社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report
SS2009-45 (2009-12)

リファクタリングのための変更波及解析を利用したテスト支援ツールの提案

吉田　昌友† 　吉田　則裕† 　松下　誠† 　井上　克郎†

† 大阪大学 大学院情報科学研究科
〒 565-0871 大阪府吹田市山田丘 1 番 5 号
E-mail: †{mstm-ysd,n-yosida,matusita,inoue}@ist.osaka-u.ac.jp

あらまし　リファクタリングとは，外部的振る舞いを保ちつつ，ソースコードの保守性を改善する作業を指す．リファクタリングを行う開発者は，テストケースを使用してテストを行うことによって，外部的振る舞いが保たれていることを確認する．しかし，大規模ソフトウェアの開発では，膨大な数のテストケースを定義することが多く，その中から外部的振る舞いの検証に必要なもののみを選択し，使用することは困難である．本研究では，Java 言語で記述されたプログラムに対して変更波及解析を行い，実行結果が変化する可能性があるテストケースを編集内容を基に推測することで，リファクタリング支援を行うツールを提案する．

キーワード　リファクタリング，ソフトウェアテスト，変更波及解析，Eclipse プラグイン

A Change Impact Analysis Based Testing Tool for Refactoring

Masatomo YOSHIDA†, Norihiro YOSHIDA†, Makoto MATSUSHITA†, and Katsuro INOUE†

† Graduate School of Information Science and Technology, Osaka University
1-5 Yamadaoka, Suita, Osaka, 565-0871 Japan
E-mail: †{mstm-ysd,n-yosida,matusita,inoue}@ist.osaka-u.ac.jp

Abstract　Refactoring is the process that can improve the software maintainability without changing its external behavior. When developers perform refactoring, they preserve software's external behavior by running test cases as regression test. In many cases of large scale software developments, developers create a large number of test cases. However, it is difficult to select and run only necessary test cases for preserving software's external behavior. In this paper, we propose a testing tool for Java program refactoring. This testing tool applies change impact analysis to two programs before and after editing for the purpose of refactoring, and then infers test cases that have possibility to change their results.

Key words　Refactoring, Software Test, Change Impact Analysis, Eclipse Plug-in

1. はじめに

リファクタリング [1] とは，外部的振る舞いを保ちつつ，ソースコードの保守性を改善する作業を指す [2]．Fowler はリファクタリングを行うべきソースコードの状態に応じて，さまざまなリファクタリングパターンを提案している [2]．そして，いくつかのリファクタリングパターンは，ソースコードを編集する作業に限り統合開発環境 Eclipse [3] のプラグイン JDT(Java development tools) [4] で支援されている（図 1(a)）．JDT とは Java 言語でソフトウェアを開発する際に利用できる Eclipse のプラグインである．JDT を使用する開発者が，リファクタリングを行う対象や行うリファクタリングの内容を設定すれば，そのリファクタリングを行うことを目的としてコード変換を適用したソースコードのプレビューが Eclipse 上に提示される．プレビューが提示された時点ではソースコードは変換されていないため，そのコード変換を実際に行うか選択することができる．

リファクタリングを行う開発者は，ソフトウェアの外部的振る舞いが保たれていることを確認するために，テストケースを使用してテストを行い，ソースコードを編集する前と編集した後とでテストの実行結果が同じであることを確認する [2]．

しかし，大規模ソフトウェアの開発においては，膨大な数のテストケースを定義していることが多く，リファクタリングを行う度にすべてのテストケースを使用することは効率が悪い．そこで，外部的振る舞いの検証に必要なテストケースのみを選

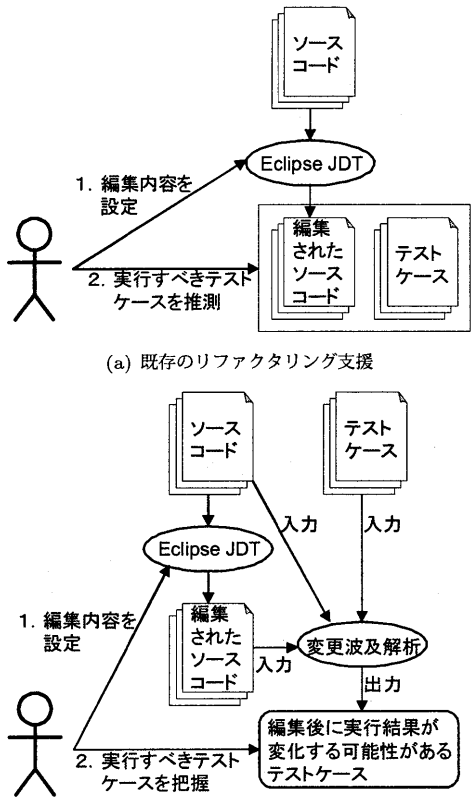

図 1　既存のリファクタリング支援機能と提案手法

択し，使用することが推奨されている [2]．しかし，特にオブジェクト指向プログラムでは，動的に動作が決まる部分があるため，必要なテストケースを開発者が選択することは困難である．

このような問題に対して，Ryder らが提案する変更波及解析と呼ばれる技術を用いると，ソースコードを編集した後に実行結果が変化する可能性があるテストケース，つまり外部的振る舞いの検証に必要なテストケースを，編集内容を基に推測することができるため，リファクタリング支援になると言われている [5]．しかし，リファクタリング支援に変更波及解析を利用したツールや研究は，まだ確認されていない．

本研究では，変更波及解析を用いて，リファクタリングの一環として行うテスト作業を支援するツールを提案する（図 1(b)）．具体的には，JDT のリファクタリング支援機能を拡張し，外部的振る舞いの検証に必要なテストケースを推測した上でリファクタリングを行うことができるようにする．

提案するツールを使用してケーススタディを行い，その結果について議論することで，リファクタリング支援に変更波及解析を利用したツールの有効性を検証する．今回は，リファクタリングパターンの 1 つである Pull Up Method リファクタリング [2] を行い，提案するツールを使用しない場合と使用した場合とでどのような違いがあるかを検証する．

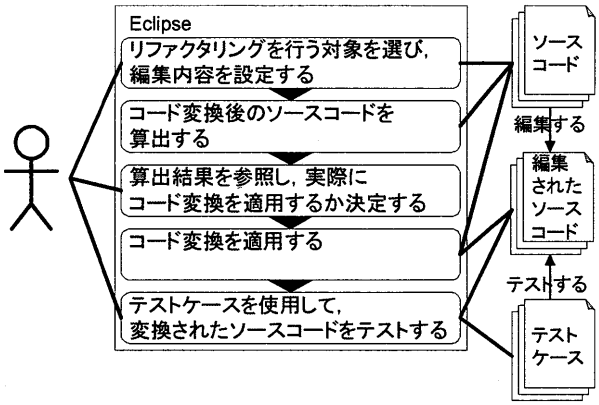

図 2　Eclipse JDT のリファクタリング支援機能を使用する

2. 背　　景

2.1 リファクタリング

リファクタリング [1] とは，外部的振る舞いを保ちつつ，ソースコードの保守性を改善する作業を指す [2]．ここで言う**ソフトウェアの外部的振る舞い**とは，ソフトウェアの仕様と言い換えてもよい．**内部の構造**とは，ソースコードと同義である．ソフトウェアの外部的振る舞いを保つことで，ソフトウェアはソースコードを編集する前と同じ機能を提供することができ，ソフトウェアの利用者は，ソースコードが編集されたことに気が付かない [2]．

リファクタリングを行う上で，信頼できるテストケースがあることは必須条件である [2]．本研究では，**信頼できるテストケース**とは，そのテストケースを使用したテストを行い，ソースコードの編集前後で実行結果が同じであれば，そのテストケースで実行する範囲においては外部的振る舞いが保たれているとみなすことができるテストケースであるとする．信頼できるテストケースが存在しない場合，外部的振る舞いが保たれているか確認できないため，リファクタリングを始める前に信頼できるテストケースを用意する必要がある．

リファクタリングは次に示す手順で行う．

（1） 開発者がソースコード中からリファクタリングを行った方がよい部分を特定する

（2） リファクタリングを行う対象部分の設計を開発者が理解する

（3） リファクタリングを行った後の設計を開発者が考える

（4） 開発者が考えた設計になるように，ソースコードを編集する

（5） 開発者がソフトウェアの外部的振る舞いが保たれていることを確認する

（6） （4），（5）の手順を繰り返し，最終的に開発者が考えた設計にする

Fowler はリファクタリングを行うべきソースコードの状態に応じて，さまざまなリファクタリングパターンを提案している [2]．そして，いくつかのリファクタリングパターンは，ソースコードを編集する作業については統合開発環境 Eclipse のプラグイン JDT [4] で支援されている（図 2）．JDT とは Java 言

語でソフトウェアを開発する際に利用できる Eclipse のプラグインである．JDT を使用する開発者が，リファクタリングを行う対象や行うリファクタリングの内容を設定すれば，そのリファクタリングを行うことを目的としてコード変換を適用したソースコードのプレビューが Eclipse 上に提示される．プレビューが提示された時点ではソースコードは変換されていないので，そのコード変換を実際に行うか選択することができる．

ただし，ソフトウェアの外部的振る舞いが保たれていることを確認する作業は開発者が行わなければならない．リファクタリングを行う開発者は，ソフトウェアの外部的振る舞いが保たれていることを確認するために，テストケースを使用してテストを行い，ソースコードを編集する前と編集した後とでテストの実行結果が同じであることを確認する．このテスト作業は，ソースコードを編集してコンパイルをする度に行うことがリファクタリングの手順に定められている．そのため，リファクタリングには自動的に実行できるテストケースが必要であると言われている [2]．文献 [2] では，自動的に実行できるテストケースを用意するために，JUnit テストフレームワーク [6] を使用している．Eclipse 上で JUnit テストフレームワークを使用すれば，パッケージ，クラス，メソッドのいずれかの単位でテストケースを指定し，実行結果を参照することができる．

また，テストを頻繁に行うため，外部的振る舞いの検証に必要なテストケースのみを使用することが推奨されている [2]．外部的振る舞いの検証に必要なテストケースを絞り込む作業は，JDT でも支援されていないため，開発者がソースコードとテストケースを理解して必要なテストケースを選択する必要がある．

2.2 変更波及解析

変更波及解析とは，ソースコードを編集した影響を特定する技術である [5]．この技術が研究される背景には，オブジェクト指向言語では継承やダイナミックディスパッチを使用しているために，ソースコードを編集した部分が原因で，編集していないソースコードの実行結果が変化することがあり，ソフトウェアを保守することが困難になっているという問題がある [5]．

例えば，Java 言語で記述されたソースコード中にクラス Price を継承したクラス RegularPrice があるとする（図 3）．オブジェクト指向言語では，継承元のクラス (親クラス) の型である変数に継承先のクラス（子クラス）のオブジェクトを代入することができるため，Price 型の変数に RegularPrice クラスのオブジェクトを代入して実行することができる．図 3(a) のように，親クラス Price でのみ getCharge() メソッドを定義している場合，子クラス RegularPrice に getCharge() メソッドが継承されるので，Price クラスに定義されている getCharge() メソッドが実行される．ここで子クラス RegularPrice に，親クラス Price に定義されている getCharge() メソッドと同じシグネチャのメソッドを定義して，オーバーライドをするようにソースコードを編集したとする（図 3(b)）．この場合，getCharge() メソッドを呼び出すソースコードは編集していないが，ダイナミックディスパッチが原因で，実行される getCharge() メソッドは，RegularPrice クラスに定義されている getCharge() メソッドになる．このような状況では，RegularPrice クラスをテストするテストケース

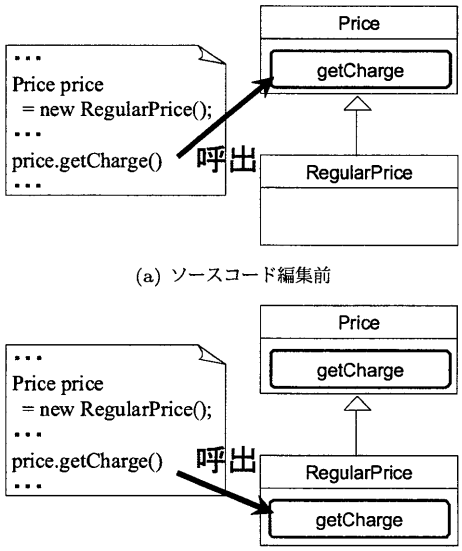

図 3 編集が波及する例

に加えて，ダイナミックディスパッチを利用して RegularPrice クラスのオブジェクトに対して getCharge() メソッドを呼び出すソースコードをテストするテストケースも使用して，外部的振る舞いが保たれていることを確認する必要があると考えられる．

Ryder らが提案する変更波及解析には，編集前のソースコードと編集後のソースコード，それらのソースコードをテストするテストケースを入力する．編集前後のソースコードを比較して編集部分を算出するだけでなく，その編集部分が原因でダイナミックディスパッチが変化する部分を推測し，ソースコードを編集した後に実行結果が変化する可能性があるテストケースを推測することができる．また，実行結果が変化する可能性がある各テストケースについて，実行結果が変化する可能性を引き起こした編集部分を推測することもできる．この変更波及解析を利用することで，回帰テストを行うために必要なテストケースを選択する手間が省け，また，回帰テストの実行結果が失敗であった場合に失敗の原因を特定することが容易になる．

Ryder らが提案する変更波及解析では，メソッドの呼び出し関係を表現するコールグラフを使用している [5] [7]．コールグラフを使用すると，あるメソッドがどのメソッドを呼び出しているか知ることができる．この変更波及解析で使用するコールグラフは，入力として与えたテストケース（この場合，テストケースとはテストメソッドに相当する）を始点とするグラフになる．各ノードがメソッド定義に対応し，各エッジがメソッド呼び出しに対応していて，さらにエッジにはメソッドを呼び出す際の形式についての情報が付加されている．例えば，図 3 では，RegularPrice 型のオブジェクトが Price 型の変数に代入されている状態で getCharge() メソッドを呼び出す，という情報がコールグラフのエッジに付加される．コールグラフは，テストケースと編集前のソースコードを解析したコールグラフと，テストケースと編集後のソースコードを解析したコールグラフとの 2 種類が生成される．

編集前のソースコードを使用したコールグラフとソースコードの編集部分の情報との対応をとると，編集されたメソッドに対応するノードがコールグラフ上にある場合，そのコールグラフの始点であるテストケースはそのメソッドを呼び出しているので，ソースコードを編集する前と同じ実行結果を出力しない可能性がある．また，ダイナミックディスパッチの変化に対応するエッジがコールグラフ上にある場合も同様である．そのようなテストケースが，ソースコードを編集した後に実行結果が変化する可能性があるテストケースと判定される．

また，編集後のソースコードを使用したコールグラフとソースコードの編集部分の情報との対応をとると，テストケースの実行結果はそのテストケースを始点とするコールグラフ上のノードに対応するメソッドの実行結果であるから，コールグラフ上のノードに対応するメソッドに対する編集が，テストケースの実行結果に波及していることが分かる．コールグラフ上のエッジに対応するダイナミックディスパッチの変化についても同様である．

Ryder らが提案する変更波及解析は Chianti [7] という Eclipse のプラグインとして実装されている．Chianti は Java 言語を対象とし，Eclipse でソフトウェア開発を行う際の単位であるプロジェクト 2 つを入力とする．この 2 つのプロジェクトをそれぞれ，編集前のソースコード，編集後のソースコードを含むプロジェクトとして扱う．さらにユーザが編集前のソースコードを含むプロジェクトから変更波及解析に使用するテストケースを選択することで，実行結果が変化する可能性があるテストケースと，実行結果が変化する可能性を引き起こした編集部分についての情報を出力する．

2.3 リファクタリングを行う際に Chianti を使用した場合の問題点

Ryder らは，変更波及解析がリファクタリング支援にも利用できることを述べている [5]．JDT のリファクタリング支援機能と Chianti を単純に併用した場合，次のような手順で開発者はリファクタリングを行うと考えられる．

（1） Chianti は編集前後のソースコードを入力とするため，プロジェクトをコピーし，2 つ用意する

（2） プロジェクト 2 つのうち一方に含まれるソースコードを，JDT のリファクタリング支援機能を利用して変換する（JDT のリファクタリング支援機能で問題が見つかった場合は，リファクタリングを中止する）

（3） 編集前後のソースコードを用意できたので，Chianti を実行する

（4） Chianti で出力されたテストケースを使用して，外部的振る舞いを確認する

（5） テストの実行結果を確認し，リファクタリング成功だと判断したならば，編集前のソースコードを含むプロジェクトを削除する．リファクタリング中止と判断したならば，編集後のソースコードを含むプロジェクトを削除する

この手順において，次のような問題点が考えられる．

- プロジェクトを手動でコピーしなければならない
- 手順 (2) においてリファクタリングを中止した場合，コピーしたプロジェクトを削除しなければならない
- 手順 (5) において，プロジェクト 2 つのうち一方を削除しなければならない．コピーをした際に名前を変更したプロジェクトを残す場合は，プロジェクトの名前を元の名前に変更しなければならない

この問題点となる作業は，機械的に実行することができるため，ツールとして実装すれば解決できると考えられる．しかし今のところ，リファクタリング支援に変更波及解析を利用したツールは確認されていない．

3. 提案するツール

この節では，2.3 節で述べた問題点を解決するために本研究で提案するツールについて説明する．なお本研究では，リファクタリングパターンの 1 種である Pull Up Method リファクタリング [2] を支援する．

提案するツールを使用した際の手順を次に示す（図 4）．

手順 A（対象のメソッドと引き上げ先のクラスを指定）
Pull Up Method リファクタリングにおいて引き上げる対象のメソッドと，そのメソッドを引き上げる先のクラスを指定する．

手順 B（コード変換後のソースコードを確認）
JDT が提示したコード変換後のソースコードを参照して，問題がないことを確認する．

手順 C（Chianti を実行）
Chianti に入力するテストケースを指定し，Chianti を実行する．

手順 D（Chianti が提示したテストケースを用いてテストを実行）
Chianti が実行結果として提示したテストケースを使用してテストを行い，外部的振る舞いを検証する．

手順 E（対象ソースコードにコード変換を適用）
リファクタリング対象であるソースコードに手順 B で参照した内容のコード変換を適用する．

以降，手順 A，B，C，D，E についてそれぞれ詳述する．

3.1 手順 A（対象のメソッドと引き上げ先のクラスを指定）

Pull Up Method リファクタリングにおいて引き上げる対象とするメソッドとそのメソッドを引き上げる先のクラスを開発者が指定する．JDT のリファクタリング支援機能は，開発者が指定した情報を元にリファクタリングを目的としたコード変換を行った後のソースコードを算出する．

3.2 手順 B（コード変換後のソースコードを確認）

JDT が提示したコード変換後のソースコードを参照し，開発者は自分が意図した通りのコード変換が行われるか確認する．また，このときエラーや警告が表示される場合があるので，コード変換に問題がないかどうかも確認する．

3.3 手順 C（Chianti を実行）

Chianti を実行するために必要なテストケースを開発者が指定する．提案するツールは，リファクタリング対象であるソースコードをコピーし，手順 A で算出したコード変換をそのコピーしたソースコードに適用する．コピー元のソースコードとコピーしてコード変換を適用したソースコード，開発者が指定したテストケースを入力として，Chianti を実行し，外部的振る舞いの検証に必要なテストケースを提示する．

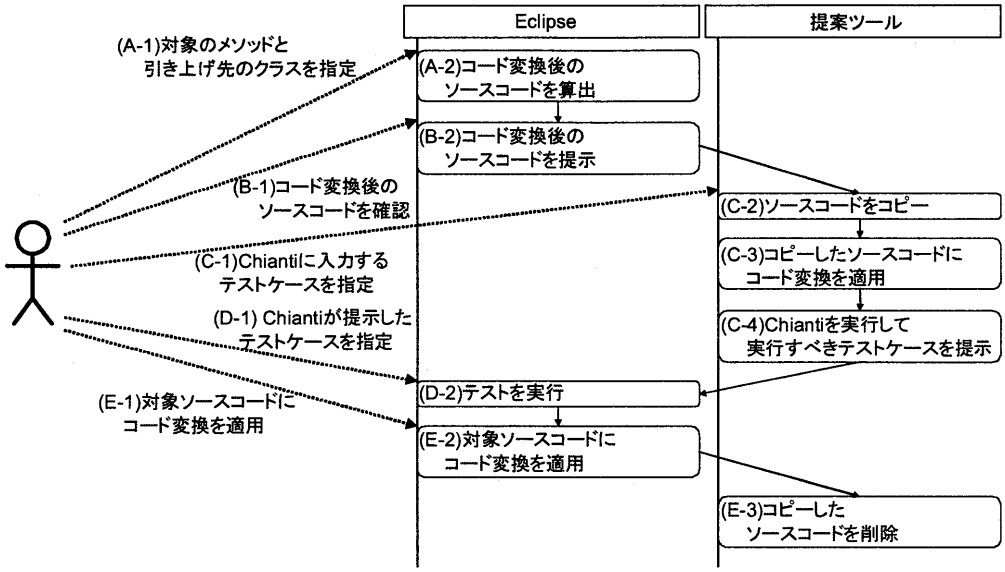

図 4　提案するツールを使用したリファクタリング

3.4　手順 D（Chianti が提示したテストケースを用いてテストを実行）

Chianti が提示したテストケースを使用して，コピーしてコード変換を適用したソースコードに対するテストを行う．開発者はテストの実行結果を参照し，外部的振る舞いが保たれていることを確認する．

3.5　手順 E（対象ソースコードにコード変換を適用）

リファクタリングを行う対象であるソースコードに対して，コード変換を適用する．Chianti を実行するためにコピーしてコード変換を適用したソースコードは削除する．

この提案するツールを使用して Pull Up Method リファクタリングを行うと，外部的振る舞いを検証するために必要なテストケースを選択する手間が省け，外部的振る舞いが検証できないコード変換を知ることができる．よって，より安全に Pull Up Method リファクタリングを行うことができるようになると考えられる．

4.　ケーススタディ

4.1　適　用　例

提案するツールを使用して，次のソースコードに対して Pull Up Method リファクタリングを行った場合について説明する．DiamondShape クラスと Rectangle クラスに定義されている calculateArea() メソッドを親クラス Parallelogram クラスに引き上げる．

```
package drawtool.figure;
public interface Figure {
    abstract public int calculateArea();
}
```

```
package drawtool.figure;
public abstract class Parallelogram implements Figure {
    protected int base;
    protected int height;
    public Parallelogram(int base, int height) {
        this.base = base;
        this.height = height;
    }
}
```

```
package drawtool.figure;
public class DiamondShape extends Parallelogram {
    public DiamondShape(int base, int height) {
        super(base, height);
    }
    public int calculateArea() {
        return this.base * this.height;
    }
}
```

```
package drawtool.figure;
public class Rectangle extends Parallelogram {
    public Rectangle(int base, int height) {
        super(base, height);
    }
    public int calculateArea() {
        return this.base * this.height;
    }
}
```

このソースコードに対して，次のテストケースを用意する．

```
package drawtool.figure;
import junit.framework.TestCase;
public class DiamondShapeTest extends TestCase {
    public void testCalculateArea() {
        Figure figure = new DiamondShape(3, 4);
        assertEquals(12, figure.calculateArea());
    }
}
```

4.2　適 用 結 果

このリファクタリングを提案するツールを使用して行うと，Chianti で推測した結果が図 5 のように表示される．図 5 から読み取ることができる情報は，次の通りである．

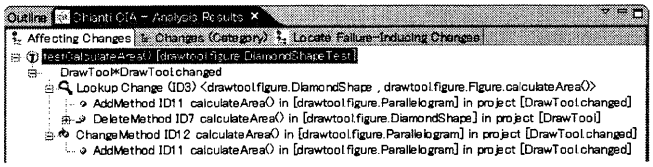

図 5　提案するツールの出力

- DiamondShapeTest クラスの testCalculateArea() メソッドが，ソースコード編集後に実行結果が変化する可能性があるテストケースである
- Figure 型の変数に DiamondShape クラスのオブジェクトを代入して calculateArea() メソッドを呼び出した場合の外部的振る舞いがテストされている
- 抽象クラス Parallelogram に引き上げられた calculateArea() メソッドがテストされている

逆に，Figure 型の変数に Rectangle クラスのオブジェクトを代入して calculateArea() メソッドを呼び出した場合の外部的振る舞いはテストされていないことが分かる．リファクタリングを行うには信頼できるテストケースが必須であるため，この場合は Figure 型の変数に Rectangle クラスのオブジェクトを代入して calculateArea() メソッドを呼び出した場合の外部的振る舞いをテストするテストケースを追加する必要がある．よって，このリファクタリングは中止と判断できる．

4.3 議　　論

4.1 節の例は，信頼できるテストケースが十分に用意されていないために，リファクタリングを中止する例であった．この例では，3 つの点で提案するツールが有効であったと考えられる．

1 つめは，開発者の労力である．提案するツールを使わなかった場合，外部的振る舞いの検証に必要なテストケースを開発者が事前に把握した上でリファクタリングを行わなければならない．4.1 節の例は小規模のソースコードとテストケースであるため，開発者が理解をするために時間はほとんどかからないが，ソフトウェアが大規模になるほど理解に時間がかかると考えられる．提案するツールを使用することで，必要なテストケースを短時間で推測することができ，また，テストケースでテストされていないコード変換を推測することもできる．

2 つめは，開発者が必要なテストケースを推測する際には誤りが入り込む点である．継承やダイナミックディスパッチが原因でオブジェクト指向言語で記述されたソースコードは理解が難しいため，開発者が外部的振る舞いの検証に必要なテストケースの推測を誤ることがあると考えられる．結果として，外部的振る舞いを変化させてしまったのに，外部的振る舞いを検証するためのテストに使用するテストケースの選択を誤って，外部的振る舞いの変化に気が付かない可能性がある．提案するツールを使用することで，外部的振る舞いの検証に必要なテストケースを確実に推測できると考えられる．

3 つめは，テストの実行時間である．外部的振る舞いの検証に必要なテストケースを使用しなかったために，外部的振る舞いの変化を見落とすという事態は，コード変換を適用する度にすべてのテストケースを使用してテストを行うことで，ある程度防ぐことができると考えられる．しかし，ソフトウェアが大規模になるほど大量のテストケースが定義されているため，テストを実行するために時間がかかる．提案するツールを使用すれば，必要最小限のテストケースを使用してテストを行うことができるため，有効だと考えられる．

なお，現状のツールでは，実行結果が変化する可能性があるテストケースをツールで提示しているが，テストを行う際には開発者がツールで提示されたテストケースを選択して Eclipse 上で行うようになっている．テストケースを選択する作業もツールで支援することが可能だと考えられる．

5. ま と め

本研究では，リファクタリング作業の一環であるテスト作業を，Ryder らが提案する変更波及解析という技術を利用して支援するツールを提案した．提案するツールは，JDT のリファクタリング支援機能を拡張し，Pull Up Method リファクタリングを目的としてコード変換を行う前に，コード変換の内容を基に実行結果が変化する可能性があるテストケースを変更波及解析で推測して，ソフトウェアの外部的振る舞いが保たれていることを確認するためのテスト作業で使用するテストケースを提示するツールである．この提案するツールを使用することで，より安全に Pull Up Method リファクタリングを行うことができると考えられる．

今後の課題は，他のリファクタリングパターンもツールで支援することが挙げられる．また，開発者に提示する情報をリファクタリングにより適した形式で提示することも挙げられる．提案するツールの実行時間については，規模が大きいソフトウェアを対象として実験を行うことで実用的な時間で収まるか検証する必要がある．

謝辞　変更波及解析ツール Chianti を提供していただいたバージニア工科大学 Barbara G. Ryder 教授に深く感謝する．本研究は一部，日本学術振興会科学研究費補助金基盤研究（A）（課題番号:21240002），文部科学省科学研究費補助金若手研究（B）（課題番号:20700024），日本学術振興会特別研究員奨励費（課題番号:20・1964）の助成を得た．

文　　献

[1] W. F. Opdyke: “Refactoring object-oriented frameworks”, PhD thesis, University of Illinois at Urbana-Champaign (1992).
[2] M. Fowler: “Refactoring: improving the design of existing code”, Addison Wesley (1999).
[3] Eclipse: “http://www.eclipse.org”.
[4] Eclipse JDT: “http://www.eclipse.org/jdt/”.
[5] B. G. Ryder and F. Tip: “Change impact analysis for object-oriented programs”, the 3rd ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering (PASTE 2001), Snowbird, UT, USA, pp. 46–53 (2001).
[6] JUnit: “http://www.junit.org”.
[7] X. Ren, F. Shah, F. Tip, B. G. Ryder and O. Chesley: “Chianti: A tool for change impact analysis of Java programs”, the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2004), Vancouver, BC, Canada, pp. 432–448 (2004).