修士学位論文

題目

プログラム動作の変更要因を特定するリファクタリング支援ツール

指導教員
井上 克郎 教授

報告者
吉田 昌友

平成 22 年 2 月 8 日

大阪大学 大学院情報科学研究科
コンピュータサイエンス専攻 ソフトウェア工学講座

平成 21 年度 修士学位論文

プログラム動作の変更要因を特定するリファクタリング支援ツール

吉田 昌友

内容梗概

リファクタリングとは，ソフトウェアの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である．リファクタリングでは，一連の変更をソースコードに対して行う．この一連の変更は，複数のクラスやメソッドに対して行う，複数種類の変更からなる列である．そして，ソースコードを変更する度に，外部的振る舞いが保たれていることを確認するため，テスト作業を行う．

統合開発環境 Eclipse には，リファクタリング支援機能がある．この機能を使用すれば，リファクタリング中に行う一連の変更を自動で行うことができる．ただし，ソースコードを変更した後のテスト作業で，テストを行った結果が失敗であった場合，複数のクラスやメソッドに対して行われた一連の変更のうち，どの変更が原因であるのかを開発者が特定することは困難である．また，既存のテストケースではテストできない変更を行っていても，開発者が気が付くことが難しいという問題がある．

これらの問題に対して，Eclipse 上で動作する変更波及解析の機能を使用すると，テストが失敗した原因である変更を特定することができ，かつ，既存のテストケースではテストできない変更を特定することができる．しかし，リファクタリング支援機能と変更波及解析の機能は独立して存在するので，リファクタリング支援機能で適用した一連の変更と，変更波及解析の結果とを対応付けて提示することが困難である．

そこで，Eclipse のリファクタリング支援機能に，変更波及解析の機能を統合したリファクタリング支援ツールを提案する．提案ツールでは，リファクタリング支援機能で適用した一連の変更と，変更波及解析の結果とを対応付けて提示する．リファクタリング支援機能の変更を表すモデルと変更波及解析の機能の変更を表すモデルが異なっていたため，モデル間の対応付けを行った．

本研究では，メソッドの引き上げを行うリファクタリングにおいて適用する一連の変更から，テストが失敗した原因である変更を特定するツールを実装した．そして，ケーススタディを行い，提案するツールが提示した変更の中に，テストが失敗した原因である変更が含まれることを確認できた．

主な用語

リファクタリング
ソフトウェアテスト
変更波及解析

目次

1 まえがき **4**

2 背景 **6**

2.1 リファクタリング 6

2.1.1 リファクタリングパターン 6

2.1.2 リファクタリングの一環として行うテスト作業 9

2.1.3 リファクタリングの問題点 10

2.2 統合開発環境 Eclipse 10

2.2.1 リファクタリング支援機能 10

2.2.2 テスト支援機能 13

2.2.3 リファクタリング支援機能の問題点 13

2.3 変更波及解析 (Change Impact Analysis) 14

2.4 変更波及解析ツール Chianti 18

2.5 統合開発環境 Eclipse を用いてリファクタリングを行う際の問題点 19

3 提案手法 **22**

4 実装 **23**

5 ケーススタディ **27**

6 関連研究 **32**

6.1 JUnit/CIA 32

6.2 動的な影響波及解析 (Dynamic Impact Analysis) 32

6.3 プログラムスライシング 32

6.4 リファクタリング時におけるテストケースの変更 33

6.5 自動リファクタリングにおける条件判定 33

7 むすび **34**

謝辞 **35**

参考文献 **36**

1 まえがき

ソフトウェア開発では一般に，開発が進むほどソースコードの保守性が悪化する．その結果，開発が長期化する，保守作業が困難になるなどの問題が発生する．この問題を解決する方法の一つとして，リファクタリング [4, 11] が挙げられる．

リファクタリングとは，今後行われる保守作業に備えて，ソフトウェアの外部的振る舞いを保ちつつ，理解・修正しやすくなるようにソースコードを変更する作業である．そして，頻繁に行われるリファクタリングの手順がリファクタリングパターンとして定義されている [4, 7].

リファクタリングでは，一連の変更をソースコードに対して行う．この一連の変更は，複数のクラスやメソッドに対して行う，複数種類の変更からなる列である．各変更では，開発者がソースコードを変更した後，ソフトウェアの外部的振る舞いを変化させたことに気が付いたときに，すぐに原因を特定できるようにするため，少量ずつソースコードを変更するように定められている．またリファクタリングでは，ソースコードを変更する度に外部的振る舞いが保たれていることを確認する作業を行うことを手順に定めている．そして，ソースコードを少量ずつ変更し，外部的振る舞いを確認する一連の作業を繰り返して，最終的に理解・修正しやすいソースコードに変更する．

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，テストケースを使用してテストを行い，テスト結果が変更前後で同じであることを開発者が確認する作業が，リファクタリングには必須である [4]．ソースコードを変更した後にテスト結果が変化した場合，外部的振る舞いが変化したと判断し，開発者は直前に行った変更から原因を調べ，作業をやり直す．

いくつかのリファクタリングパターンは，統合開発環境 Eclipse[2] の JDT(Java development tools)[3]（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）に支援機能がある．JDT のリファクタリング支援機能を使用すれば，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るだけで，ソースコードに対する一連の変更を自動で適用することができる．すなわち，手動でリファクタリングを行う場合には少量ずつ行っていたソースコードの変更を，一度に行うことができる．

ただし，JDT を使用してリファクタリングを目的としたソースコードの変更を行い，変更後にテスト結果が変化した場合，複数のクラスやメソッドに対して行われた一連の変更のうち，どの変更が原因であるのかを開発者が特定することは困難であると考えられる．また，JDT では解消していないリファクタリングの問題点として，既存のテストケースではテストできない変更を行っていても，開発者が気が付くことが困難な点が挙げられる．

これらの問題は，特にオブジェクト指向プログラムでは実行時に動作が決まる処理がある

ので，より対処が困難になる．この問題に対して Ryder らは，文献 [15] で提案した変更波及解析 (Change Impact Analysis) という技術で対処でき，リファクタリングを支援できると述べている．変更波及解析とは，ソースコードを変更した影響を特定する技術である [15]. Ryder らが提案した変更波及解析を使用すると，ソースコードの変更が原因でテスト結果が変化する可能性があるテストケースを特定できる．加えて，テスト結果が変化する可能性がある各テストケースについて，そのテストケースのテスト結果を変化させる可能性がある変更を特定することができる．よって，ソースコード変更後にテスト結果が変化した場合，その原因である変更を絞り込むことができる．また，既存のテストケースではテストできない変更を特定することもできる．

Ryder らが提案した変更波及解析は統合開発環境 Eclipse のプラグインである Chianti[14] として実装されている．Chianti をリファクタリングにおいて利用すれば，ソースコード変更後にテスト結果が変化した場合，その原因である変更を絞り込むことができる．また，既存のテストケースではテストできない変更を特定することができる．

しかし，JDT と Chianti は，独立したプラグインであるため，リファクタリング支援機能で適用した一連の変更と，Chianti が出力した，テスト結果を変化させた可能性がある変更や既存のテストケースではテストできない変更とを対応付けて提示することが困難である．

そこで本研究では，JDT のリファクタリング支援機能に変更波及解析の機能を統合したリファクタリング支援ツールを提案する．提案ツールでは，リファクタリング支援機能で適用した一連の変更と，変更波及解析の出力結果を対応付けて提示する．統合するにあたり，リファクタリング支援機能の変更を表すモデルと変更波及解析の変更を表すモデルが異なっていたため，モデル間の対応付けを行った．

本研究では，“メソッドの引き上げ”[4] を行うリファクタリングにおいて適用する一連の変更から，テストが失敗した原因である変更を特定するツールを実装した．そして，ケーススタディを行い，提案するツールが提示した変更の中に，テストが失敗した原因である変更が含まれることを確認できた．

2 背景

2.1 リファクタリング

リファクタリング [11] とは，“ソフトウェアの外部的振る舞いを保ったままで，内部の構造を改善していく作業” を指す [4]．本研究では，“ソフトウェアの外部的振る舞い” とは，テストで確認できるソフトウェアの動作であると解釈する．また，“内部の構造” とは，ソースコードの構造であると解釈する．

リファクタリングは次に示す手順で行う．

手順 1 開発者がソースコード中からリファクタリングを検討すべき部分を特定する

手順 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

手順 3 リファクタリングを行った後の構造を開発者が考える

手順 4 開発者が考えた構造になるように，ソースコードを変更する

手順 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

手順 6 手順 4，手順 5 を繰り返し，最終的に開発者が考えた構造にする

リファクタリングでは，ソースコードを少量ずつ変更し，その度に外部的振る舞いが保たれていることをテストを行うことで確認する．そのため，開発者がソースコードを変更する際に，誤ってソフトウェアの外部的振る舞いを変化させてしまっても，その原因を容易に絞り込むことができ，短時間で修正できる [4]．

リファクタリングは特に，機能追加や，バグ修正，コードレビューの際に行うことを推奨されている [4]．いずれもソースコードの理解が必要な作業であり，リファクタリングを行った結果，ソースコードを深く理解することができ，バグを混入しにくくなり，バグを発見しやすくなる．

また，リファクタリングは，あくまでもソフトウェアを理解しやすく，修正を容易にするために行う作業であり，機能を追加する作業とは区別する必要がある [4]．

2.1.1 リファクタリングパターン

頻繁に行われるリファクタリングの手順がリファクタリングパターンとして定義されている [4, 7]．本節では，リファクタリングパターンのいくつかを説明する．

メソッドの抽出 (Extract Method)[4] メソッド内に，ひとまとめにできるコード片がある場合，そのコード片を抽出して新たなメソッドとして定義し，抽出されたコード片を抽

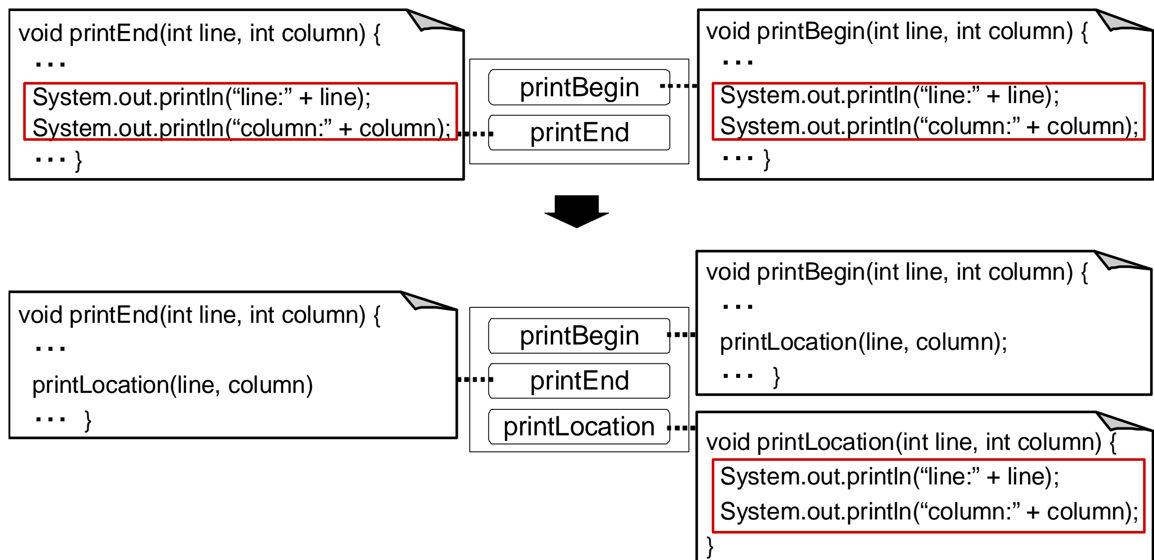


図 1: メソッドの抽出 (Extract Method)[4]

出先のメソッドを呼び出す文に置き換えるリファクタリングである．長すぎて理解することが困難なメソッドを，短いメソッドに分割して処理内容の理解を容易にしたり，複数のメソッド内に同じコード片がある場合に，そのコード片をメソッドとして抽出し，重複したコード片を取り除いて修正を容易にする目的で，このリファクタリングは行われる．重複したコード片を取り除くために，メソッドの抽出を行う例を図 1 に示す（例は Java 言語で記述されている．以下同じ）．

図 1 の例に対して，文献 [4] に記載されている手順でメソッドの抽出を行う場合，以下の手順で作業を行うと考えられる（リファクタリングを検討すべきメソッドはすでに特定され，メソッドの抽出を行って構造を変更することを開発者が決めた後である）．

1. printLocation メソッドを新たに定義する
2. printBegin メソッド，または，printEnd メソッドから，抽出するコード片を printLocation メソッドにコピーする
3. printLocation メソッド内には，printBegin メソッドや printEnd メソッドの line 引数と column 引数を参照している処理がある．よって，printLocation メソッドのシグネチャを，line 引数と column 引数があるシグネチャに変更する
4. ソースコードをコンパイルする
5. printBegin メソッド内の抽出されたコード片を，printLocation メソッドの呼出に置き換える

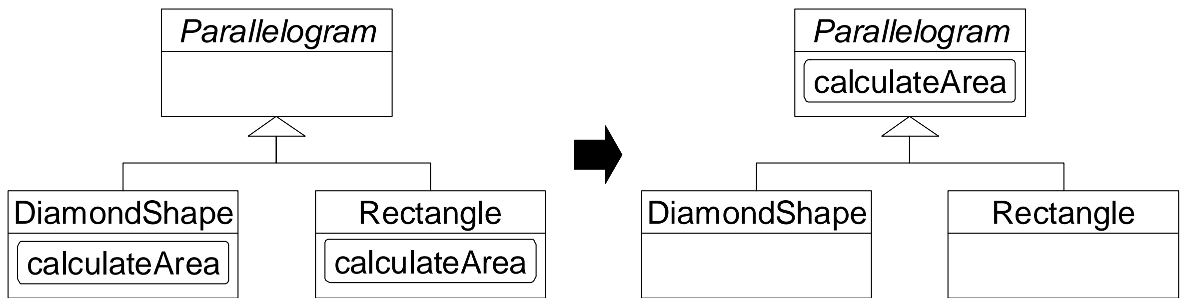


図 2: メソッドの引き上げ (Pull Up Method)[4]

6. ソースコードをコンパイルして，テストを行い，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
7. printEnd メソッド内の抽出されたコード片を，printLocation メソッドの呼出に置き換える
8. ソースコードをコンパイルして，テストを行い，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

メソッドの引き上げ (Pull Up Method)[4] 同じ処理を行うメソッドが複数の継承先のクラス（子クラス）に存在した場合，それらを継承元のクラス（親クラス）に引き上げる．図 2 に示した例は，複数のメソッドがまったく同じ内容を持つ，最も単純な例である．

図 2 の例に対して，文献 [4] に記載されている手順でメソッドの抽出を行う場合，以下の手順で作業を行うと考えられる（リファクタリングを検討すべきクラスとメソッドはすでに特定され，メソッドの引き上げを行って構造を変更することを開発者が決めた後である）．

1. DiamondShape クラスと Rectangle クラスに定義されている calculateArea メソッドの処理内容が同一であることを確認する
2. DiamondShape クラス，または，Rectangle クラスから，calculateArea メソッドを Parallelogram クラスにコピーする
3. ソースコードをコンパイルする
4. DiamondShape クラスから calculateArea メソッドを削除する
5. ソースコードをコンパイルして，テストを行い，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. Rectangle クラスから calculateArea メソッドを削除する

7. ソースコードをコンパイルして，テストを行い，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

また，複数の子クラスに存在するメソッドの内容が一部異なる場合，メソッドの抽出を行えば，メソッドの内容を一致させ，メソッドの引き上げを行うことができる場合がある．

2.1.2 リファクタリングの一環として行うテスト作業

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では，開発者がテスト作業を行う．リファクタリングでは，ソースコードの変更前後でテスト結果が一致すれば，外部的振る舞いが保たれていると判断する．逆に，ソースコード変更後にテスト結果が変化した場合は，外部的振る舞いが変化したと判断し，開発者は作業をやり直す．

したがって，リファクタリングを行う上で，信頼できるテスト (solid tests) があることが必須条件である [4]．そのため，リファクタリングを行う前に，信頼できるテストケースを用意する必要がある．

本研究では，信頼できるテストケースとは，仕様に基づいたテストケースであり，かつ，そのテストケースを使用したテストを行い，ソースコードの変更前後でテスト結果が一致すれば，そのテストケースで実行する範囲においては，ソフトウェアの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．

リファクタリングでは，ソースコードを少量ずつ変更する度にテスト作業を行うので，テスト作業にかかる手間をできるだけ省くことが推奨される．文献 [4] では，テスト結果を自動で判定するプログラム形式のテストケース（自己テストコード）を作成することが推奨されている．また，テストの実行時間を短縮するため，ソースコードを変更したことで外部的振る舞いが変化した可能性がある部分をテストするテストケースのみを選択して，テストを行うことが推奨されている [4]．

JUnit テストフレームワーク

文献 [4] では，テスト結果を自動で判定する自己テストコードを作成する方法について，JUnit テストフレームワーク [5] を使用する方法を紹介している．JUnit テストフレームワークは，Java 言語で開発されたソフトウェアの単体テストを支援する．

JUnit テストフレームワークでは，Java 言語のソースコードとしてテストケースを作成する．一般に，ソースコードのメソッド 1 つを，JUnit テストフレームワークを使用した自己テストコードのメソッド（テストメソッド）1 つでテストするように作成する．本研究では，テストメソッド 1 つを 1 つのテストケースとして扱う．

2.1.3 リファクタリングの問題点

本研究に関係する，リファクタリングを行う際の問題点について説明する．

Roberts と Brant は文献 [4] で，手動でリファクタリングを行うと時間がかかるので，開発者がリファクタリングを避ける，と指摘している．ソフトウェアの外部的振る舞いを保ちつつソースコードを変更するために，開発者がソースコードの構造を確認し，どのように変更するか考案する作業や，テスト作業に時間がかかる問題がある．

また，Opdyke は文献 [4] で，“開発者が自分たちのプログラムをリファクタリングしようと思わない理由” の 1 つとして，“リファクタリングは既存のプログラムを壊してしまう” 心配が開発者にあることを述べている．すなわち，ソースコードを変更する際に，新たなバグを混入する可能性があることが問題である．さらに Opdyke は，オブジェクト指向プログラムでは継承を使用しているので，テストケースが複雑になり，ほぼ間違いなくテストケースでテストできない状況が発生することを指摘している．

2.2 統合開発環境 Eclipse

本節では，Java 言語でソフトウェア開発を行う際に広く使用されている，統合開発環境 Eclipse[2] に統合されているリファクタリングに関連する機能について説明する．

2.2.1 リファクタリング支援機能

文献 [4] に記載されているリファクタリングパターンのいくつかは，統合開発環境 Eclipse の JDT (Java development tools)[3] （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）に支援機能がある．

JDT のリファクタリング支援機能では，リファクタリングを目的としてソースコードを変更する作業が，自動化されている．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るだけで，開発者が意図した構造のソースコードに自動で変換することができる．開発者が設定した内容では，適切にソースコードを変換することができないと判定した場合は，エラーを表示する．

JDT のリファクタリング支援機能を使用すれば，以下に示す 2 つの作業にかかる時間を削減でき，リファクタリングにかかる時間を削減できる．

- ソースコードをどのように変更するか開発者が考える時間
- 外部的振る舞いが保たれていることを確認する時間

後者については，手動でリファクタリングを行う場合は少量ずつ行っていたソースコードに対する一連の変更を，リファクタリング支援機能を使用すれば，一度に行うことができる

ため，コンパイルして，テストを行い，外部的振る舞いが保たれていることを確認する作業を行う回数が少なくなるので，時間を削減できるからである．

2.1.1 節で使用した 2 つの例を用いて，JDT のリファクタリング支援機能を使用する例を示す．1 つ目の例である図 1 に示したメソッドの抽出を JDT のリファクタリング支援機能を使用して行う場合，開発者は以下の情報を設定する．

1. 抽出する対象のコード片を，printBegin メソッドから選択する
2. 抽出先のメソッドのシグネチャを設定する．図 1 の例では，メソッド名を printLocation とし，引数の名前をそれぞれ line，column とする

以上の通り開発者が設定すると，デフォルトで，printEnd メソッドからも，選択したコード片と同一のものが抽出される．そしてコード変換を実行すれば，printBegin メソッドと printEnd メソッドから同一のコード片が printLocation メソッドとして抽出されたソースコードに，自動で変換される．

JDT のリファクタリング支援機能では，printBegin メソッド，printEnd メソッドで使用している引数，ローカル変数のうち，抽出先の printLocation メソッドで使用するものを自動的に判定し，printLocation メソッドのシグネチャの雛形を生成する．そのため，開発者が抽出先メソッドのシグネチャを考案する手間が削減できる．

2 つ目の例である図 2 に示したメソッドの引き上げを JDT のリファクタリング支援機能を使用して行う場合，開発者は以下の情報を設定する．

1. 引き上げる対象のメソッドとして DiamondShape クラスの calculateArea メソッドを選択する
2. 引き上げる先のクラスとして Parallelogram クラスを選択する（Parallelogram クラスは抽象クラスなので，calculateArea メソッドを抽象メソッドとして Parallelogram クラスに宣言する変換もできる）
3. 同時に引き上げる対象のメソッドとして Rectangle クラスの calculateArea メソッドを選択する

そしてコード変換を実行すれば，2 つのメソッドが引き上げられた構造のソースコードに，自動で変換される．

以上 2 つの例では，一連の変更が一度に行われるため，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を行う回数が 1 回で済む．よって，リファクタリングにかかる時間が削減できる．

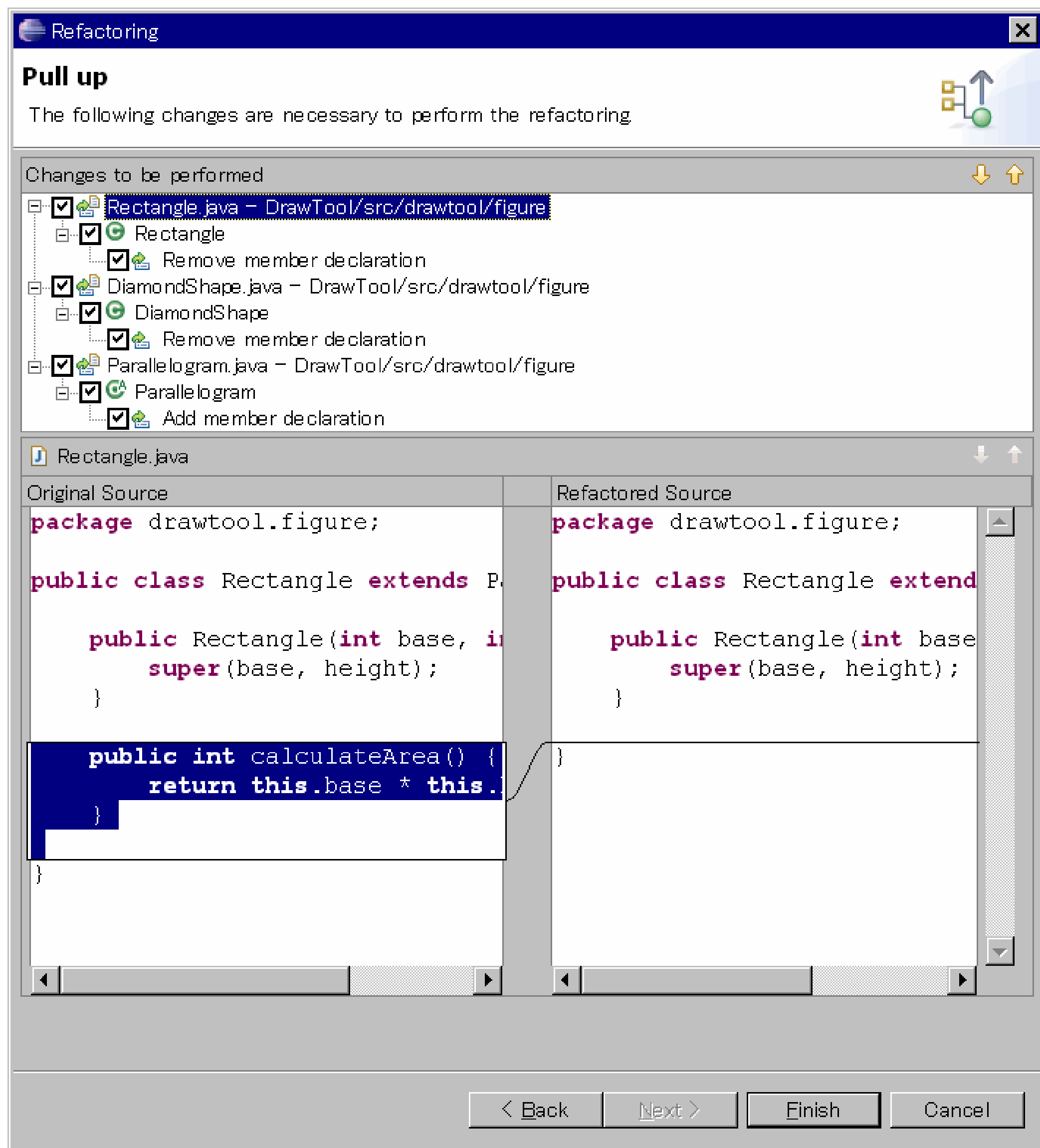


図 3: プレビュー画面

また，変換を実行する前に，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．プレビュー画面の一例を図 3 に示す．リファクタリングを目的として行うコード変換は，図 3 から読み取れるように，“メンバの宣言を削除 (Remove member declaration)” など，複数の細かい変換の集合で表現されている．プレビュー画面では，その細かいコード変換ごとに，どのようにソースコードが変換されるのか確認することができる．開発者はプレビュー画面で，細かいコード変換のうち，不適切と判断した変換を実行しないように指定できる．また，プレビュー画面が表示されている時点では，実際のソースコードはまだ変換さ

れていないので，リファクタリングを中止することもできる．

JDT では，リファクタリングを目的としたコード変換の内容を算出するために，ソースコードを解析し，ソフトウェアの外部的振る舞いを保ちつつ変換できるか条件判定を行っている．この条件判定で，コンパイル可能なソースコードに変換できない，または，外部的振る舞いが変化する変換しかできないと判定した場合は，エラーメッセージを表示し，コード変換を実行しない．

ただし，次に示す理由で，リファクタリングを自動化するツールがソースコードを解析して行っている条件判定は，外部的振る舞いを保ちつつ変換できることをある程度までしか保証しない場合がある [4]．

- ソースコード解析と条件判定に時間がかかりすぎると，開発者がツールを使用しようと思わない
- 条件判定を厳しくするほど，ツールを適用できる機会が少なくなる

例えば図 2 では，Parallelogram クラスを継承する別の子クラスを Eclipse 上で扱っていなかった場合，その子クラスについてはソースコードを解析することができない．この場合，Eclipse のバージョン 3.1 に統合されている JDT のリファクタリング支援機能では，Parallelogram クラスを継承する別のクラスの有無については何も表示しない．開発者が変換内容に問題がないかどうかを判断し，コード変換を実行するか選択できるようにしている．

2.2.2 テスト支援機能

JDT にはテスト支援機能として，JUnit テストフレームワークの使用を支援する以下の機能がある．

- JUnit テストフレームワークを使用する自己テストコードの雛形を生成する機能
- ソースフォルダ（パッケージの最上位に相当するフォルダを格納しているフォルダ），パッケージ，クラス，メソッドのいずれかの単位で，JUnit テストフレームワークを使用した自己テストコードを実行する機能
- 失敗したテストメソッドや発生したエラーを選択して，ソースコード上の該当部分をエディタで開く機能

2.2.3 リファクタリング支援機能の問題点

JDT のリファクタリング支援機能の問題点を説明するにあたって，本研究では，JDT を使用してリファクタリングを行う場合でも，開発者の判断が入るため，依然としてテスト作

業が必要であると仮定する．

テスト結果が変化した場合の原因特定が困難

JDT を使用してリファクタリングを行う場合，手動リファクタリングでは個別に行う一連の変更を，一度に適用している．したがって，テスト結果が変化した場合に，一連の変更のうち，どの変更が原因であるかを開発者が特定することが困難になると考えられる．

外部的振る舞いが保たれていることを確認されない部分の特定が困難

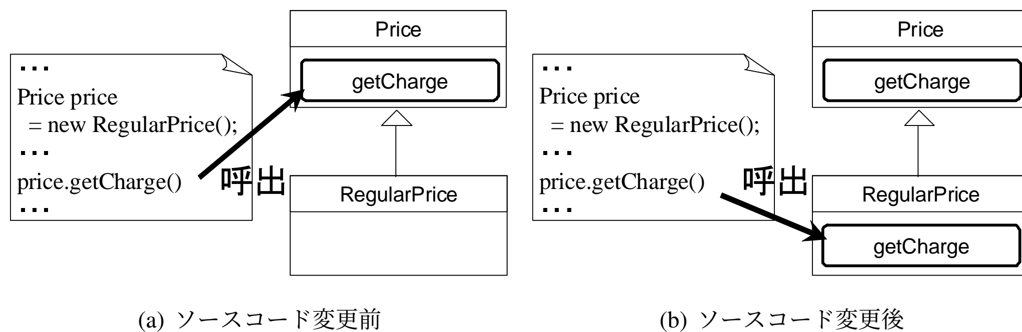
JDT では解消していないリファクタリングの問題点である．特に，オブジェクト指向プログラムでは継承を使用しているので，ソフトウェアの外部的振る舞いが変化する可能性がある部分を開発者が特定することが困難である．したがって，外部的振る舞いが変化する可能性がある部分をテストするテストケースを把握することも困難である．2.1.3 節で説明した通り，既存のテストケースではテストできない状況が発生している可能性が残っている問題がある．

2.3　変更波及解析 (Change Impact Analysis)

変更波及解析とは，ソースコードを変更した影響を特定する技術である [15]．この技術が研究される背景として，保守作業を困難にする要因の 1 つに，ソースコードの一部を変更すると，変更部分だけでなく他の部分の振る舞いが変化する可能性があることが指摘されている [14, 15]．変更していない部分の振る舞いが変化する可能性があるため，回帰テスト（変更後の振る舞いが仕様を満たしているかを確認するためのテスト）では，変更部分のテストだけでなく，他の部分についてもテストを検討する必要が生じる．特に，保守対象がオブジェクト指向プログラムである場合，Dynamic Dispatch（同じ型の参照型変数であっても，実行時におけるインスタンスの型に依存して呼び出される手続きが変化すること）が原因で，開発者にとって回帰テストを検討する部分を特定することが困難になる [14]．

図 4 に，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 がある例を示す．オブジェクト指向プログラムでは，親クラスの型である変数に子クラスのインスタンスを代入できるので，Price 型の変数に RegularPrice クラスのインスタンスを代入して getCharge メソッドを呼び出すことができる．

図 4(a) に示すように，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，図 4 のソースコードを実行すると，Price クラスに定義されている getCharge メソッドが実行される．



(a) ソースコード変更前　(b) ソースコード変更後

図 4: 変更が波及する例

ここで図 4(b) に示すように，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変更したとする．この場合，getCharge メソッドを呼び出すソースコードは変更していないが，Dynamic Dispatch が原因で，実行される getCharge メソッドは，子クラス RegularPrice に定義されている getCharge メソッドになる．この状況では，RegularPrice クラスをテストするテストケースに加え，Dynamic Dispatch 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，変更前のソースコードと変更後のソースコード，それらのソースコードをテストするテストケース（自己テストコードのテストメソッド）を入力する．変更前後のソースコードを比較して変更部分を算出するだけでなく，その変更部分が原因で Dynamic Dispatch が変化する部分を算出する．そして，それらソースコードの変更 (atomic changes[15]) が原因で実行結果が変化する可能性があるテストケース (Affected Tests[15]) を算出することができる．加えて，実行結果が変化する可能性がある各 Affected Test について，実行結果を変化させる可能性がある変更 (Affecting Changes[15]) を算出することもできる．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自動で絞り込むことができる．また，回帰テストの実行結果が失敗であった場合に，失敗の原因となった変更を絞り込むことができる．

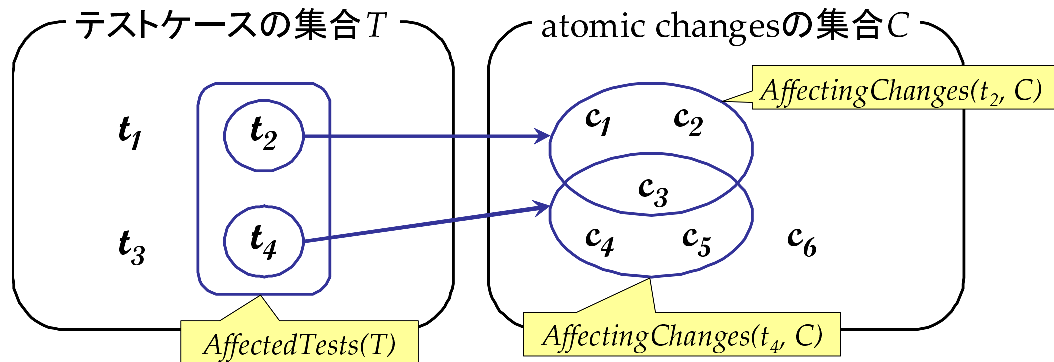


図 5: Affected Tests と Affecting Changes

図 5 を使用して例を示す．以下に示す定義をする．

$$
\begin{aligned}
T &: \text{テストケースの集合} \\
C &: \text{atomic changes の集合} \\
\mathit{AffectedTests}(T) &: \text{テストケース集合 } T \text{ の要素中で，実行結果が変化する} \\
&\quad \text{可能性があるテストケースからなる部分集合} \\
\mathit{AffectingChanges}(t, C) &: \text{atomic changes の集合 } C \text{ の要素中で，テストケース } t \text{ の} \\
&\quad \text{実行結果を変化させる可能性がある atomic changes} \\
&\quad \text{からなる部分集合}
\end{aligned}
$$

図 5 では，以下に示す結果を表現している．

$$
\begin{aligned}
T &= \{t_1, t_2, t_3, t_4\} \\
C &= \{c_1, c_2, c_3, c_4, c_5, c_6\} \\
\mathit{AffectedTests}(T) &= \{t_2, t_4\} \\
\mathit{AffectingChanges}(t_2, C) &= \{c_1, c_2, c_3\} \\
\mathit{AffectingChanges}(t_4, C) &= \{c_3, c_4, c_5\}
\end{aligned}
$$

この結果から，回帰テストを行うために必要なテストケースは，$\{t_2, t_4\}$ であると判断できる．また，仮に，テストケース t_4 の実行結果が失敗であるとすると，その原因は，$\{c_3, c_4, c_5\}$ に含まれると判断できる．加えて，テストケースの集合 T が既存のテストケースすべてを含む場合，既存のテストケースでは atomic change c_6 をテストできないと判断できる．

Ryder らが提案する変更波及解析では，Affected Tests と Affecting Changes を算出するために，メソッドの呼出関係を表現するコールグラフを使用している [14, 15]．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という情報を取得できる．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メソッドを始点とす

るグラフである．グラフの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出に対応している．さらにエッジにはメソッドを呼び出す際の形式を表現する情報が付加されている．例えば，図 4 では，RegularPrice 型のインスタンスが 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，変更前のソースコードとテストメソッドを解析して生成したコールグラフと，変更後のソースコードとテストメソッドを解析して生成したコールグラフの 2 種類がある．

以降，変更波及解析の出力について説明する．

atomic changes 変更波及解析ではまず，入力された 2 つのソースコードの差分から変更内容を算出する．

変更内容は atomic changes という細かい単位の変更の集合として表現される．atomic changes は，クラス，フィールド，メソッド，初期化子に対する変更・追加・削除という単位である．

また，Dynamic Dispatch の変化については，Lookup Change[15] と呼ばれている．実行時におけるインスタンスの型，変数の参照型，呼び出すメソッドという情報から構成される．

メソッドの追加を atomic changes で表現する場合，メソッドの追加とメソッドの変更の 2 つの atomic changes で表現する．そして，メソッドを追加した上でメソッドの変更を行うので，この 2 つの atomic changes には依存関係がある．その他にも atomic changes には依存関係が存在する．

Affected Tests 変更前のソースコードとテストメソッドを解析して生成したコールグラフと atomic changes との対応をとる．変更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に対応するテストメソッドは変更されたメソッドを呼び出しているので，ソースコードを変更する前と同じテスト結果にならない可能性がある．また，Dynamic Dispatch 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，そのコールグラフの始点に対応するテストメソッドがソースコード変更前と同じテスト結果にならない可能性がある．それらのテストメソッドが，ソースコードを変更した後にテスト結果が変化する可能性があるテストメソッドと判定され，Affected Tests として出力される．

Affecting Changes 変更後のソースコードとテストメソッドを解析して生成したコールグラフと atomic changes との対応をとる．テストメソッドの実行結果は，そのテストメソッドが呼び出すメソッドの振る舞いに依存して決まる．よって，テストメソッドに対応

図 6: Chianti における Affected Tests と Affecting Changes

するノード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に対する atomic changes が原因で，そのテストメソッドの実行結果が決まる．コールグラフ上のエッジに対応する Dynamic Dispatch の変化についても同様である．それらの atomic changes が，Affected Tests の振る舞いを変化させる可能性があると判定され，Affecting Changes として算出される．

2.4 変更波及解析ツール Chianti

Ryder らが提案する変更波及解析は，Chianti[14] という統合開発環境 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，変更前のソースコード，変更後のソースコードを含むプロジェクトとして扱う．さらに開発者が変更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メソッドを選択すれば，Affected Tests と，各 Affected Tests についてテスト結果を変化させる可能性がある Affecting Changes を提示する．合わせて，atomic changes の集合も種類ごとに提示される．図 6 と図 7 に Chianti が提示する情報の一例を示す．

Chianti は回帰テストが失敗した場合に，その原因を絞り込む目的で開発されている．したがって，ある Affected Test について，その Affected Test に対する Affecting Changes を参

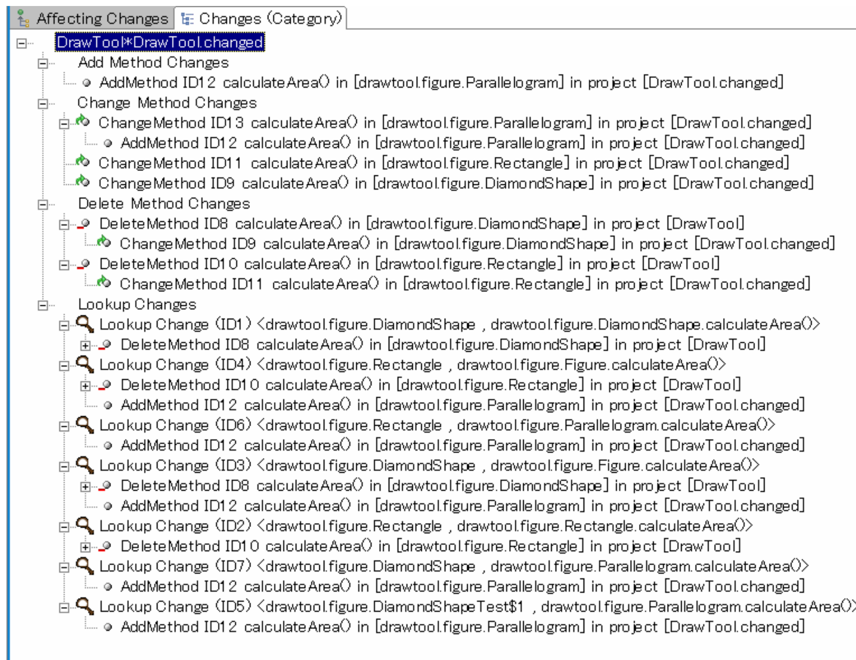


図 7: Chianti における atomic changes

照することはできるが，atomic changes から Affecting Changes，Affected Tests を参照することは困難である．

2.5 統合開発環境 Eclipse を用いてリファクタリングを行う際の問題点

Ryder らは，変更波及解析がリファクタリング支援に利用できることを述べている [15]．これは，2.2.3 節で説明した問題点が，変更波及解析で解決できるからだと考えられる．

テスト結果が変化した場合に原因を特定する際，開発者は変更波及解析の Affecting Changes に絞って調べればよいので，リファクタリング支援になる．テストケースでテストしていない状況については，atomic changes と Affecting Changes の差分を算出すれば，既存のテストケースではテストできない atomic changes を特定できる．この atomic changes は，ソフトウェアの外部的振る舞いが変化する可能性があるのに，テストされない部分に対応する．この部分をテストするテストケースを追加すれば，Opdyke が指摘している，テストケースでテストしていない状況が発生していても，解決することができると考えられる．

そこで，JDT のリファクタリング支援機能と合わせて，Chianti を単純に利用した場合を考える．開発者は次に示す手順でリファクタリングを行うと仮定する．

手順 1 Chianti は変更前後のプロジェクトを入力とするので，プロジェクトをコピーし，2

つ用意する

手順 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使用して変換する（コピーする際に名前を変更したプロジェクトのソースコードを変換すると仮定する．また，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）

手順 3 変更前後のプロジェクトを用意できたので，Chianti を実行する

手順 4 Chianti の出力結果を参照して，ソフトウェアの外部的振る舞いが変化する可能性がある部分がテストされていることを確認する

手順 5 Chianti で出力されたテストケースを使用して，テスト作業を行い，外部的振る舞いを確認する

手順 6 テスト結果を確認し，外部的振る舞いが保たれている場合は，変更前のソースコードを含むプロジェクトを削除し，変更後のソースコードを含むプロジェクトの名前を元に戻す.

手順 7 外部的振る舞いが変化した場合は，Chianti の出力結果を参照して，原因を絞り込み，特定する

手順 8 リファクタリングを目的としたコード変換の一部を調整すれば，外部的振る舞いを保つことができると判断した場合は，JDT のリファクタリング支援機能を使用して行った変換を取り消し，コード変換の内容を調整して，再度実行する

手順 9 外部的振る舞いを保ちつつ変換することができないと判断した場合は，リファクタリングを中止し，変更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，次に示す問題点が考えられる．

1. 手順 4 において，Chianti が提示した情報から，Affecting Changes に含まれない atomic changes を開発者が読み取らなければならない
2. 手順 8 において，JDT のリファクタリング支援機能を使用して行ったコード変換の内容と，Chianti が提示する情報とを，開発者が対応付けなければならない
3. プロジェクトを手動でコピー・削除しなければならない

1. は，回帰テストを支援することを主な目的として Chianti が実装されているために，ある atomic change が Affecting Change であるか提示していない，かつ，ある Affecting Change に対応する Affected Tests を提示していないので，起きている問題だと考えられる．

2. は，JDT のリファクタリング支援機能と，Chianti とが，独立した機能であるために起きている問題だと考えられる．

3. は，Chianti が入力として，変更前後のソースコードを含むプロジェクトを必要とするために起きている問題である．

3　提案手法

2.5 節で述べた問題点を解消するために，JDT のリファクタリング支援機能に Chianti の変更波及解析を統合した，よりリファクタリング支援に有用なツールを提案する．

具体的には，次の 2 つの支援機能を提供するツールを提案する．

1. コード変換を適用する前に，外部的振る舞いを変化させるコード変換を絞り込み，提示する機能
2. リファクタリングを目的として行う細かいコード変換のうち，テストケースでテストされないコード変換を特定し，提示する機能

1. の機能を利用すれば，開発者は，コード変換の一部を適用しないようにして対応する，または，リファクタリングを中止することができる．

2. の機能を利用すれば，開発者はテストケースが不足していることを知ることができる．テストケースを追加して，既存のテストケースではテストできていなかった状況をテストできるようになる．

提案するツールの機能を実現するためには，JDT のリファクタリング支援機能における一連のコード変換の内容と，Chianti の出力結果を対応付ける必要がある．この対応付けを行う上での問題点は，リファクタリング支援機能における変換内容を表すモデル（コード変換記述）と，Chianti における変更内容を表すモデル（atomic changes）とが異なる点である．コード変換記述は，追加する文字列，削除する文字数，変換対象のクラス，変換を適用するソースコード内の位置という情報で構成されている．atomic changes は，変更の対象となったメソッド，フィールド，クラス，初期化子と，それらの追加，削除，変更という情報で構成されている．

リファクタリング支援機能と Chianti における 2 つの変更モデルを統合する方法について説明する．図 3 のプレビュー画面から，JDT のリファクタリング支援機能を使用する開発者が調整できるコード変換の単位は，メソッドやフィールド単位であることがわかっている．変更波及解析における変更の単位もメソッドやフィールド単位である．よって，コード変換記述にある，変換を適用するソースコード内の位置情報から変換対象であるメソッドやフィールドの情報を取得すれば，atomic changes に対応付けることができる．

また，リファクタリング支援機能の中で Chianti の変更波及解析を実行するために，リファクタリング支援機能の実行途中で，ソースコードを変換する前後のプロジェクトを用意する必要がある．プレビュー画面でソースコードの変換内容を開発者が調整できることを踏まえて，プレビュー画面で開発者がソースコードの変換内容を調整した後に変換後のプロジェクトを生成するようにしなければならない．

4　実装

本研究では，3 章で述べた手法の一部を実装した．本章では，その実装内容について述べる．

実装した処理を次に示す.

- リファクタリングを目的として変換した後のソースコードを含むプロジェクトを生成する処理
- リファクタリングパターンの 1 つであるメソッドの引き上げ [4] において，JDT のリファクタリング支援機能のコード変換記述と，Chianti の atomic changes とを対応付ける処理
- メソッドの引き上げにおいて，テストが失敗した原因であるコード変換を絞り込む処理

JDT のリファクタリング支援機能に変更波及解析を統合するため，Eclipse バージョン 3.1 に統合されている既存のプラグインを拡張する実装を行った．Eclipse のプラグインは Java 言語で開発されている．

本研究では，Eclipse のリファクタリング支援機能を実装している，以下のプラグインを拡張している．

- Refactoring Core プラグイン
- Refactoring UI プラグイン
- Text プラグイン

Refactoring Core プラグインと Refactoring UI プラグインは，リファクタリング支援機能のうち，プログラミング言語に依存しない機能を実装している．これらのプラグインを拡張して，Java 言語を扱う Java Development Tools UI プラグインに，Java 言語を対象とするリファクタリング支援機能が実装されている．

また，リファクタリング支援機能において，ソースコードの変換を行う機能は，Text プラグインで実装されている．提案手法を実装するために，この Text プラグインも改変した．

変換後のプロジェクトを生成する処理

Chianti を使用した変更波及解析を行うには，入力として，変更前後のプロジェクトが必要である．そこで，コード変換後のプロジェクトを自動で生成する処理を実装した．

コード変換後のプロジェクトを生成する処理は，2 つの処理から構成される．

1. リファクタリングの対象になっているプロジェクトをコピーする処理
2. コピーしたプロジェクトに，開発者がプレビュー画面で設定した内容のコード変換を適用する処理

これらの実装を説明する上で，Eclipse におけるリソースについて説明する．Eclipse におけるリソースとは，ファイルやフォルダのことである．Eclipse 上でソフトウェアを開発する単位であるプロジェクトはフォルダの一種であり，リソースである．さらに Java 言語で開発を行う際のリソースは，Java 要素で表現されている．

1. 対象プロジェクトのコピー プロジェクトをコピーするためには，実行しているリファクタリング機能が対象にしているプロジェクトを取得する必要がある．対象にしているのは，Java 言語で開発をしているプロジェクトであるため，JDT から情報を取得する．Java Development Tools UI プラグインに含まれる，Refactoring Core プラグインの Refactoring クラスを継承しているクラスのインスタンスを取得して，対象プロジェクトを取得している．ここで取得したプロジェクトの情報は，Java Development Tools Core プラグインに定義されている IJavaProject という Java 要素の一種として表現されている．

そして IJavaProject を，Core Resource Management プラグインに定義されている一般のプロジェクトを表す IProject に変換し，copy メソッドを呼び出してコピーする．

2. コピーしたプロジェクトにコード変換を適用 リファクタリング支援機能におけるコード変換の内容は，Refactoring Core プラグインの Change クラスで表現されている．この Change を取得し，コピーしたプロジェクトのソースコードに対して適用すればよい．

メソッドの引き上げにおいて，Change クラスの実行時におけるインスタンスの型で，変換対象のソースコードについての情報を保持しているのは，Java Development Tools UI プラグインに定義されている CompilationUnitChange である．この CompilationUnitChange にある変換対象のソースコードを指す CompilationUnitChange をコピー後のプロジェクト内のソースコードを指す CompilationUnitChange に置き換えたインスタンスを生成する．

さらに CompilationUnitChange からは，ソースコードの変換内容を表現する TextEdit を取得することができる．TextEdit は，Text プラグインに定義されている，テキストの変換を行うためのクラスである．TextEdit は各インスタンスが一度しか変換にしようできないように定義されているので，本研究のように，2 つのソースコードに適用する場合は，TextEdit をコピーして変換を適用する必要がある．さらに TextEdit には，

開発者がプレビュー画面で適用するかどうかをチェックボックスで設定した情報がある．したがって，チェックボックスで設定した情報もコピーし，プレビュー画面で開発者が設定したコード変換を適用したプロジェクトを生成する．

変更モデルの対応付け

プレビュー画面に表示されているコード変換の単位には，Refactoring Core プラグインに定義されている TextEditChangeGroup クラスが対応する．したがって，TextEditChangeGroup と atomic changes を対応付ければよい．

メソッドの引き上げでは，変換の対象がメソッドであるので，Java 言語におけるメソッドを表現する Java 要素である IMethod を TextEditChangeGroup から取得すれば，TextEditChangeGroup と atomic changes を対応付けることができる．

Chianti で算出した atomic changes からは，対象の IMethod を取得することができる．しかし，TextEditChangeGroup クラスには，対象の IMethod を直接取得する処理は存在しない．

そこで，TextEditChangeGroup から，変換を適用するソースコードを表す ICompilationUnit と，変換内容を表す TextEdit を取得する．TextEdit からは，テキストの変換位置を表すオフセット値（ファイル先頭からの byte 値）が取得できる．ICompilationUnit には，オフセット位置からメソッドやフィールドを取得するメソッドがある．この 2 つを組み合わせて，TextEditChangeGroup が対象にしている IMethod を取得する．

ただし，メソッドを削除する TextEditChangeGroup と，メソッドを追加する TextEditChangeGroup とでは，変換対象の IMethod を取得する処理が異なる．メソッドを削除する場合，変換する対象である IMethod は，変換前の ICompilationUnit にしか存在しない．メソッドを追加する場合，変換対象である IMethod は，変換後の ICompilationUnit にしか存在しない．TextEditChangeGroup は，変換前のソースコードに対して，どのように変換をするかという情報を扱っているため，TextEditChangeGroup から取得できる TextEdit は変換前のソースコードに対するオフセット値しか保持していない．よって，削除する対象である IMethod は問題なく取得できる．

しかし，追加する対象である IMethod を取得する場合は，変換後の ICompilationUnit における対象メソッドのオフセット値を計算する必要がある．取得する対象であるメソッドと同じソースコード内で，そのメソッドより上で適用された変換の差分を計算し，合計して，メソッド開始位置のオフセット値を計算する．このようにすることで，変換前の情報から変換後の ICompilationUnit 内の IMethod を取得する．

テストが失敗した原因であるコード変換の絞込

前述の処理で，TextEditChangeGroup と atomic changes とを対応付けることができたので，その対応付けを利用する．

Chianti の変更波及解析では，各 Affected Test から Affecting Changes を取得することができる．そして，Affecting Changes から atomic changes を取得することもできる．したがって，各 Affected Test から対応する Affecting Changes である atomic changes を取得し，その atomic changes と TextEditChangeGroup との対応付けを利用すれば，Affected Tests と TextEditChangeGroup を対応付けることができる．Affected Tests のうち，実行結果が失敗であるテストケースに対応する TextEditChangeGroup を取得することで，テストが失敗した原因であるコード変換を絞り込むことができる．

5　ケーススタディ

以下に示す Java 言語のソースコードを例として作成し，実装したツールを適用した．すべてのクラスは同じパッケージに定義されているので，パッケージ宣言は省略している．

Figure インタフェース

```
public interface Figure {

    abstract public int calculateArea();

}
```

Parallelogram クラス

```
public abstract class Parallelogram implements Figure {

    protected int base;

    protected int height;

    public Parallelogram(int base, int height) {
        if (base <= 0 || height <= 0)
            throw new IllegalArgumentException();

        this.base = base;
        this.height = height;
    }

}
```

DiamondShape クラス

```
public class DiamondShape extends Parallelogram {

    public DiamondShape(int base, int height) {
```

```
        super(base, height);

        if (base < height)
            throw new IllegalArgumentException();
    }

    public int calculateArea() {
        return this.base * this.height;
    }

}
```

Rectangle クラス

```
public class Rectangle extends Parallelogram {

    public Rectangle(int base, int height) {
        super(base, height);
    }

    public int calculateArea() {
        return this.base * 4;
    }

}
```

Square クラス

```
public class Square extends Rectangle {

    public Square(int base, int height) {
        super(base, height);

        if (base != height)
            throw new IllegalArgumentException();
```

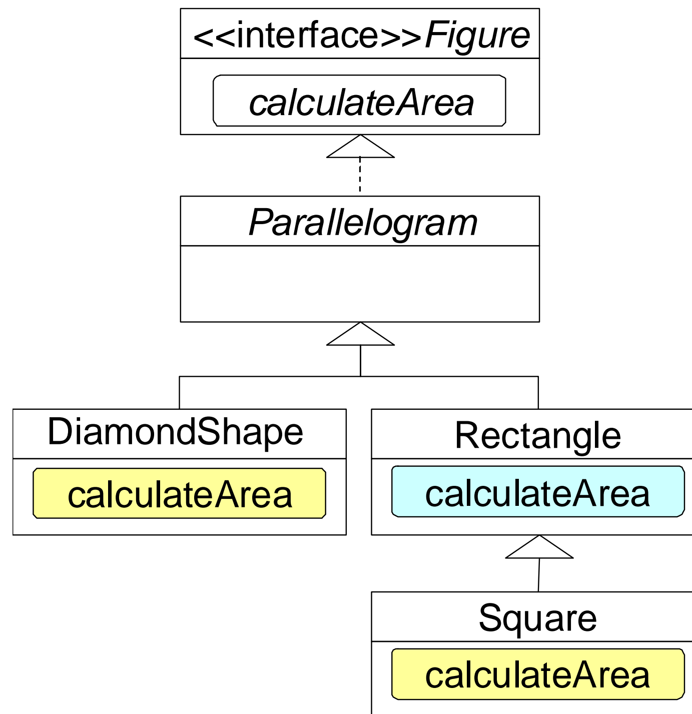


図 8: ケーススタディで扱うソースコードの構造

```
    }

    public int calculateArea() {
        return this.base * this.height;
    }

}
```

このソースコードの構造を図 8 に示す.

以下は JUnit テストフレームワークを使用した自己テストコードである.

DiamondShapeTest クラス

```
import junit.framework.TestCase;

public class DiamondShapeTest extends TestCase {
```

```
    public void testCreateDiamondShapeWithIllegalArgument() {
        try {
            Figure figure = new DiamondShape(3, 4);
        } catch (Exception e) {
            assertTrue(true);
            return;
        }
        fail();
    }

    public void testCalculateArea() {
        Figure figure = new DiamondShape(3, 2);
        assertEquals(6, figure.calculateArea());
    }
}
```

SquareTest クラス

```
import junit.framework.TestCase;

public class SquareTest extends TestCase {

    public void testCreateSquareWithIllegalArgument() {
        try {
            Figure figure = new Square(3, 2);
        } catch (Exception e) {
            assertTrue(true);
            return;
        }
        fail();
    }

    public void testCalculateArea() {
        Figure figure = new Square(3, 3);
        assertEquals(9, figure.calculateArea());
```

```
        }
    }
```

このソースコードに対して，以下のメソッドを Parallelogram クラスに引き上げる．

- DiamondShape クラスの calculateArea メソッド
- Square クラスの calculateArea メソッド

本研究で実装したツールを適用し，メソッドの引き上げを行った．自己テストコードのクラスに定義されている 4 つのテストメソッドはすべて入力した．

結果，SquareTest クラスの testCalculateArea メソッドを実行した結果が失敗であり，Square クラスからメンバ宣言を削除するコード変換が，その失敗の原因である可能性があると絞り込めた．

このケーススタディでは，Square クラスから，その親クラス Rectangle に定義されている calculateArea メソッドを無視するようにメソッドの引き上げを行った．JDT のリファクタリング支援機能では，この例に対して警告やエラーは表示しなかった．

しかし，実際には，Square クラスから calculateArea メソッドを削除すると，オーバーライド関係がなくなり，Rectangle クラスに定義している calculateArea メソッドが Square クラスに継承されるようになる．したがって Dynamic Dispatch が変化し，Square クラスに対して calculateArea メソッドを呼び出すと，Rectangle クラスに定義している calculateArea メソッドが呼び出されるように振る舞いが変化している．その結果，異なる処理内容であるメソッドが呼び出され，SquareTest クラスの testCalculateArea メソッドを実行した結果が失敗になった．

この振る舞いの変化の原因であるメンバの削除が，実装したツールでテストが失敗した原因である可能性として提示されることが確認できた．

6 関連研究

6.1 JUnit/CIA

Stoerzer らは，変更波及解析が算出した Affecting Changes を，欠陥を作り込んだ確からしさに応じて分類するツール JUnit/CIA を提案している [16]．欠陥を作り込んだ確からしさは，各 Affecting Change に対応するテストメソッドを実行した結果の変化から算出している．スーパークラスの抽出 (Extract Superclass)[4] を行うリファクタリングなど，規模の大きいリファクタリングを支援する場合，JUnit/CIA のように，Affecting Changes を欠陥を作り込んだ確からしさに基づいて分類し提示することで，リファクタリング時に欠陥を作り込んだ開発者が欠陥を特定することを支援できると考えられる．

6.2 動的な影響波及解析 (Dynamic Impact Analysis)

これまでに，回帰テストやデバッグなどの保守作業を支援することを目的として，動的に影響波及解析を行う手法である CoverageImapct[12] や PathImapact[9] が提案されている．これらは，ソースコード中のあるメソッドが修正されたときに，振る舞いが変化する可能性があるメソッドの集合を特定することができる．

本研究で提案するツールは，リファクタリング中に行う回帰テストを支援するために Ryder らの変更波及解析を用いている．CoverageImpact や PathImpact は振る舞いが変化する可能性があるメソッドを特定することに対し，Ryder らの変更波及解析は，振る舞いが変化した可能性があるテストメソッド (Affected Tests) や，テストメソッドの振る舞いを変化させた可能性がある変更 (Affecting Changes) を特定する．リファクタリングを行う開発者は，JUnit テストフレームワークに基づくテストメソッドを定義し，実行することで，外部的振る舞いが変化していないことを確認するので，テストメソッドの振る舞いの変化に着目した Ryder らの変更波及解析の方が，メソッドの振る舞いの変化に着目した CoverageImapct や PathImapact と比べて，リファクタリング支援に適していると考えられる．

6.3 プログラムスライシング

プログラムスライシング [17, 21] を用いて，テストが失敗する原因となったコード片の特定を行う手法 [1, 8] の研究が行われている．

提案ツールは，変更波及解析に基づいているため，リファクタリングを目的とした変更前後のソースコードから差分を解析し，テストメソッドが失敗した原因になった可能性がある変更（メソッドの追加・削除など）を特定している．

プログラムスライシングを用いる方法は，ソースコードの差分を分析することはないた

め，変更されていない部分についても，プログラムスライスに含まれるコード片であれば，テストメソッドが失敗する原因として提示する可能性がある．一方で，プログラムスライスに基づく手法は，文単位でコード片を提示することが可能であるため，メソッド単位の変更を特定する提案ツールと比べて，細粒度の提示が可能である．提案ツールとプログラムスライスに基づく方法を組み合わせることで，テストメソッドが失敗した原因となった可能性がある文単位の変更を提示できると考えられる．

6.4 リファクタリング時におけるテストケースの変更

本稿では，テストメソッドが正しく，かつ，リファクタリング前後で変更されないことを前提に議論を行ってきた．しかし，実際の開発では，リファクタリングはプログラムの構造を大きく変化させることがあるため，それに伴ってテストメソッドも修正する必要が生じる場合が考えられる [13, 20]．今後，このような場合に対する提案ツールの有効性や支援方法の改善を検討する必要がある．

6.5 自動リファクタリングにおける条件判定

本研究では，自動リファクタリングを支援するために，コード変換を適用した後，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行うテスト作業を支援している．

同時に，リファクタリングを目的としたコード変換を自動で行うため，外部的振る舞いを保つために，グラフ表現を利用して，変換前のソースコードが満たすべき条件を考案する研究もある [6, 10, 18, 19]．

本稿で使用した変更波及解析の Affected Tests は，変更・追加・削除されたメソッドを呼び出しているかどうかで定義されている．そのため，メソッド内部のみに対してリファクタリングを目的としたコード変換が適用されるとき，そのメソッドを呼び出すテストメソッドが存在していても，条件分岐が原因でその変換された部分をテストしていない場合を判別することができない．したがって，本研究で提案しているツールを用いても外部的振る舞いが保たれていることを完全に確認することはできない．

よって，変換前のソースコードが満たすべき条件を判定する手法と組み合わせることで，より確実に，リファクタリングを目的としたコード変換で外部的振る舞いが変化することを防止できると考えられる．

7　むすび

本研究では，リファクタリング支援に変更波及解析を利用する際の問題点を説明し，その問題点を解消するリファクタリング支援ツールを提案した．提案したツールの一部を実装したツールを使用して，ケーススタディを行い，既存のリファクタリング支援機能では検出できなかった，外部的振る舞いが変化する可能性があるコード変換を検出することができた．

今後の課題は，テストケースが失敗した場合に，その原因を絞り込む機能とテストケースでテストされていないコード変換を特定する機能を，ユーザインタフェースを含めて実装し，人を対象にした実験を行って，ツールの有用性を評価することが挙げられる．作業時間の短縮と，バグを混入しにくくなったかという点で有用性を評価することを考えている．加えて，JDT のリファクタリング支援機能におけるコード変換記述と，Chianti の変更波及解析における atomic changes を対応付けることが，メソッドの引き上げ以外のリファクタリングパターンにも有効か調査することを考えている．

また，今回のケーススタディで使用した例で，JDT のリファクタリング支援機能は警告やエラーを出力しなかった．この点に着目し，自動リファクタリングにおける条件判定を調査して，自動リファクタリングを使用する開発者に，条件判定で外部的振る舞いを保証している範囲を提示する，という別の側面からリファクタリング支援機能を改善することを考えている．

謝辞

本研究の全過程を通して，常に適切な御指導，御助言を賜りました大阪大学大学院情報科学研究科 コンピュータサイエンス専攻井上 克郎 教授に心より深く感謝いたします.

本論文を作成するにあたり，常に適切な御指導，御助言を頂きました大阪大学大学院情報科学研究科 コンピュータサイエンス専攻松下 誠 准教授に心から感謝いたします.

本論文を作成するにあたり，適切な御助言を頂きました大阪大学大学院情報科学研究科 コンピュータサイエンス専攻石尾 隆 助教に心から感謝いたします.

本研究を通して，様々な御協力を頂きました大阪大学大学院情報科学研究科 コンピュータサイエンス専攻吉田 則裕 氏に深く感謝致します.

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 教授に深く感謝いたします.

最後に，その他様々な御指導，御助言等を頂いた大阪大学大学院情報科学研究科 コンピュータサイエンス専攻井上研究室の皆様に深く感謝致します.

参考文献

[1] R. A. DeMillo, H. Pan, and E. H. Spafford. Critical slicing for software fault localization. In *Proc. of the 1996 ACM SIGSOFT international symposium on Software testing and analysis(ISSTA 1996)*, pp. 121–134, San Diego, California, USA, 1996.

[2] Eclipse. `http://www.eclipse.org`.

[3] Eclipse Java development tools. `http://www.eclipse.org/jdt/`.

[4] M. Fowler, K. Beck, J. Brant, W. F. Opdyke, and D. Roberts. *Refactoring: improving the design of existing code*. Addison Wesley, 1999.

[5] JUnit. `http://www.junit.org`.

[6] H. Kazato, M. Takaishi, T. Kobayashi, and M. Saeki. Formalizing refactoring by using graph transformation. In *IEICE TRANSACTIONS on Information and Systems Vol.E87-D No.4*, pp. 855–867, 2004.

[7] J. Kerievsky. *Refactoring to Patterns*. Addison Wesley, 2004.

[8] S. Kusumoto, A. Nishimatsu, K. Nishie, and K. Inoue. Experimental evaluation of program slicing for fault localization. *Empirical Software Engineering*, 7(1):49–76, 2002.

[9] J. Law and G. Rothrmel. Whole program path-based dynamic impact analysis. In *Proc. of the 25th International Conference on Software Engineering(ICSE 2003)*, pp. 308–318, Portland, Oregon, USA, 2003.

[10] T. Mens, N. V. Eetvelde, S. Demeyer, and D. Janssens. Formalizing refactorinsg with graph transformations. In *Software Maintenance and Evolution: Research and Practice 17*, pp. 247–276, 2005.

[11] W. F. Opdyke. *Refactoring object-oriented frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[12] A. Orso, T. Apiwattanapong, and M. J. Harrold. Leveraging field data for impact analysis and regression testing. In *Proc. of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering(ESEC/FSE 2003)*, pp. 128–137, Helsinki, Finland, 2003.

[13] J. U. Pipka. Refactoring in a “test first”-world. In *Proc. of the 3rd International Conference on eXtreme Programming and Agile Processes in Software Engineering (XP2002)*, pp. 178–181, 2002.

[14] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: A tool for change impact analysis of Java programs. In *Proc. of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2004)*, pp. 432–448, Vancouver, BC, Canada, 2004.

[15] B. G. Ryder and F. Tip. Change impact analysis for object-oriented programs. In *Proc. of the 3rd ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering (PASTE 2001)*, pp. 46–53, Snowbird, UT, USA, 2001.

[16] M. Stoerzer, B. G. Ryder, X. Ren, and F. Tip. Finding failure-inducing changes in Java programs using change classification. In *Proc. of the 14th ACM SIGSOFT international symposium on Foundations of software engineering(FSE 2006)*, pp. 57–68, Portland, Oregon, USA, 2006.

[17] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.

[18] F. Tip. Refactoring using type constraints. In *Static Analysis, 14th International Symposium, number 4634 in Lecture Notes in Computer Science*, pp. 1–17, Kongens, Lyngby, Denmark, 2007.

[19] F. Tip, A. Kiezun, and D. Bäumer. Refactoring for generalization using type constraints. In *Proc. of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications (OOPSLA 2003)*, pp. 13–26, Anaheim, California, USA, 2003.

[20] A. van Deursen and L. Moonen. The video store revisited - thoughts on refactoring and testing. In *Proc. of the 3rd International Conference on eXtreme Programming and Agile Processes in Software Engineering (XP2002)*, pp. 71–76, 2002.

[21] M. D. Weiser. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigan, 1979.