用する．さらに，変換後のコードに対して，整形を行う．

入力情報： 構文木，ソースコードモデル，制御フローグラフ(CFG)

プログラム依存グラフ(PDG)，リファクタリング操作名

出力情報： リファクタリング後（変換および整形後）の Java ソースコード

### （ 4 ） リファクタリング操作部

GUI のメニューを介して与えられたリファクタリング要求に対して，適切な変換操作コマンドを決定する．また，同時に，リファクタリング操作の取り消し機能を提供する．

入力情報： リファクタリング要求（リファクタリングの種類や名前）

出力情報： 操作列（操作実行コマンド）

### （ 5 ） 操作列検索部

直前（1つ前および2つ前）のリファクタリング操作に応じて過去の操作列を検索することで，次に行うリファクタリング操作を提案する．また，実行操作列をリファクタリング履歴としてファイルに蓄積する

入力情報： 直前のリファクタリング操作，履歴ファイル

出力情報： 履歴から検索した変換操作の集合



#### 4.3.2 ユーザインタフェース(GUI)部

ソースコードの編集機能を持ち、マウスおよびキーボードイベントを取得する機能を提供する。与えられたイベントに従い、リファクタリング要求や変更目標を取得し、それぞれリファクタリング操作部あるいは操作列検索部に引き渡す。

##### (1) 編集機能

ソースコード(ファイル)の新規作成・読み込み・保存・名前変更・印刷、ソースコードの書き換え・コピー・切り取り・貼り付け・追加・削除、ソースコード上での字句検索・置換、リファクタリング前後のソースコードを切替え表示する機能を提供する。

入力情報：マウスおよびキーボード入力

出力情報：表示ソースコード、ファイル

##### (2) イベント取得機能

マウスイベントおよびキーボードイベントを取得する機能を提供する。与えられたイベントに従い、リファクタリング要求や変更目標を取得する。

入力情報：マウスイベント、キーボードイベント

出力情報：リファクタリング要求あるいはリファクタリング目標

#### 4.4 システムの特徴

本システムは、次に示す特徴を持つ。

- リファクタリングにおける変換を実行する際に、構文木だけではなく、制御フローグラフ、プログラム依存グラフ、プログラムスライシングを活用することで、従来は不可能であったリファクタリング操作が実現されている。たとえば、「スライスによるメソッド抽出」、「polymorphism による条件分岐(switch 文)の置き換え」というリファクタリング操作が実行できる。
- リファクタリング操作を履歴として蓄積することで、次に行うべき操作を提案する。また、過去の操作履歴を、操作列単位、コマンド別、ファイル別、利用者別、時刻順で表示することで、プログラムのリファクタリング操作を支援する。
- 編集機能におけるファイルごと取り消しのほかに、複数のファイルに関して、リファクタリング操作を一括して取り消すことが可能である。この取り消し操作は、ファイルごとの取り消しおよびやり直しと矛盾しないように設計されている。

#### 4.5 稼動環境

##### (1) ハードウェア環境

CPU：Intel Pentium 互換，SunSparc

メモリ：128MB 以上

ハードディスク：300MB 以上

グラフィックカード：VGA 互換ボード，256 色以上

## (2) ソフトウェア環境

OS：Windows2000/98/Me, Solaris8

Sun Microsystems Java2 SDK Standard Edition, 1.3

Borland JBuilder4 Foundation (本システムのエディタとして利用する場合)

## (3) 条件：Java プログラムが起動できること。

### 4.6 インタフェース仕様

本研究開発システムの外部とのインタフェースは，以下の2つである。

#### (1) ユーザとシステム間のインタフェース

#### (2) 操作列検索部と履歴ライブラリとのインタフェース

#### 4.6.1 ユーザとシステム間のインタフェース

JRBの基本画面を図8に示す。基本画面は，ファイル選択画面，ソースコード編集画面，クラス名/メソッド名表示画面の3つに分けられている。

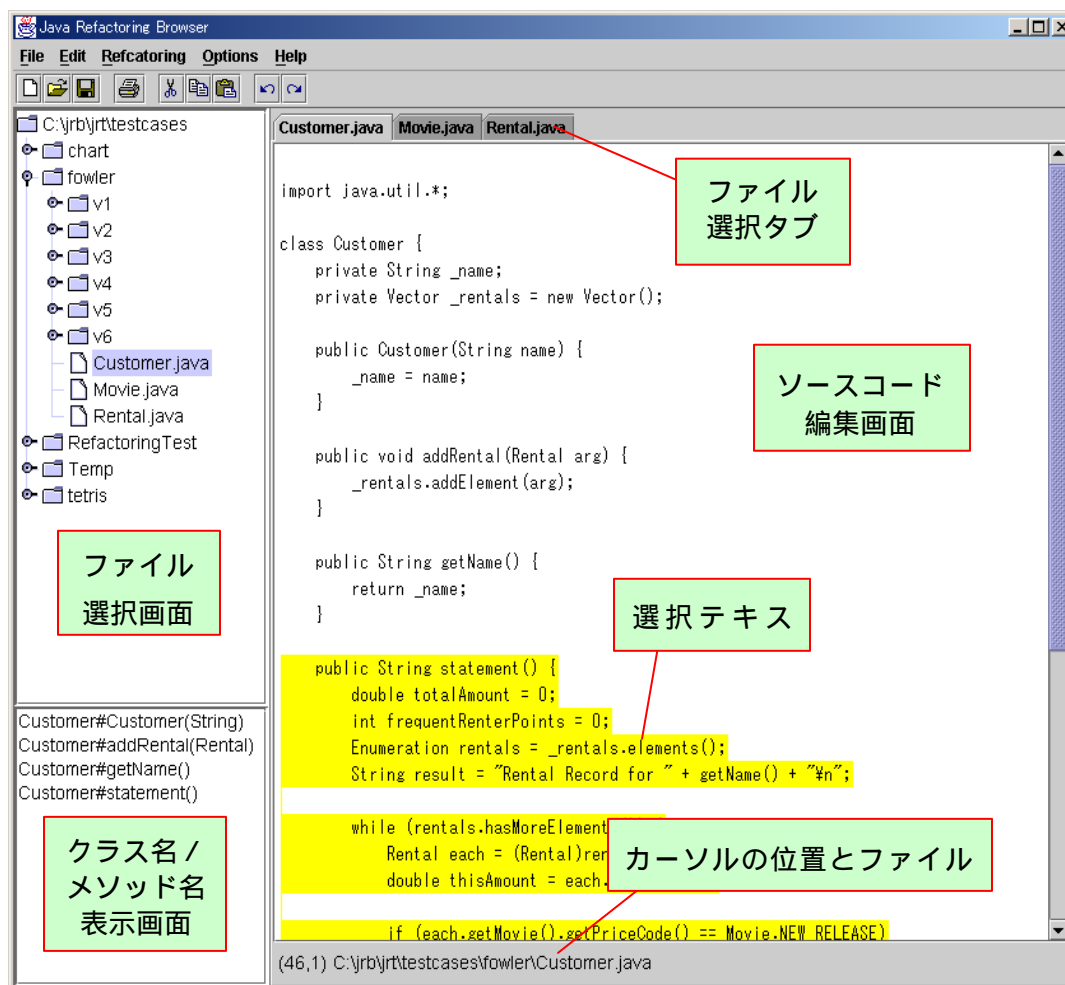


図8 JRBの基本画面

### ( 1 ) ファイル選択画面

探索ディレクトリ ( 環境変数 Root.Dir で指定されたディレクトリ ) 以下の Java ソースコード ( 拡張子が java である ) ファイルがディレクトリの階層に従い表示される . この画面上でファイル名をクリックすると , 指定されたファイルがファイル編集画面に読み込まれる . すでに読み済みのファイルの場合は , 指定ファイルの編集画面が最前面に呼び出される .

### ( 2 ) ソースコード編集画面

利用者に指定されたファイルの内容 ( ソースコード ) が表示される . 複数のソースコードが読み込まれている場合 , それぞれのソースコードのタブをクリックすることで , 指定ソースコードを最前面に呼び出すことができる . 利用者は , この画面内でマウスをドラッグすることにより対象テキストを選択し ( 実際のソフトウェアでは編集画面上で黄色表示になる ) , ソースコードの編集 , 検索 , 置換 , リファクタリング操作を行う .

### ( 3 ) クラス名 / メソッド名表示画面

最前面のソースコード内に存在するクラスとメソッドを表示する . 表示形式は , 「クラス名 #メソッド名(引数の型の並び)」である .

図 9 ~ 13 にリファクタリングメニューを示す . 各操作の内容は 5 章で述べる . 各操作の実行手順は , 「ソフトウェア操作説明書」の 4 章を参照のこと .

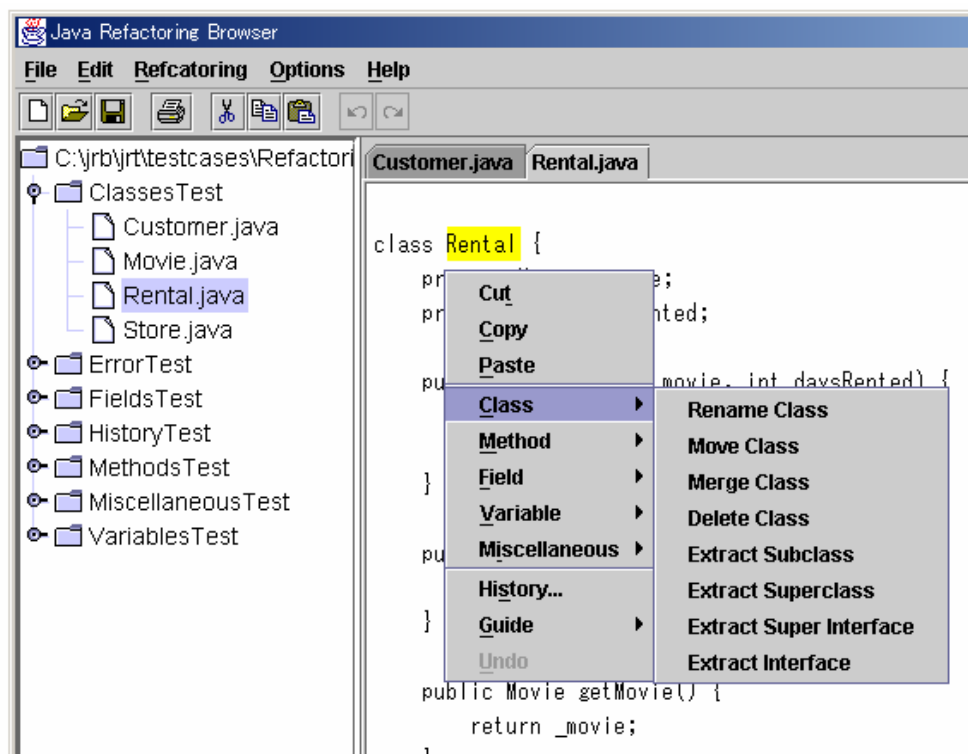


図 9 クラスリファクタリングメニュー

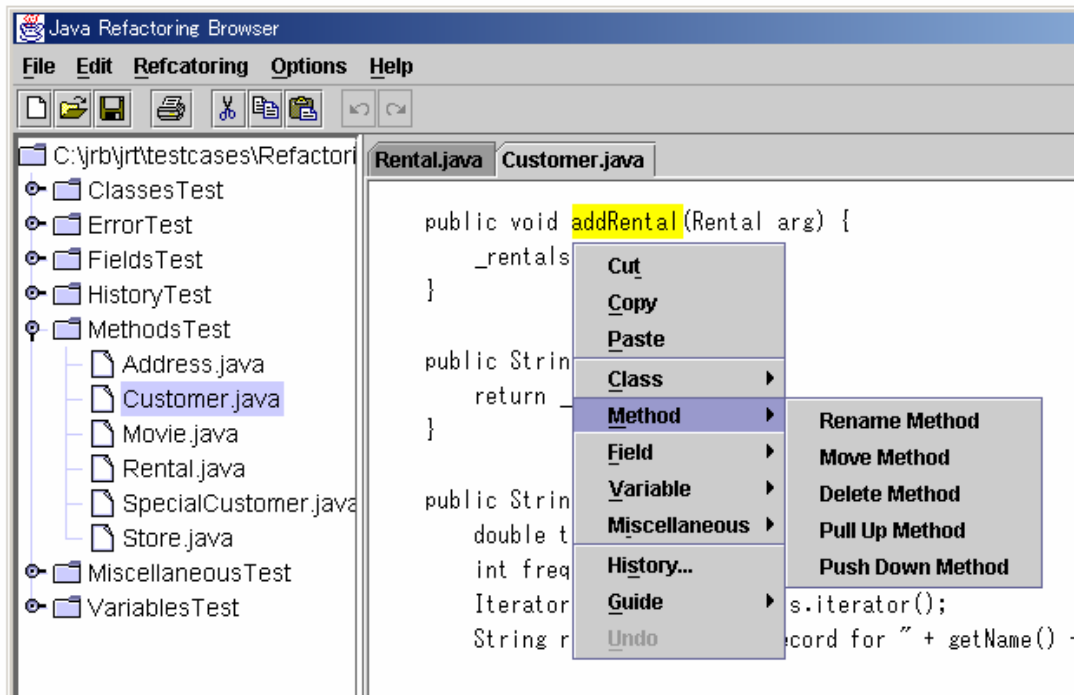


図 1 0 メソッドリファクタリングメニュー

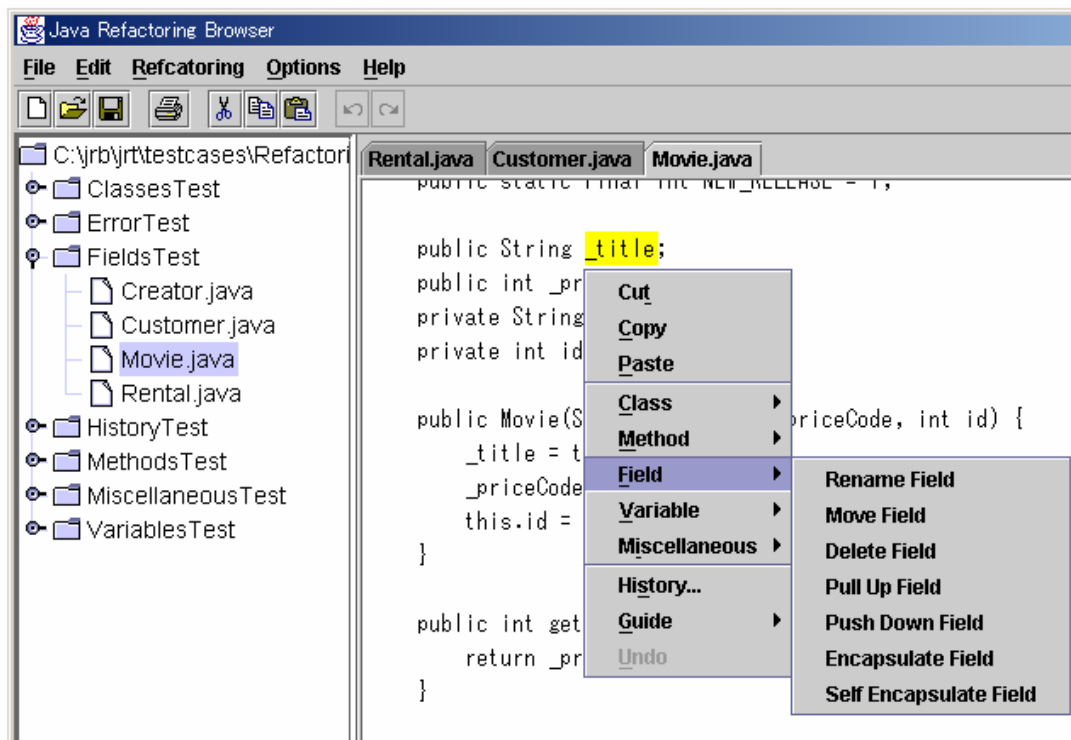


図 1 1 フィールドリファクタリングメニュー

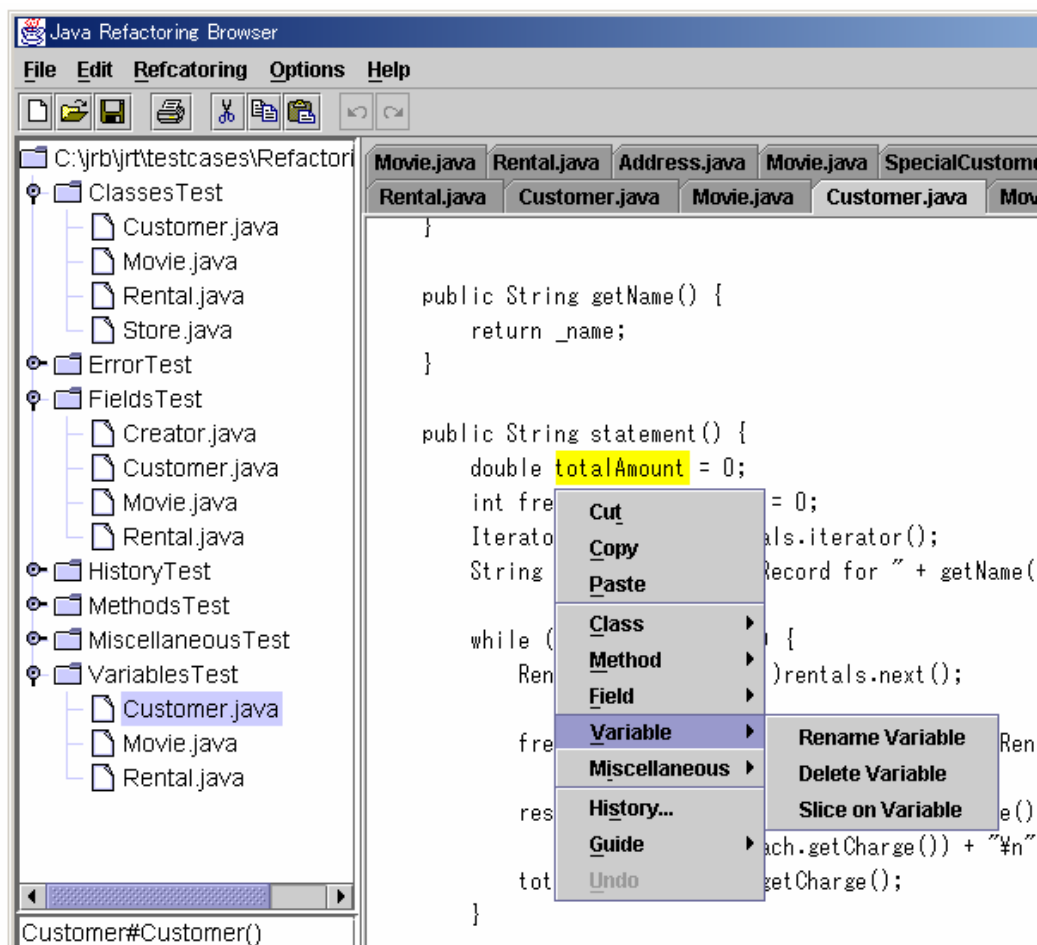


図 1 2 ローカル変数リファクタリングメニュー

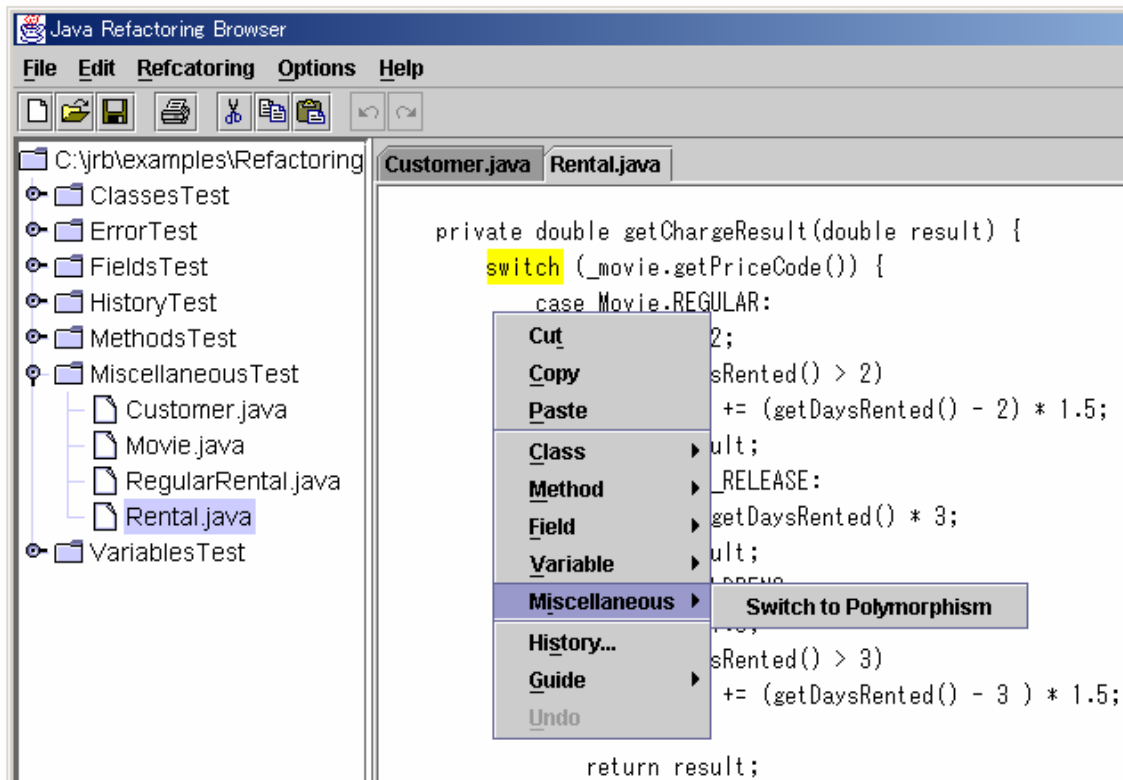


図 1 3 その他リファクタリングメニュー

#### 4.6.2 操作列検索部と履歴ライブラリとのインタフェース

JRB では、リファクタリング履歴を Java オブジェクトとしてファイルに蓄積する。クラス RefactoringRecord が蓄積されるオブジェクトのクラスである。このクラスのフィールドを以下に示す。

```
public class RefactoringRecord implements Serializable {
    private String command;        // 操作の名前
    private String fileName;       // 変換ファイル
    private Timestamp timestamp;   // 実行時刻
    private String userName;       // 利用者
}
```

## 5. 実装と適用実験

JRBの動作を確認するために、Javaソースコードに対して、リファクタリングの適用実験を行った。本章では、まず実装した各リファクタリング操作の内容と、変換前後でソースコードの挙動を保存するための前提条件について説明する。次に、実験内容とその結果について報告する。

### 5.1 実装したリファクタリング操作

表1に本研究開発において実装したリファクタリングの種類と操作の一覧を示す。これらのリファクタリング操作の内容と、実装した前提条件を5.1.1～5.1.5に示す。

説明において、探索ディレクトリとは、リファクタリング操作時に検査されるファイル群が格納されているトップディレクトリを指す。これは、あらかじめ利用者が環境変数あるいはOptionsメニューにより指定する。また、同一のクラスが存在するとは、同じ名前を持つクラスが存在することを指す。同一のメソッドが存在するとは、同じシグニチャを持つメソッドの宣言が存在すること指す。同一のフィールドが存在するとは、同じ名前のフィールドが宣言されていること指す。同一のローカル変数が存在するとは、同じ名前のローカル変数が宣言されていることを指す。

表1 リファクタリングの種類と操作

対象	リファクタリング操作
クラス	名前の変更
	移動
	合併
	削除
	サブクラスの抽出
	スーパークラスの抽出
	スーパーインタフェースの抽出
	インタフェースの抽出
メソッド	名前の変更
	移動
	削除
	引き上げ
	引き下げ

フィールド	名前の変更
	移動
	削除
	引き上げ
	引き下げ
	カプセル化
	自己カプセル化
ローカル変数	名前の変更
	削除
	スライスの抽出
その他	polymorphism による条件分岐(switch 文)の置き換え

#### 5.1.1 クラスリファクタリング(Class Refactoring)

##### ( 1 ) 名前の変更(Rename Class)

指定されたクラスの名前を変更する .

前提条件 :

- (a) 変更後の名前が Java 言語の名前として妥当
- (b) 変更後の名前を持つクラスが , もとのクラスのパッケージ内に存在しない
- (c) 変更後の名前を持つクラスが , もとのクラスで利用されていない

##### ( 2 ) 移動(Move Class)

指定されたクラスを異なるファイルに移動する .

前提条件 :

- (a) 移動先の Java ファイルが存在する .
- (b) 移動先のファイルに同一のクラスが存在しない

##### ( 3 ) 合併(Merge Class)

指定されたクラスのメソッドとフィールド全体を異なるクラスに挿入する .

前提条件 :

- (a) 合併先クラスが存在する
- (b) 合併先クラスに同一のメソッドが存在しない
- (c) 合併先クラスに同一のフィールドが存在しない

##### ( 4 ) 削除>Delete Class)

指定されたクラスを削除する .

前提条件 :



- (a) 削除クラスが探索ディレクトリ下の他のクラスで利用されていない。
- ( 5 ) サブクラスの抽出(Extract Subclass)  
指定されたクラスのサブクラスの宣言を新規に作成する。  
前提条件：
  - (a) 新規の名前が Java 言語の名前として妥当
  - (b) 新規クラスと同一のクラスが，もとのパッケージ内に存在しない
- ( 6 ) スーパークラスの抽出(Extract Superclass)  
指定されたクラスのスーパークラスの宣言を新規に作成する。  
前提条件：
  - (a) 新規の名前が Java 言語の名前として妥当
  - (b) 新規クラスと同一のクラスが，もとのパッケージ内に存在しない
- ( 7 ) スーパーインタフェースの抽出(Extract Super Interface)  
指定されたクラスのスーパーインタフェースの宣言を新規に作成する。  
前提条件：
  - (a) 新規の名前が Java 言語の名前として妥当
  - (b) 新規クラスと同一のクラスが，もとのパッケージ内に存在しない
- ( 8 ) インタフェースの抽出(Extract Interface)  
指定されたクラスのインタフェースを抽出する。  
前提条件：
  - (a) 新規の名前が Java 言語の名前として妥当
  - (b) 新規クラスと同一のクラスが，もとのパッケージ内に存在しない

#### 5.1.2 メソッドリファクタリング(Method Refactoring)

- ( 1 ) 名前の変更(Rename Method)  
指定されたメソッドの名前を変更する。  
前提条件：
  - (a) 変更後の名前が Java 言語の名前として妥当
  - (b) 変更後の名前を持つメソッドがもとのクラス内に存在しない
- ( 2 ) 移動(Move Method)  
指定されたメソッドを異なるクラスに移動する。  
前提条件：
  - (a) 移動先クラスが存在する
  - (b) 移動メソッドと同じメソッドがもとのクラスのスーパークラスに存在しない
  - (c) 移動メソッドと同じメソッドがもとのクラスのサブクラスに存在しない
  - (d) 移動メソッドがもとのクラスのスーパークラスのメソッドを呼び出していない
  - (e) 移動先クラスに移動メソッドと同一のメソッドが存在しない
  - (f) 移動先クラスを参照するオブジェクトが一意に決定できる(委譲メソッドあるい

は転送メソッドの作成に必要)

( 3 ) 削除(Delete Method)

指定されたメソッドを削除する .

前提条件:

- (a) 属性が private のとき , クラス内でメソッドが呼び出されてない
- (b) 属性が private 以外 のとき , 削除メソッドが探索ディレクトリ下の他のクラスから呼び出されていない

( 4 ) 引き上げ(Pull Up Method)

指定されたメソッドをスーパークラス ( 直属のクラス ) に移動する .

前提条件 :

- (a) 引き上げ先クラスが探索ディレクトリ下に存在する
- (b) クラス内フィールドを利用 ( 定義あるいは使用 ) していない .
- (c) 引き上げ先クラスに同一のメソッドが存在しない
- (d) もとのクラス内において呼び出していたメソッドと同一のメソッドが引き上げ先に存在しない

( 5 ) 引き下げ(Push Down Method)

指定されたメソッドをサブクラス ( 直属の子クラス ) に移動する .

前提条件 :

- (a) もとのクラスにおいて , 他のメソッドに呼び出されていない
- (b) もとのクラスにおいて , 他の private メソッドを呼び出していない
- (c) もとのクラスにおいて , private フィールドを利用していない
- (d) 引き下げ先クラスに同一のメソッドが存在しない
- (e) 引き下げメソッドが呼び出しているメソッドと同一のメソッドが引き下げ先クラスに存在しない
- (f) 引き下げメソッドが利用しているフィールドと同一のフィールドが引き下げ先クラスに存在しない

### 5.1.3 フィールドリファクタリング(Field Refactoring)

( 1 ) 名前の変更(Rename Field)

指定されたフィールドの名前を変更する .

前提条件 :

- (a) 変更後の名前が Java 言語の名前として妥当
- (b) 変更後の名前を持つフィールドがもとのクラス内に存在しない

( 2 ) 移動(Move Field)

指定されたフィールドを異なるクラスに移動する .

前提条件 :

- (a) 移動先クラスが存在する

- (b) 移動フィールドの宣言部において他のフィールドを利用（使用）していない
- (c) 移動フィールドの属性が `private` である
- (d) 移動フィールドがアクセッサ(getter と setter)を持つ
- (e) 移動フィールドのアクセッサと同一のメソッドが移動先クラスに存在しない
- (f) 移動先クラスに移動フィールドと同一のフィールドが存在しない
- (g) もとのクラスにおいて、移動フィールドが直接（アクセッサを介さずに）利用されてない

( 3 ) 削除(Delete Field)

指定されたフィールドを削除する．

前提条件：

- (a) 属性が `private` のとき、クラス内でメソッドが利用されていない
- (b) 属性が `private` 以外るとき、削除フィールド探索ディレクトリ下の他のクラスで利用されていない

( 4 ) 引き上げ(Pull Up Field)

指定されたフィールドをスーパークラス（直属の親クラス）に移動する．

前提条件：

- (a) 引き上げ先クラスが探索ディレクトリ下に存在する
- (b) 移動フィールドの宣言部において他のフィールドを利用（使用）していない
- (c) 引き上げ先クラスに同一のフィールドが存在しない

( 5 ) 引き下げ(Push Down Field)

指定されたメソッドをサブクラス（直属の子クラス）に移動する．

前提条件：

- (a) もとのクラス内で他のフィールドに利用されてない
- (b) もとのクラスにおいて、他の `private` フィールドを利用していない
- (c) 引き下げ先クラスに同一のフィールドが存在しない
- (d) 引き下げフィールドが利用しているフィールドと同一のフィールドが引き下げ先クラスに存在しない

( 6 ) カプセル化(Encapsulate Field)

指定された公開(`public`)フィールドを非公開(`private`)フィールドに変更する．

前提条件：

- (a) 指定されたフィールドの属性が `private` でない
- (b) もとのクラスにおいて、他の `private` フィールドを利用していない

( 7 ) 自己カプセル化(Self Encapsulate Field)

指定されたフィールドを、アクセッサを介してのみ利用するように変更する．

前提条件：

- (a) 指定されたフィールドの属性が `private` である
- (b) 指定されたフィールドが単純な文(そのフィールドの定義と使用のどちらか一方

しか現れない文)でのみ利用されている

#### 5.1.4 ローカル変数リファクタリング(Variable Refactoring)

##### (1) 名前の変更(Rename Variable)

指定されたローカル変数(メソッドの引数を含む)の名前を変更する。

前提条件:

- (a) 変更後の名前が Java 言語の名前として妥当
- (b) 変更後の名前を持つローカル変数がもとのメソッド内に存在しない
- (c) 変更後の名前を持つフィールドがもとのクラス内に存在しない

##### (2) 削除>Delete Variable)

指定されたローカル変数を削除する。

前提条件:

- (a) 指定されたローカル変数が、その変数を所有するメソッド内で宣言されているだけ(定義や参照がない)、あるいは、その値が一度だけ定義され参照がない

##### (3) スライスの抽出(Slice on Variable)

指定された変数に基づくスライスを作成し、メソッドとして整形する。ただし、このリファクタリングは試験的なものであるため、抽出後のソースコードは Java のコメントとなる。これにより、変換後のソースコードの挙動は保存される。

前提条件: なし

#### 5.1.5 その他リファクタリング(Miscellaneous)

##### (1) polymorphism による条件分岐(switch 文)の置き換え(Switch to Polymorphism)

switch 文の各条件分岐のアクション部をサブクラスのオーバーライドメソッドとして移動する。

前提条件:

- (a) switch 文を含むメソッド内に switch 文のみが存在する。(他の文を含む場合は、あらかじめ switch 文だけをメソッドとして抽出しておく)
- (b) switch 文を含むメソッドと同一のメソッドが移動先サブクラスに存在しない

#### 5.1.6 リファクタリングの取り消し(Undo)

JRB では、リファクタリング操作の履歴を保存することで、複数のファイルに対して一括して取り消しを行うことができる。たとえば、図 14 に示す操作列履歴があるとする、Undo メニューにより、(i)の操作列を持つファイルと(ii)の操作列を持つファイルのメソッド移動操作(Move Method)が同時に取り消される。ツール内部では、同時に行われたリファクタリング操作に、同じ識別番号を割り当てることで判断している。

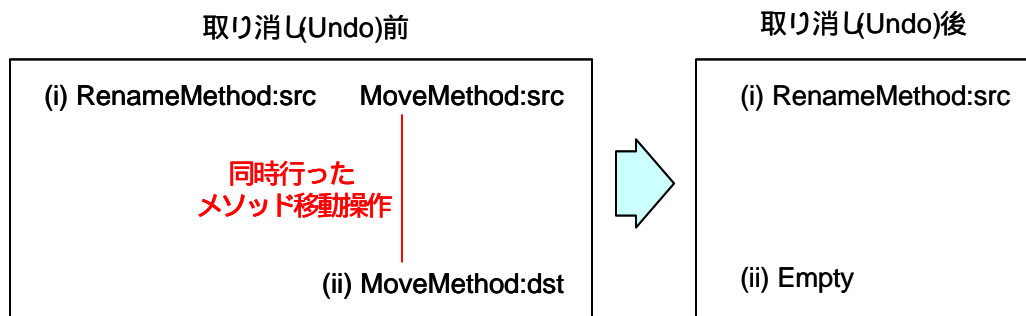


図 1 4 リファクタリング操作の取り消し

このように、同時に行われた操作が履歴の最後尾にそろっている場合は、取り消し操作によって、矛盾は生じない。しかしながら、図 1 5 に示すような操作列履歴に対して、「メソッド移動(Move Method)」操作を取り消すと、(ii)における「フィールドの移動(Move Field)」操作に関する整合性が失われる。すなわち、フィールド移動を同時に行った別のファイルとの整合性が取れなくなる。

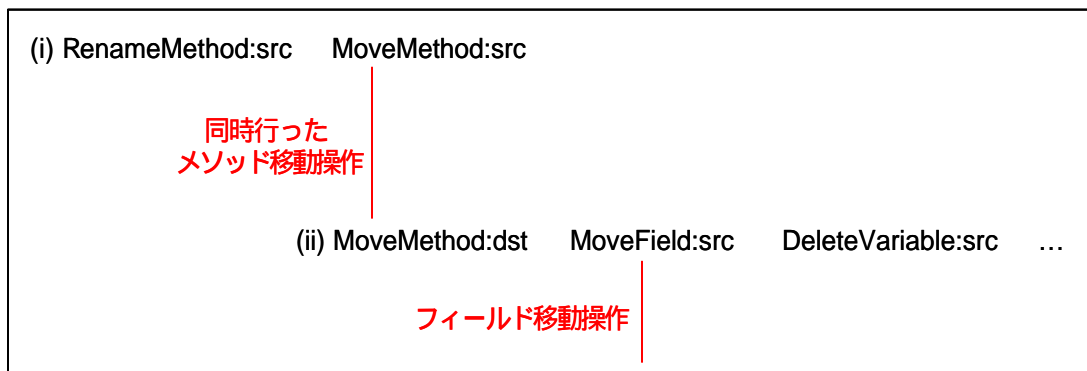


図 1 5 リファクタリング操作の履歴

そこで、JRB では、履歴の最後尾に一括取り消し可能な操作がそろっているかどうかを監視し、そろった場合にのみ取り消しメニューが選択可能となるように実装してある（この条件が満たされない場合、メニュー項目が選択できない）。ただし、リファクタリング操作以外の編集操作（たとえば、文字の挿入など）は一括取り消しの際に同時に取り消される。

#### 5.1.7 リファクタリングに関する次操作の提案

JRB では、リファクタリング操作を取り消しのためだけでなく、利用者のリファクタリング操作を支援することに活用する。いま、利用者が、編集中のソースコードにおいて、特定のリファクタリング操作を行ったとすると、JRB はその操作が過去に行われていないかどうか検索する。同じ操作が過去に行われている場合、過去における次の操作を履歴から取得し、メニューを介して利用者に提示する。たとえば、図 1 5 に示す履歴を持つ JRB において、利用者が「メ

ソッド名の変更(Rename Method)」操作を行うと、次の操作として、「メソッドの移動(Move Method)」操作が提案される。

JRB では、1 つ前の操作だけでなく、2 つ前の操作（つまり、2 つの操作の列）でも検索を行い、結果を提示する。また、検索結果が複数になる場合は、その頻度に応じてそれぞれ上位 5 つをメニュー項目の上位から配置し、利用者に提示する。

## 5.2 評価実験

本実験では、JRB を用いて、Java ソースコードに対してリファクタリングを適用し、実装した操作が実行可能であること、および、変換においてソースコードの挙動が保存されることを確認した。実験環境と内容を以下に示す。

### (1) 実験環境

- CPU: Intel Pentium III 1 GHz
- メモリ: 256MB
- OS: Windows2000 professional
- Java2 SDK Standard Edition, 1.3.1

### (2) 実験内容

5.1 に示したリファクタリング操作に応じて、表 2 に示す Java ソースコードを用意した。実験に用いたソースコードは、JRB の配布パッケージに含まれる。

表 2 実験に用いたソースコード

リファクタリングの種類	ソースコードの格納場所
クラスリファクタリング	examples/RefactoringTest/ClassesTest/
メソッドリファクタリング	examples/RefactoringTest/MethodsTest/
フィールドリファクタリング	examples/RefactoringTest/FieldsTest/
ローカル変数リファクタリング	examples/RefactoringTest/VariablesTest/
その他	examples/RefactoringTest/MiscellanerousTest/

これらのソースコードに対して、それぞれリファクタリングを実行し、5.1 に示した合計 24 つのリファクタリング操作が実行可能であること、および、変換後のソースコードがその挙動を保証することを手動で確認した。また、5.1 に示す前提条件を満たさない場合に、警告ダイアログが表示されることを確認した。図 16, 17 に JRB を用いたリファクタリング操作の様子を示す。

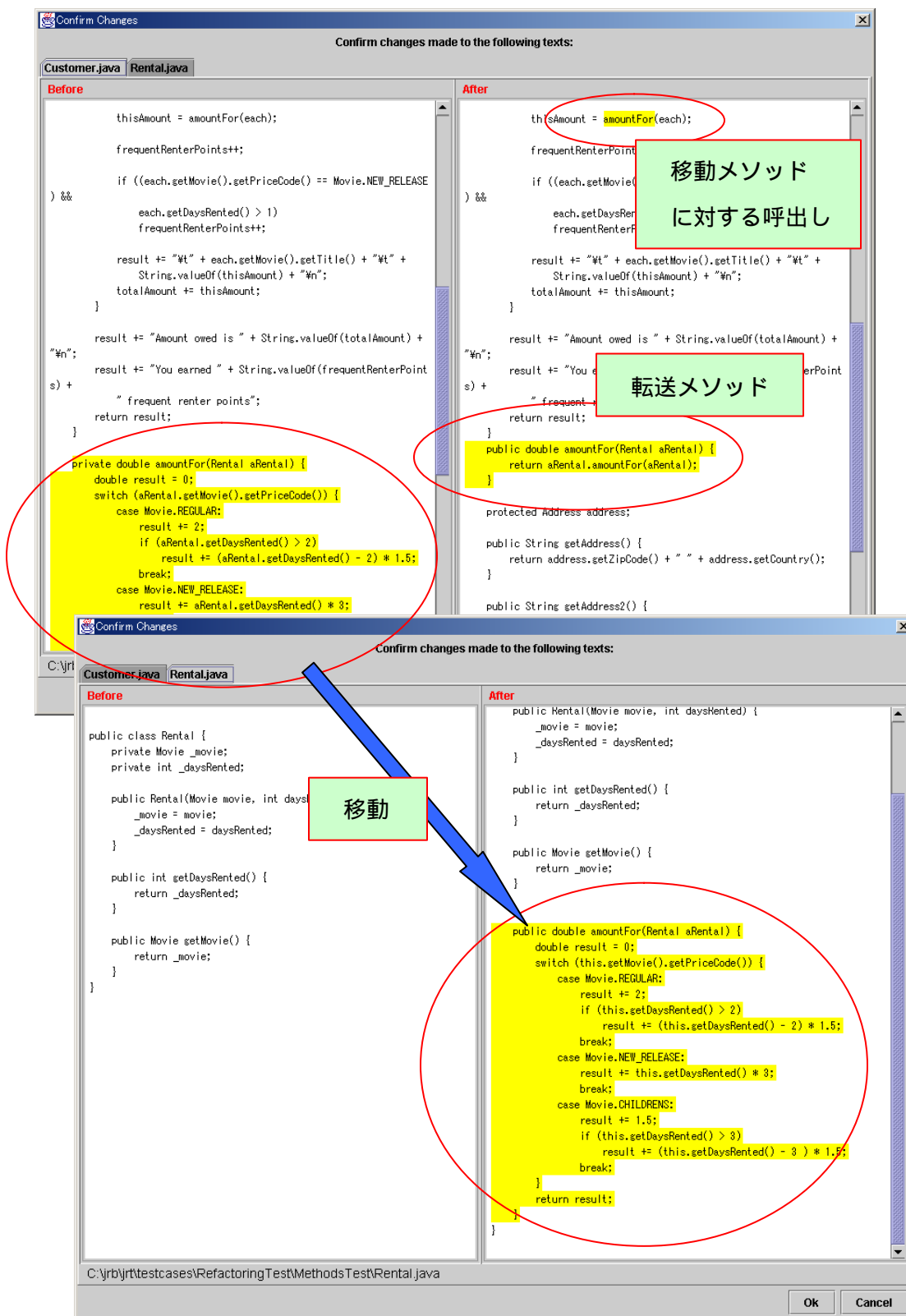


図 16 メソッドの移動リファクタリングの様子

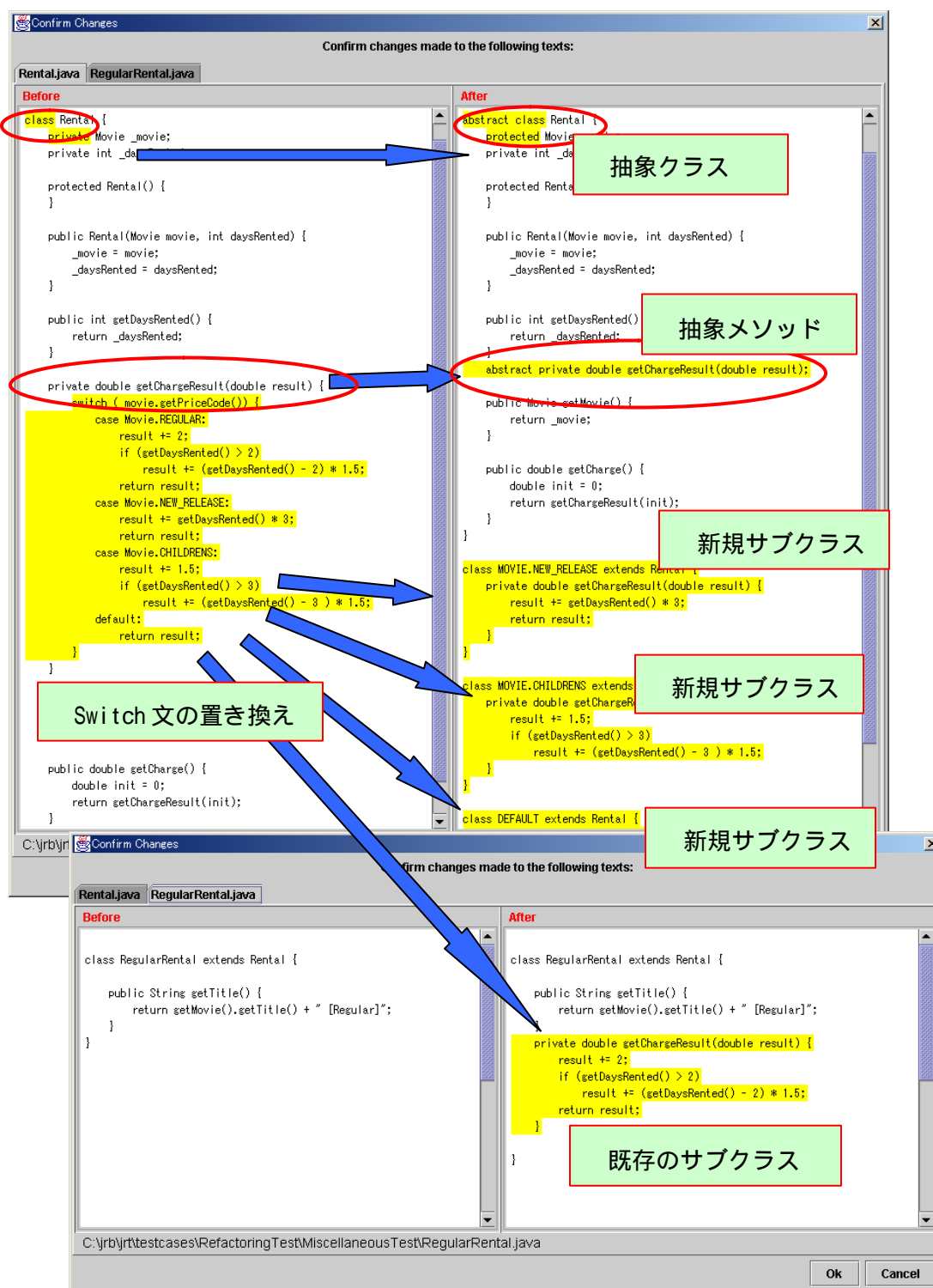


図 1 7 polymorphism による条件分岐の置き換えリファクタリングの様子



さらに、リファクタリング操作履歴と照らし合わせることで、リファクタリング操作の一括取り消しが実際にできることを確認した。また、リファクタリング履歴に応じて、次の操作が提案されることを確認した。図 1 8 に実験時のリファクタリング履歴を示す。

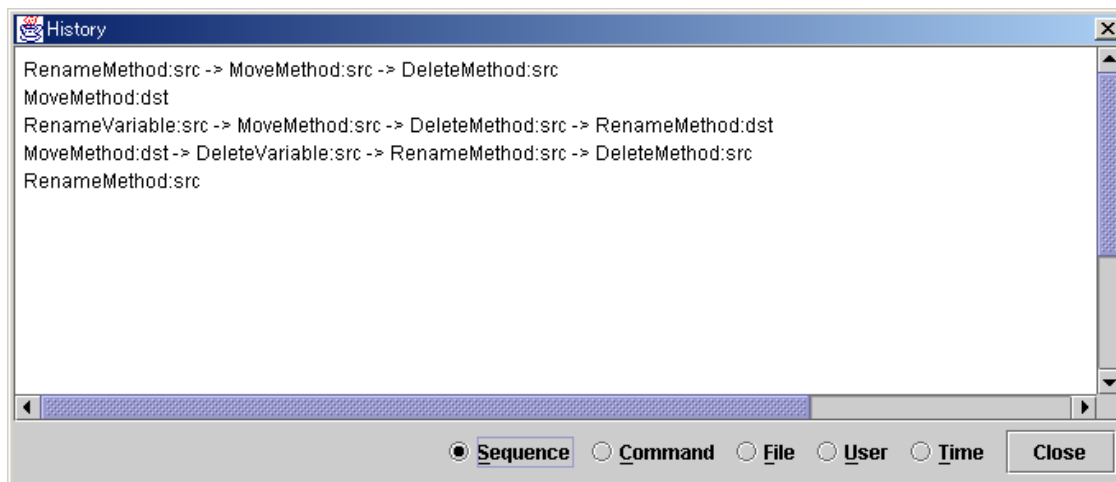


図 1 8 リファクタリング履歴の表示

いま、利用者が「メソッドの移動(Move Method)」操作を実行した場合、図 1 8 の履歴により、次の操作として、「メソッドの削除>Delete Method)」操作が提案される（1 行目と 3 行目が一致する）。このときのリファクタリングメニューの様子を図 1 9 に示す。

さらに、利用者が続けて「メソッドの削除>Delete Method)」操作を実行した場合、「メソッドの移動(Move Method)？ メソッドの削除>Delete Method)」と 3 行目の操作列が一致し、「メソッド名の変更(Rename Method)」が提案される。このときのリファクタリングメニューの様子を図 2 0 に示す。本実験を通して、リファクタリング履歴により、次操作が提案されることが確認できた。

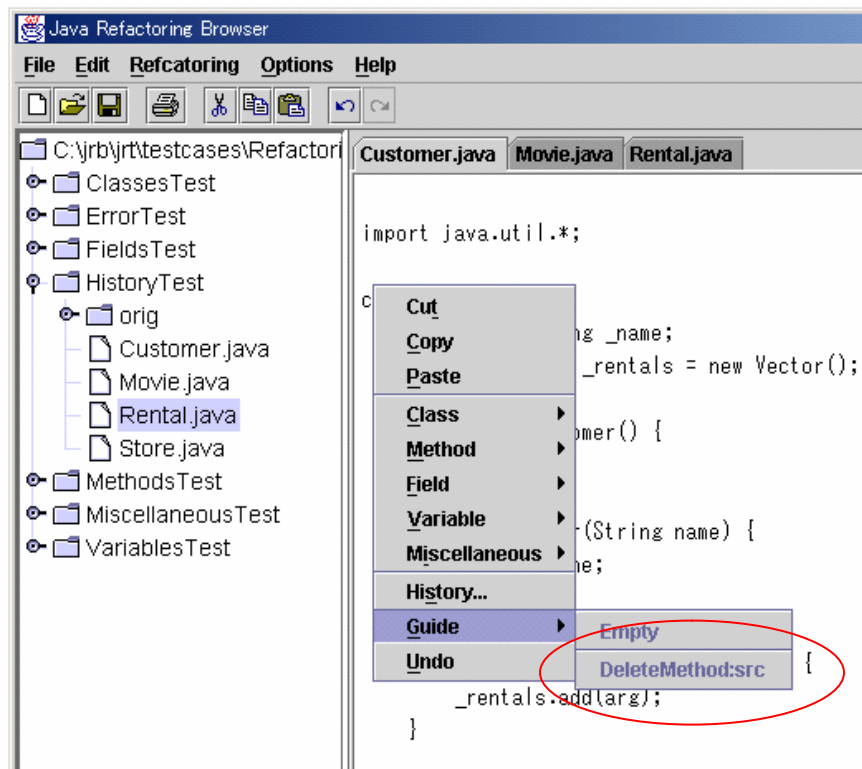


図 1 9 リファクタリング操作の提案 (1 つ前の操作が一致)

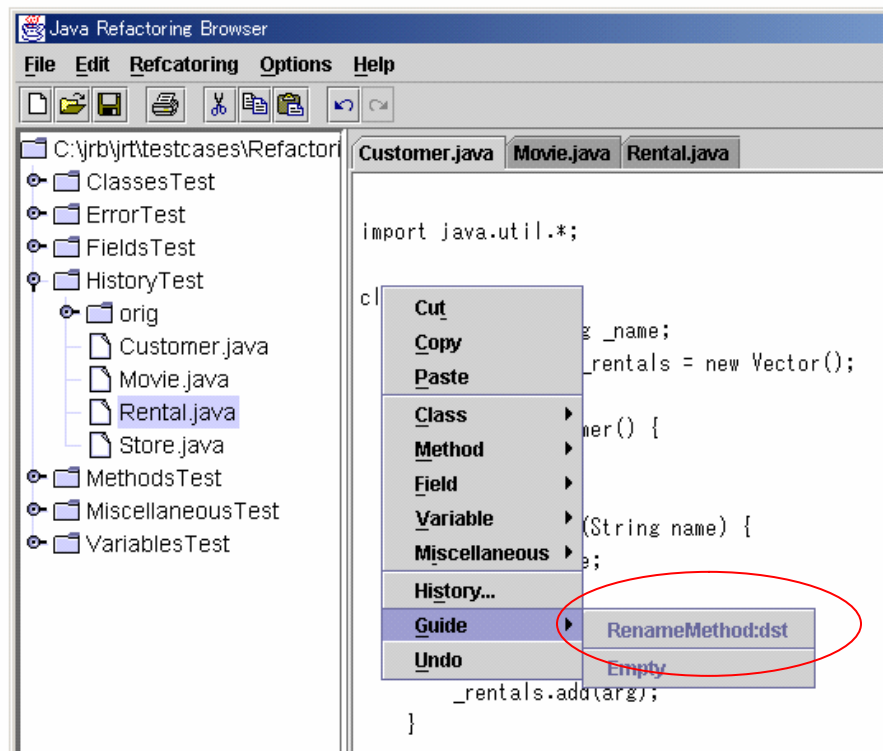


図 2 0 リファクタリング操作の提案 (1 つ前と 2 つ前の操作が一致)

## 6 . 評価・考察

### 6.1 リファクタリングの自動化に関して

リファクタリング支援ツールを開発し、実際に適用実験を行うことで、リファクタリングの自動化に対する可能性と有用性を確認できた。特に、本ツールでは、プログラムの制御フロー解析と依存解析を用いることで、従来は自動化が困難であった「スライスによるメソッド抽出」、および、「polymorphism による条件分岐の置き換え」という高度なリファクタリング操作の自動化に成功した点が大きい。プログラム解析技術の導入による成果は、オブジェクト指向ソフトウェア開発支援に対する大きな進展につながる可能性を持つ。

また、ファクタリングでは、ソースコードの挙動をその変換前後で保存することが必須である。このため、変換操作自体は単純な作業であっても（に見えても）、その変換を適用可能であるかどうかの検査、および、変換によって影響を受ける箇所の特定には非常に手間がかかることが、実際に自動化を進める過程で確認できた。たとえば、メソッドの引き上げ、引き下げなど操作自体はそれほど複雑ではないが、移動対象となるメソッドと他の構成要素との関係を正確に捉えていなければ、挙動を保存した変換は難しい。このような操作を手動で行っている限り、たとえリファクタリングの重要性が認められても、その作業を推進することは困難である。本研究開発を通して、リファクタリングを普及させるためには、その操作の自動化（半自動化）は欠かせないことが確認でき、リファクタリング操作を自動化することに大きな意義があることが示された。

自動化に関する異なる知見として、本研究開発の成果は完全自動化への限界も一部示している。当初、本ツールではできる限り人間の介入を行わない方向で自動化を目指した。しかしながら、現在のプログラム解析技術では、設計や実装の意図までは完全に抽出することはできない。たとえば、「フィールドの移動」に伴う参照オブジェクトやアクセッサは解析だけで十分特定可能と見ていたが、実際にはその特定が困難であることが確認できた。本ツールでは、このような場面において、利用者にダイアログを表示し、その情報を取得している。逆に、プログラムの解析だけをたよりに自動化を進めると、前提条件が厳しくなり、多くの操作が適用不可能であると判断されてしまう。よって、正確さは増すものの有用性が極端に低下する。プログラミング自体が人間の行う作業である以上、それを改善する作業においても人間の介入は避けられない。このようなツールを開発する際には、単に自動化を推し進めるのではなく、人間との協調を積極的に取り入れ、その上で自動化できる部分を判断することが重要である。

### 6.2 リファクタリング計画の提示に関して

本ツールでは履歴の検索による次操作の提示にとどまったが、履歴の活用方法の一つとして、その有用性の一部は立証できたと考えられる。また、リファクタリング操作の履歴が自動的に収集されるという点で、本ツールの利用価値は高い。しかしながら、実際にツールを利用してみると、提示された操作が次に行うべき操作として妥当であるのかどうか疑問を持つこともあった。たとえば、「メソッド移動」操作の次に、「メソッドの削除」操作や「メソッド名の変

更」操作が提示されるが、これらの操作が「メソッド移動」操作の後に必須なものであるとはいえない。リファクタリングの最終目標がソースコードや設計の改善である以上、単純に履歴を検索するだけで次操作を決定してしまう手法では限界があることがわかった。履歴の活用に関しては、リファクタリング支援という点で改良の余地が多く残されており、履歴の収集方法や提示方法にさらなる工夫が必要である。熟練プログラマーがどのような場面で、どのようなリファクタリング操作を行っているかを調査することが、リファクタリング計画の提示の研究開発を進めていく上で重要である。

### 6.3 開発ツールの実装に関して

当初予定していたよりも、多くのリファクタリングメニューが提供できた。この理由として、リファクタリングにおける変換処理、および、前提条件の検査処理の多くが共通していることが挙げられる。本ツールの開発を通して、共通部分の部品化（クラス化）がうまくできると、それほど大きな負担なしでメニュー項目を追加していけることが確認できた。また、本ツールの実装においては、各リファクタリング操作に対して、前提条件を検査するクラスと実際にコードを変換するクラスを常に1つずつ用意し、Template Pattern[18]を活用して処理の流れの共通化を図った。これにより、メニューの追加が容易に行えるようになっている。コンポーネント化をさらに進めることで、リファクタリングメニューを利用者がカスタマイズできるリファクタリングツール（リファクタリングビルダー）を構築できる可能性は高いと考えられる。

本ツールの実用性に関しては改良の余地が残されている。1つに対象プログラミング言語の制約がある。本ツールは、Java 言語に完全には対応していない。たとえば、インナークラスに対する操作や例外処理を含むメソッドに対しては挙動を保存した変換を保証していない。さらに、フレームワーク全体まで含めた依存解析には膨大なコストがかかるため、現実装ではJDKの内部まで解析していない。さらに、本ツールの設計および実装においては、機能の実現と変更の容易さを重視し、高速化や小容量化を意識していない。よって、選択したリファクタリング操作によっては、変換に時間がかかり、利用者を待たせることがある。今後、より複雑なリファクタリング操作を実装していくにあたり、変換にかかる時間も考慮する必要があると考えられる。

## 7. 今後の課題

今後の課題として、以下の3点が挙げられる。

### (1) リファクタリング操作の安全性の保証に関して

依存解析の精度不足および対象プログラミング言語に対する制約により、現時点では自動化された変換が必ずソースコードの挙動を保存することが証明されていない。また、各リファクタリング操作に関してその前提条件が正しいことも証明していない。これらの点に関して、本ツールを検証する必要がある。

### (2) 評価実験に関して

本研究開発における適用実験では、数少ないソースコードおよび利用者での結果しか得られていない。多くのJavaソースコードに対して、リファクタリング操作を行い、提案手法およびツールの有用性を確認する。さらに、多くのプログラマに対して、リファクタリング履歴を収集し、リファクタリング計画の提示手法のさらなる確立を目指す。

### (3) 実用性に関して

本ツールは、オブジェクト指向ソフトウェア開発において単独でできるように、簡単な編集機能を持つエディタを有する。しかしながら、編集機能の充実という点では、既存のソフトウェア開発統合環境に及ばない。今後は、既存の開発環境にリファクタリング部を組み入れ、開発環境としての利便性を向上させる予定である。さらに、実用に耐えられることを目指し、高速化および小容量化という点から本ツールを再構成することを考えている。

### (4) ソースコードの公開

本研究開発の成果(論文, プログラム説明書)とともにソースコードをWebにおいて積極的に公開することで、リファクタリング技術の普及を促進する。

## 8 . 参考文献

- [1] Opdyke, W. F., “Refactoring Object-Oriented Frameworks”, Ph.D. dissertation, Univ. Illinois, 1992
- [2] Fowler, M., “Refactoring: Improving the Design of Existing Code”, Addison-Wesley, 1999
- [3] Beck, K., “Extreme Programming Explained”, Addison-Wesley, 1999
- [4] Roberts, D. Brant, J. and Johnson, R., “A Refactoring Tool for Smalltalk”, Theory and Practice of Object Systems (TAPOS), Vol.3, No.4, pp4.39-42, 1997
- [5] jFactory, Instantiations Inc, <http://www.instantiations.com/jfactor/>
- [6] Seguin, C., JRefactory, <http://jrefactory.sourceforge.net/>
- [7] Transmogrify, <http://transmogrify.sourceforge.net/>
- [8] Aho, A.V., Sethi, R. and Ullman, J.D., “Compilers: Principles, Techniques, and Tools”, Addison-Wesley, 1986
- [9] Ferrante, J., Ottenstein, K.J. and Warren, J.D., “The Program Dependence Graph and Its Use in Optimazation” ACM Trans. Programming Language Systems, Vol.9, No.3, pp.319-349, 1987
- [10] Rothermel, G and Harrold, M.J., “Selecting Regression Tests for Object-Oriented Software”, Proc. Intl. Conf. Software Maintenance, pp.14-25, 1994
- [11] Weiser, M., “Progarm Slicing”, IEEE Trans. Software Engineering, Vol.10, No.4, pp.352-357, 1984
- [12] Agrawal, H. and Horgan, J.R., ”Dynamic Program Slicing”, Proc. Conf. Programming Language Design and Implementation. , pp.246-256, 1990
- [13] Korel, B. and Laski, J., “Dynamic Program Slicing”, Information Processing Letters, Vol.29, No.3, pp.155-163, 1988
- [14] Maruyama, K, “Automated Method-Extraction Refactoring by Using Block-Based Slicing”, Symp. Software Reusability, pp.31-40, 2001
- [15] Maruyama, K. and Shima, K, “An Automatic Class Generation Mechanism by Method Inetgration”, IEEE Trans. Software Engineering, Vol.26, No.5, pp.425-440, 2000
- [16] 丸山勝久, 高橋直久, ”区間設定可能なプログラムスライシングを用いたソフトウェア部品の作成“ . 情報処理学会論文誌 , Vol.37, No.4, pp.520-535, 1996
- [17] 丸山勝久 ,島健一, “重み付き依存グラフを用いたメソッドの再構成” ,情報処理学会論文誌 , Vol.41 , No.6, pp.1777-1790, 2000
- [18] Gamma, E., helm, R., Johnson, R. and Vlissides, J., ‘Design Patterns: Elements of Reusable Object-Oriented Software”, Addison-Wesley, 1995
- [19] Tokuda, A. L., Evolving Object-Oriented Designs with Refactorings, Ph.D. dissertation, University of Texas, 1999
- [20] Gosling, J., Joy, B., and Steel, G., “The Java Language Specification”, Addison-Wesley, 1996
- [21] Metamata, Inc. and Sun Microsystems, Java Compiler Compiler, [http://www.webgain.com/products/java\\_cc/](http://www.webgain.com/products/java_cc/)

## おわりに

本研究開発では、オブジェクト指向プログラムに対するリファクタリング操作の自動化手法の提案とその支援ツールの開発を行った。本ツールの特徴は、リファクタリング操作の自動化に、プログラムの制御フロー解析および依存解析技術を導入したことである。さらに、実装ツールを用いた簡単な適用実験を行うことで、リファクタリング操作の自動化の意義を示した。リファクタリング技術が普及し、JRB のような自動化リファクタリングツールが多くのソフトウェア開発統合環境に組み込まれることが強く望まれる。

本研究開発は、情報処理振興事業協会より委託を受けた財団法人ソフトウェア工学研究財団が実施した「平成 13 年度高度情報化支援ソフトウェアシーズ育成事業」の支援のもとで遂行した。ここに、深く感謝の意を表します。