Vol.2021-SE-207 No.19
2021/3/1

リファクタリングがテストコードに与える影響の定量的調査

清水 一輝[1,a)]　柏 祐太郎[1,b)]　亀井 靖高[1,c)]　鵜林 尚靖[1,d)]

概要：リファクタリングはソフトウェア品質を向上させるために広く実施されている．しかし，リファクタリングを行うことは多大な時間と労力を要することが多く，開発期間が短く限られているプロジェクトでは敬遠される傾向にある．開発者にリファクタリングを躊躇させる心理的要因の一つとして「リファクタリングによるテストスイートの破壊」が挙げられる．一般的に，リファクタリングは外部から見た挙動を変更しない．しかし，ソースコードの内部構造を変化させるため，メソッドレベルでの挙動が変わり，従来のテストメソッドがコンパイルエラーに陥ることや予期していた出力と異なる出力となり失敗する可能性がある．本研究では，どのようなリファクタリングがテストを破壊しやすく，それぞれがどの程度の変更（修正テストメソッド数および修正行数）を必要とするについてを定量的な調査を行い，以下の 4 点が分かった．(1) 最もコンパイルエラーに陥りやすいリファクタリングは Rename Class であり，34.5% のテストメソッドがコンパイルエラーに陥る．(2) 最もテストメソッドが修正されるリファクタリングは Rename Class であり，10.3%のテストメソッドが変更される．(3) Rename Class では 17.1%の行数が変更される．(4) Change Attribute Type は 28.6%の確率でテストメソッドが追加される．

キーワード：リファクタリング，テスト，マイニングソフトウェアリポジトリ

1. はじめに

リファクタリングはプロジェクトの生産性の向上やソースコードの可読性の向上，バグの混入を防ぐことを目的として広く実施されている [2][5]．その一方で，リファクタリングは開発者に嫌厭される傾向にある [4][13]．その理由の 1 つにテストコストの増加がある [4]．Vassallo ら [13] のリファクタリングの障壁に関する調査によると，開発者がリファクタリングを躊躇する心理的な要因の 1 つとして「リファクタリングによるテストスイートの破壊」が挙げられると報告している．

定義上リファクタリングは，ソフトウェアの入力や出力といった，外部から見た挙動を変更しない．しかし，ソースコード内部の構造を変更するため，従来のテストメソッドがコンパイルエラーに陥り実行不可能となることや，予期していた出力と異なる出力となりテストが失敗する可能性がある．実際に壊れたテストメソッドの半数がリファクタリングの影響を受けていたことが報告されている [6]．

近年のソフトウェア開発ではテストコードが大きくなる傾向にあり [12]，壊れてしまったテストメソッドを修正する作業は多大な時間と労力が必要となる．特に競争が激しいソフトウェア業界では，多くのプロジェクトが短い期間でリリースを行う傾向にあり [1]，テストスイートの修正に開発工数を割く余裕がないと考えられる．

リファクタリングを行なった際のテストコードへの影響について多くの研究が行われている [3][10][13]．リファクタリングの影響を受け，壊れたテストスイートはソフトウェアの品質保証のため，修正する必要がある．しかし，リファクタリングによってテストスイートがどの程度修正を必要とするのかということに着目した調査は行われていない．修正が必要なテストメソッドの数や修正が必要なテストコードの行数について調査を行うことで，リファクタリングを行うための時間的，人的コストを見積もることができると考えられる．例えば，多くのテストメソッドがリファクタリングによって壊れるということがわかっていた場合でも，少量の変更でテストメソッドを修正できることがわかっているならば，リファクタリングへの躊躇が緩和されると考えられる．

これらの背景から本論文では以下の 4 つの調査課題（以下 RQ：Research Question）について調査を行う．

RQ1　どのようなリファクタリングがテストメソッドを破壊しやすいか

[1] 九州大学
Kyushu University
[a)] shimizu@posl.ait.kyushu-u.ac.jp
[b)] kashiwa@ait.kyushu-u.ac.jp
[c)] kamei@ait.kyushu-u.ac.jp
[d)] ubayashi@ait.kyushu-u.ac.jp

RQ2 どのようなリファクタリングが多くの修正箇所（テストメソッド）を必要とするか

RQ3 どのようなリファクタリングが多くの修正行数を必要とするか

RQ4 どのようなリファクタリングがテストメソッドを追加で必要とするか

上記の RQ を達成することによって開発者にリファクタリングの種類毎にどの程度テストメソッドが壊れるか，どの程度テストメソッドに修正が必要か，どの程度テストメソッドの追加が必要かということを開発者が理解する助けとなると考えられる．

章構成：2 章では本論文の背景と関連研究について説明し，3 章で実験手法について説明する．4 章では本論文で用いたデータ収集方法について説明する．5 章では RQ を分析する手法とその結果について述べ，最後に 6 章で本論文の結論と展望を述べる．

2. 関連研究と動機

Rachatasumrit ら [6] はリファクタリングを伴って壊れるテストメソッドの数について調査した．Rachatasumrit らは 18,000 個以上のテストケースを実行し，失敗した 80 個のテストケースのうち 39 個がリファクタリングと判定される変更を含んでいたことを報告している．しかし，Rachatasumrit らの調査はいくつか問題点がある．

エラーの種類が区別されていない．プロダクトコードの変更で壊れるテストメソッドにはコンパイルエラー，実行時エラー，失敗といった 3 種のエラーが考えられる．例えば Rename Class や Add Parameter などのリファクタリングによって，プロダクトメソッドのシグネチャが変更されたとき，リファクタリングされたプロダクトメソッドを用いているテストメソッドはコンパイルエラーに陥ってしまう．他にも Change Return Type によってプロダクトメソッドの返り値が変更されたとき (int 型から double 型への変更など)，コンパイラはエラーを検出できないが，テストメソッドは予期していた値と違う値を受け取ってしまうため（int 型では 1 だが double 型では 1.0 など）テストが失敗してしまう．Rachatasumrit らはエラーの種類を区別した調査は行なっていない．エラーの種類によっては修正に必要なコストが異なると考えられる．具体的には，失敗したテストメソッドの修正は問題点を発見することの難しさから，コンパイルエラーの修正よりも時間がかかると考えられる．そのためエラーの種類を区別した調査を行う必要がある．

実際の開発を想定していない．リファクタリングを行う際，開発者はプロダクトコードを変更し，その後テストコードを修正すると考えられる．テスト駆動開発においても，リファクタリングの際は同様の手順でテストコードの修正を行うと考えられる．しかし，Rachatasumrit らはリファクタリングが行われたプロダクトコードと，リファクタリングにしたがって修正された後のテストコードを用いて調査を行なっている．これは実際の開発環境を想定しておらず，テストが本当にリファクタリングによって壊れたかどうか疑問が残る．対して，Xinye ら [9] によるバグ修正のテストへの影響についての調査では，修正前のテストコードを対象とした調査を行なっている．そこで本論文ではリファクタリングされたプロダクトコードと，修正前のテストコードを用いて，まず以下の調査を行う．

> RQ1：どのようなリファクタリングがテストメソッドを破壊しやすいか

プロダクトの品質保証のため，壊れたテストメソッドは修正を行う必要がある．修正が必要なテストメソッドの数によって，リファクタリングに必要なコストが変化すると考えられる．そこで以下の調査を行う．

> RQ2：どのようなリファクタリングが多くの修正箇所（テストメソッド）を必要とするか

テストメソッド数のみに着目している．テストを修正する際に必要な情報は修正が必要なテストメソッドの数だけではない．メソッドやクラスのシグネチャを変更するリファクタリングが行われた際は，他クラスから呼び出される数に応じて多数のメソッドが影響を受ける事になる．しかし，多数のメソッドが影響を受けていた場合でも，1 行の変更で十分な場合はリファクタリングへの躊躇が緩和されると考えられる．

図 1 はリファクタリングの影響を受けたテストメソッドの変更行数を表しており，変更行数は 1 行から 30 行以上の範囲があることがわかる．Elish ら [3] はリファクタリングによるテストの修正作業について調査を行った．しかし，Elish らは実際のテストコードを使って調査を行ったわけではなく，プロダクトコードの複雑度などからテストの修正にかかるコストを調査している．リファクタリングによる修正が必要なテスト行数に着目した調査は未だ行われていない．そこで以下の RQ について調査を行う．

> RQ3：どのようなリファクタリングが多くの修正行数を必要とするか

修正だけではテストが十分でない可能性がある．Aminata ら [7] によるアンチパターンがテストに与える影響についての調査によると，プログラムにアンチパターンがある場合，多くのテストが必要であると報告している．リファクタリングを行なった際も同様に，リファクタリングの種類によってはテストメソッドを追加する必要があると

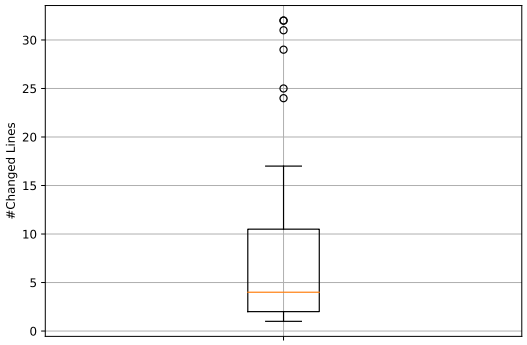

図 1 テストメソッドに対する変更行数

考えられる．具体的には，Add Parameter によってメソッドの引数が追加された場合，新たな引数の挙動を確認するため新たなテストメソッドが必要となる．そのためテストメソッドの変更に加えて，テストメソッドの追加が必要となる可能性が考えられる．そこで最後の RQ として以下の調査を行う．

> どのようなリファクタリングがテストメソッドを追加で必要とするか

3. リファクタリングによる影響の分析手法

本実験ではリファクタリングによって影響を受けたテストメソッドを明らかにする必要がある．テストメソッドはプロダクトコードを出来るだけ網羅するように設計されているが，リファクタリングはプロダクトコードの一部分が変更される．そのためテストメソッドがリファクタリングの行われた部分を実行するか解析する必要がある．本実験では動的解析を使い，プロダクトコードで呼び出される行を解析する．その結果，テストメソッドの何行目がどのプロダクトコードを呼び出しているのかを解析することが可能となる

本実験は以下の 3 つの段階で表すことができる．以下では各段階の詳しい実験手法について述べる．

3.1 リファクタリングの特定

本実験ではまず各コミットのプロダクトコードで起きたリファクタリングの種類とリファクタリングが行われた位置を特定する．リファクタリングを特定するツールとして Tsantalis ら [11] が開発した Refactoring Miner を用いる．このツールでは 55 種のリファクタリングを特定することができ，その精度は高い適合率と高い再現率であることが報告されている [11].

実験対象のプロジェクトを GitHub からクローンした後，全てのコミットに対して Refactoring Miner を実行し，リファクタリングの種類と位置を特定する．

3.2 動的解析

リファクタリングによってテストメソッドが影響を受けているかどうかを明らかにするため，テストメソッドがプロダクトコードのどの行を呼び出したかという情報が必要である．本実験では動的解析を用いた．呼び出し関係の解析にはコールグラフもよく用いられるが，テストメソッドが実行したプロダクトコードで，リファクタリングが行われたかという行単位の細かい解析を行うことができない．テストはプロダクトコードの網羅率が高くなるよう設計されるので，コールグラフを用いた解析は本研究において適切でないと考えられる．テストメソッドの何行目がプロダクトコードのどの部分を実行しているかという解析を行うため動的解析を用いた．

動的解析では各テストメソッドが実行したプロダクトコードの順序を解析することができる．この順序の解析結果からテストメソッドの各行でどのプロダクトコードの行が実行されたかという分析を行う．動的解析は使用する際，時間やメモリなどのコストが高いことが知られている．本実験ではコストの問題を緩和するため，SElogger [8] というツールを用いた．SElogger では大部分のプロダクトコードの解析を行える上に，for ループなどの冗長な解析を省略することができる．SElogger を用いるため，実験対象プロジェクトのビルドファイルを変更し Java エージェントを利用する．具体的には，Maven を利用することで maven-surefire というプラグインから SElogger を実行することができる．ビルドファイルを変更し，SElogger を全てのコミットに対して実行する．コンパイルエラーが起きた場合，コンパイルエラーを起こしたメソッドを除外し，テストが終了するかテストがなくなるまで再実験を行う．

3.3 影響分析

図 2 は本実験の影響分析手法の概要を示している．動的解析により，テストメソッドが実行したプロダクトコードを分析できる．テストメソッドが実行したプロダクトコードでリファクタリングが行われた際，そのテストメソッドはリファクタリングの影響を受けたと考えられる．

Refactoring Miner によって，あるコミット X とその前のコミット X-1 の間で起きたリファクタリング種類と位置が分かる．コミット X とコミット X-1 の 2 つのコミットに対して影響分析を行い，テストメソッド各行がリファクタリングの影響を受けたかどうかを解析する．

コミット X とコミット X-1 の 2 つのコミットで解析を行う理由は片方のコミットにのみ現れるリファクタリングがあるためである．具体的には，Rename Method はメソッドの名称の変更であり，コミットの前後でリファクタリングが検出できる．しかし，Extract Method はメソッドの抽出であり，新たなメソッドを追加するためコミット X でのみリファクタリングを検出できる．最終的

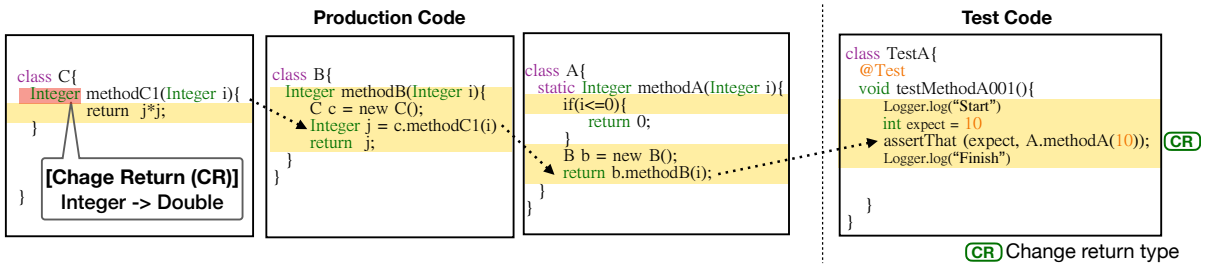

図 2 影響分析手法の概要

表 1 本実験の対象プロジェクト

プロジェクト	概要	統計量					
		コミット数	リファクタリングコミット数	リファクタリングされたインスタンス数	ソースコード行数 (プロダクト)	ソースコード行数 (テスト)	テストメソッド数
COMMONS-IO	Apache commons ライブラリ	2,740	444	4,064	13,736	24,714	1,310
SPRING	SQL マッピングフレームワーク	1,615	155	782	1,924	3,933	120
JODA-BEANS	コード生成フレームワーク	840	279	3,515	17,068	34,185	444
JSOUP	Java 用 HTML ライブラリ	1,386	289	1,358	12,570	9,201	725
SPARK	Java 用 Web フレームワーク	1,062	309	1,789	6,253	5,045	320
LITTLEPROXY	HTTP プロキシ	1,003	252	1,616	4,180	4,665	55
RXJAVA-JDBC	データベース呼び出しライブラリ	850	186	849	4,611	3,330	74
SPOON	ソースコード分析ライブラリ	3,278	878	9,564	64,038	44,960	1,938

にクラス名，メソッド名，引数からなるシグネチャによって，コミット X とコミット X-1 のテストメソッドを結びつける．コミット X とコミット X-1 の両方で検出されたリファクタリングについては，同じリファクタリングの重複を防ぐため除外する．

4. データの収集

4.1 プロジェクトの選定

本実験では以下の条件を設定し，GitHub から 8 プロジェクトを選定した．選定したプロジェクトを表 1 に示す．

使用言語が Java である． 本実験ではリファクタリングを検出するため Refactoring Miner を用いている．Refactoring Miner は Java で書かれていることを前提としていため，本実験では Java プロジェクトを用いる．

Maven を利用している． 本実験では各コミットのテストを実行する必要があるため，自動でプロジェクトのコンパイルとテストの実行を行える Maven を利用している．Maven は有名なビルドツールの 1 つであり，Java のプロジェクトで広く採用されている．

JUnit を利用している． 本実験ではテストへの影響を調査するため，テストコードを分析する必要がある．そこで JUnit4 もしくは JUnit5 を用いているプロジェクトを対象とした．JUnit は Java プロジェクトでは標準的なテストフレームワークである．

モジュール構造ではない． プロジェクトの中には開発を容易にするためモジュール開発を行なっているものがある．モジュール開発では各モジュールがプロダクトコードとテストコードをもつが，モジュール同士で依存関係にあることがある．壊れたモジュールが存在する場合，依存元のコンパイルが通らないため，本実験ではモジュール開発ではないプロジェクトを対象としている．

4.2 ビルドファイルの設定

Maven でテストを実行するためには sure-fire というプラグインが必要となる．Maven を利用しているほとんどのプロジェクトは sure-fire を使っているが，中には sure-fire を使わずローカルでテストを行なっているプロジェクトもある．そこで sure-fire がない場合，プロジェクトのビルドファイルへ sure-fire を導入し Meven によるテストを可能としている．

通常 Maven でテストを行う際，一度テストメソッドが失敗するとその後のテストメソッドが実行されない．本実験では全てのテストメソッドを実行したいため，失敗したテストメソッドを無視する設定を導入している．

4.3 テストメソッドの特定

Maven によってテストを行うため，プロダクトコードとテストコードを区別する必要がある．また，RQ1 の調査のため本実験ではコミット X のプロダクトコードを，その前のコミット X-1 のテストメソッドでテストを行う．テストコードかの判別には，Maven の設定ファイルである pom.xml を参照する．さらにテストメソッドであることを判別するため，@Test のアノテーションがついたテストコード内のメソッドをテストメソッドとしている．

4.4 テストメソッドのフィルタリング

本実験では以下の条件のいずれかを満たすテストメソッドを除外している．

表 2　RQ1 実験結果

リファクタリングタイプ	#成功	#コンパイルエラー	#実行時エラー	#失敗	%成功	%コンパイルエラー	%実行エラー	%失敗
Extract Variable	1,076	0	0	0	100	0.00	0.00	0.00
Rename Variable	477	0	0	0	100	0.00	0.0	0.00
Replace Variable With Attribute	444	0	0	0	100	0.00	0.00	0.00
Rename Method	336	5	0	0	98.5	1.47	0.00	0.00
Extract Method	287	0	1	2	99.0	0.00	0.34	0.69
Add Parameter	135	11	0	2	91.2	7.43	0.00	1.35
Add Class Annotation	141	0	0	0	100	0.00	0.00	0.00
Move Class	110	12	9	0	84.0	9.16	6.87	0.00
Change Return Type	121	7	0	1	93.8	5.43	0.00	0.78
Extract And Move Method	101	0	0	3	97.1	0.00	0.00	2.88
Extract Attribute	99	0	1	0	99.0	0.00	1.00	0.00
Add Method Annotation	79	0	0	0	100	0.00	0.00	0.00
Merge Attribute	69	0	0	0	100	0.00	0.00	0.00
Change Attribute Type	57	0	5	0	91.9	0.00	8.06	0.00
Rename Parameter	51	0	0	1	98.1	0.00	0.00	1.92
Change Variable Type	34	0	0	0	100	0.00	0.00	0.00
Rename Class	19	10	0	0	65.5	34.5	0.00	0.00
All	3,636	45	16	9	98.1	1.21	0.43	0.24

マージコミット．マージコミットでは同じ変更がそのコミット以前で行われていることとなり，マージコミットを含めるとリファクタリングが重複してしまう可能性があるため，マージコミットは分析対象外としている．

1 つ前のコミットで失敗している．リファクタリングによって壊れたテストメソッドであることを保証するため，1 つ前のコミットですでに壊れているテストメソッドを除外している．

複数のリファクタリングの影響を受けている．複数のリファクタリングの影響を受けているテストメソッドがある場合，必ず全てのリファクタリングがテストの破壊やテストの修正に影響を与えているとは限らない．本実験ではそのような複数のリファクタリングから，実際にテスト破壊やテスト修正に影響を与えたリファクタリングを区別することができないため除外した．

4.5　実行環境

全てのテストの実行と影響分析を実行するためには 1 コミットあたり 3 時間ほどの時間が必要となる．そのため本実験では Kubernetes を用いて，分散環境を構築し，全実験をおよそ 2 週間ほどで実行できるようにしている．

5.　調査課題

5.1　[RQ1] どのようなリファクタリングがテストメソッドを破壊しやすいか

開発者がリファクタリングを行う際，まずプロダクトコードを変更し，その後テストコードを変更する．定義上リファクタリングはプロダクトの外部への挙動を変更しないため，開発者はテストコードに変更を加えない可能性がある．しかし，リファクタリングはプロダクトコードのメソッドなど内部の挙動を変更するため，テストメソッドの中には壊れるものがある．そこで RQ1 ではリファクタリングによるテストメソッドの破壊について調査する．

調査手法．プロダクトコードへリファクタリングが行われた際のテストコードへの影響を調査するため，開発者がプロダクトコードへリファクタリングを行い，テストは変更しない状況を作る必要がある．そこで RQ1 ではまずコミット X からテストディレクトリを削除し，その前のコミットであるコミット X-1 のテストディレクトリをコピーする．これにより，コミット X のプロダクトコードをコミット X-1 のテストコードでテストすることができる．このような前処理を行った後，テストの実行，影響分析を行い，壊れたテストメソッドを集計する．

結果と考察．表 2 は成功，コンパイルエラー，実行時エラー，失敗したテストメソッドの数を各リファクタリングで集計し，テストが壊れる可能性を表している．なおテストメソッド数が全テストメソッド数の 1%未満であるリファクタリングは除外している．

成功したテストメソッドの割合が 100%であるものが多いことから，多くのリファクタリングはテスト結果に影響を与えないことがわかる．特に Extract Variable や Replace Variable With Attribute，Add Class Annotation は影響を受けたテストメソッド数は 100 以上あるが，全て成功していることがわかる．これはリファクタリングによる影響がプロダクトコード内で吸収され，テストコードへの影響が及ばなかったためと考えられる．

その一方，いくつかのリファクタリングはテストメソッドを破壊することが観察された．最もテストメソッドを破

壊するリファクタリングは Rename Class であり，19 個のテストメソッドが成功したのに対して，コンパイルエラーとなったテストメソッドが 10 個であった．成功したテストメソッドと壊れたテストメソッドが存在するのは，開発者がプロダクトコード内で影響を吸収できるようなラッパークラスを作成したために，影響がテストメソッドまで波及したものとそうでないものがあるためであると考えられる．コンパイルエラーが起きるリファクタリングには他にも Move Class や Add Parameter があるが，これらのリファクタリングの特徴として，メソッドのシグネチャに影響があるリファクタリングであることが挙げられる．

対して，Extract Method や Change Attribute Type ではコンパイラが検出できない実行時エラーやテストの失敗が起きていることがわかる．これらのリファクタリングはメソッドのシグネチャに影響を与えないが，メソッド内の挙動や返り値が変更される特徴があり，その結果予期せぬ出力となるということが起きているためだと考えられる．具体的には Extract Method によって，あるメソッド内部の処理が新たな引数とともに抽出され，引数が追加されたために，返り値が予期していたものと違う値になることが確認された．

5.2 [RQ2] どのようなリファクタリングが多くの修正箇所（テストメソッド）を必要とするか

RQ1 では壊れたテストメソッドに着目したが，テストメソッドがリファクタリングによって壊れた状態で放置される可能性は少ない．開発者はリファクタリングを行なった際，プロダクトコードの変更に基づき，テストコードが壊れていないかを確認し，壊れている場合は修正するという作業を行う．Rachatasumrit ら [6] はリファクタリングに伴って変更されたテストメソッドが壊れている可能性の調査を行なったが，実際にリファクタリングによってテストメソッドがどのくらい修正されるかについては調査が行われていない．RQ2 ではリファクタリングによって修正が必要なテストメソッドについて調査を行う．

調査手法．テストメソッドが修正されたかどうかを解析するため，Git コマンドによって diff ファイルを抽出する．テストの実行，影響分析により，テストメソッドの各行がリファクタリングの影響を受けたかどうかを解析できる．リファクタリングの影響を受けたテストメソッドが 1 行以上修正されているものを，リファクタリングによって修正されたテストメソッドとみなす．

結果と考察．表 3 はリファクタリングの種類ごとにリファクタリングの影響を受けたテストメソッド数と修正されたテストメソッド数を集計し，テストメソッドが修正される確率を表している．なおテストメソッド数が全テストメソッド数の 1%未満であるリファクタリングは除外している．

表 3 RQ2 実験結果

リファクタリングタイプ	#テストメソッド総数	#修正されたテストメソッド数	%変更確率
Extract Variable	1,076	0	0.00
Rename Variable	477	0	0.00
Replace Variable With Attribute	444	0	0.00
Rename Method	341	6	1.76
Extract Method	290	3	1.03
Add Parameter	148	11	7.43
Add Class Annotation	141	0	0.00
Move Class	131	8	6.11
Change Return Type	129	9	6.98
Extract And Move Method	104	5	4.81
Extract Attribute	100	1	1.00
Add Method Annotation	79	0	0.00
Merge Attribute	69	0	0.00
Change Attribute Type	62	0	0.00
Rename Parameter	52	1	1.92
Change Variable Type	34	0	0.00
Rename Class	29	3	10.3
All	3,706	47	1.27

この結果から最もテストメソッドが修正されるリファクタリングは Rename Class であり，29 個のテストメソッド中 3 個が修正されていることがわかる．RQ1 で求めた最もテストメソッドを破壊するリファクタリングも Rename Class と同じであり，テストメソッドが破壊されるため修正が必要となったものと考えられる．他にもテストメソッドが変更されるリファクタリングは Move Class や Add Parameter がある．テストが破壊されても修正されないテストメソッドはいくつかある (実行時エラーとなったものなど) ものの，破壊されたテストメソッド数が多いほど，修正が必要なテストメソッド数も多いことがわかる．

5.3 [RQ3] どのようなリファクタリングが多くの修正行数を必要とするか

RQ2 では修正が必要なテストメソッド数を調査した．しかし，テストコードを修正する際に必要な情報は修正が必要なテストメソッドの数だけではない．具体的には，多くのテストメソッドに修正が必要だが，修正に必要な行数が各テストメソッドで 1 行ほどであった場合，修正に必要な時間的コストは低いと考えられる．そのような場合，開発者はリファクタリングを躊躇しない可能性がある．そこで RQ3 ではテストメソッド数ではなく行数に着目し，リファクタリングによって必要な修正行数を調査する．

調査手法．RQ2 の調査手法と同様の手順で RQ3 を分析する．まず，解析結果からリファクタリングの影響を受けたテストメソッドの行を特定する．次に diff ファイルから得た変更行の情報から，テストメソッドが変更された行を特定する．最後に，リファクタリングの影響を受け，かつ変更された行を集計する．

結果と考察．表 4 は各リファクタリング毎に影響を受けた行数と，修正された行数を集計し，影響を受けた行が変更される確率を表している．また図 3 は変更行数が 5 行以

表 4　RQ3 実験結果

リファクタリングタイプ	#総テスト行数	#テストメソッド行数の中央値	#リファクタリングの影響を受けた行数	#影響を受け修正された行数	%変更確率
Extract Variable	11,699	11.5	1,370	0	0.00
Rename Variable	6,219	10.5	744	0	0.00
Replace Variable With Attribute	5,008	9.0	585	0	0.00
Rename Method	5,663	12.0	967	11	1.14
Extract Method	11,490	12.0	1,930	4	0.21
Add Parameter	2,281	13.0	353	24	6.80
Add Class Annotation	2,626	12.0	147	0	0.00
Move Class	1,441	10.0	358	39	10.9
Change Return Type	4,676	18.0	220	20	9.09
Extract And Move Method	1,752	11.5	171	5	2.92
Extract Attribute	3,488	12.0	463	1	0.22
Add Method Annotation	2,787	13.0	173	0	0.00
Merge Attribute	484	6.0	69	0	0.00
Change Attribute Type	2,349	12.5	145	0	0.00
Rename Parameter	846	15.5	82	2	2.44
Change Variable Type	456	8.0	78	0	0.00
Rename Class	673	16.0	41	7	17.1
All	63,938	13.5	7,896	113	1.43

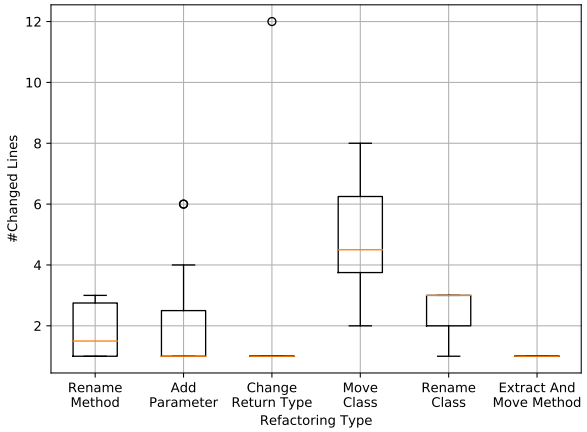

図 3　リファクタリング種別テストメソッドの修正行数

上のリファクタリングについて，テストメソッドごとの変更行数を箱ひげ図に表したものである．

この結果から，Move Class や Rename Class などクラスに対するリファクタリングを行う場合，1 テストメソッドあたりの変更行数が多くなることがわかる．RQ2 の結果も考慮に入れると，クラスに対するリファクタリングを行う場合，テストメソッドに修正が必要な可能性は 6% 以上あり，修正行数も 1 テストメソッドあたり平均 3 行ほどの修正が必要となるということが考えられる．クラスへのリファクタリングを行う際は，開発者はテストメソッドに対して，他のリファクタリングよりも修正に時間を費やす必要があるといえる．

また，Add Parameter や Change Return Type，Extract And Move Method はばらつきはあるものの 1 テストメソッドあたり平均 1 行の修正が必要となることがわかる．これらのリファクタリングはたとえ多くのテストメソッドが影響を受けて，壊れたとしても，修正行は 1 行ほどのためリファクタリングへの躊躇が緩和できると考えられる．

5.4　[RQ4] どのようなリファクタリングがテストメソッドを追加で必要とするか

リファクタリングはプロダクトコードに変更を加えるので，テストメソッドを追加で用意する必要がある可能性がある．具体的には，Change Attribute Type によってクラスのフィールドの型が変更され (int 型から Integer 型)，今まで受け取れなかった null 値を受け取ることができるようになった場合，実際にプロダクトが null 値を受け取った際の挙動を保証するため追加のテストメソッドが必要となると考えられる．そこで RQ4 ではテストメソッドが追加されるリファクタリングについて調査を行う．テストメソッド追加とは，あるコミット X-1 で存在していなかったテストメソッドがその次のコミット X で現れることとする．

調査手法．Git コマンドで diff ファイルを抽出することで，追加されたテストメソッドが判別できる．次に影響分析を行い，追加されたテストメソッドがリファクタリングの影響を受けているか分析する．RQ4 ではリファクタリングが起きた際に，テストメソッドが追加されるかどうかを分析するため，リファクタリングインスタンス（以下 RI：Refactoring Instance）単位で分析する．ここで RI とは Refactoring Miner が検出したリファクタリングであり，1 つの RI が複数のテストメソッドに影響を与える可能性

表 5 RQ4 実験結果

リファクタリングタイプ	#追加テストメソッド数	#テストメソッドの追加に影響した RI 数	#全 RI 数	%追加確率
Extract Method	1	1	20	5.00
Add Method Annotation	2	1	15	6.67
Add Parameter	1	1	15	6.67
Rename Method	2	2	12	16.7
Extract And Move Method	4	2	11	18.2
Change Attribute Type	4	2	7	28.6
Rename Variable	1	1	5	20.0

*RI: リファクタリングインスタンス

がある．

結果と考察．表 5 に追加テストメソッド数と追加されたテストメソッドに関係する RI 数，全 RI 数を集計し，テストメソッドが追加される確率を表したものである．

この結果からテストメソッドが追加されるリファクタリングには 7 種類あり，特に Change Attribute Type ではテストメソッドが追加される確率が最も高く 28.6%であることがわかる．これはフィールドの型が変更されることによって，受け取れる値，受け取れない値が変更され，その結果品質を保証するため新たなテストメソッドが必要となったためであると考えられる．また, Add Method Annotation や Extract And Move Method，Change Attribute Type では 1 つの RI で，テストメソッドが 2 つ以上追加されることがわかる．これらのリファクタリングが行われた際は，いくつか異なる入力を用意して品質を保証する必要があると考えられる．

6. まとめと今後の展望

本論文では，リファクタリングによるテストコードへの影響を分析するとともに，本実験で行なったリファクタリングの影響を分析する手法について説明した．

本論文の RQ への調査によって多くのリファクタリングはテストを壊すことがなく，テストの修正も必要としないということがわかった．しかし，クラスに対するリファクタリングはテストメソッドを壊しやすく，Rename Class では 34.5%のテストメソッドがコンパイルエラーを起こすことがわかった．また同様にクラスに対するリファクタリングはテストメソッドに修正が必要となる可能性が高く，また修正行も平均 3 行ほど必要であることがわかった．それに対して，Add Parameter などではテストメソッドへの修正は 1 行ほどという結果が得られ，開発者はこのようなリファクタリングを行う際にテストコードの修正に対する躊躇は軽減されると考えられる．

今後の展望として，リファクタリングによるテストメソッドの修正が自動ツールによってどれほど緩和されるのかという調査を行うことや，リファクタリングを行なった際にテストコードを自動修正するようなツールの開発が挙げられる．

謝辞：本研究の一部は JSPS 科研費 JP18H03222，および，JSPS・国際共同研究事業の助成を受けた．

参考文献

[1] Adams, B. and McIntosh, S.: Modern Release Engineering in a Nutshell - Why Researchers Should Care, *In Proc. of the 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering*, pp. 78–90 (2016).

[2] Bibiano, A. C., Soares, V., Coutinho, D., Fernandes, E., Correia, J. L., Santos, K., Oliveira, A., Garcia, A., Gheyi, R., Fonseca, B., Ribeiro, M., Barbosa, C. and Oliveira, D.: How Does Incomplete Composite Refactoring Affect Internal Quality Attributes?, *In Proc. of the 28th International Conference on Program Comprehension*, pp. 149–159 (2020).

[3] Elish, K. O. and Alshayeb, M.: Investigating the Effect of Refactoring on Software Testing Effort, *In Proc. of the 16th Asia-Pacific Software Engineering Conference*, pp. 29–34 (2009).

[4] Kim, M., Zimmermann, T. and Nagappan, N.: A field study of refactoring challenges and benefits, *In Proc. of the 20th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, p. 50 (2012).

[5] Pantiuchina, J., Zampetti, F., Scalabrino, S., Piantadosi, V., Oliveto, R., Bavota, G. and Penta, M. D.: Why Developers Refactor Source Code: A Mining-based Study, *ACM Transactions on Software Engineering and Methodology*, Vol. 29, No. 4, pp. 29:1–29:30 (2020).

[6] Rachatasumrit, N. and Kim, M.: An empirical investigation into the impact of refactoring on regression testing, *In Proc. of the 28th IEEE International Conference on Software Maintenance*, pp. 357–366 (2012).

[7] Sabane, A., Penta, M. D., Antoniol, G. and Guéhéneuc, Y.: A Study on the Relation between Antipatterns and the Cost of Class Unit Testing, *In Proc. of the 17th European Conference on Software Maintenance and Reengineering*, pp. 167–176 (2013).

[8] Shimari, K., Ishio, T., Kanda, T. and Inoue, K.: Near-Omniscient Debugging for Java Using Size-Limited Execution Trace, *In Proc. of the 2019 IEEE International Conference on Software Maintenance and Evolution*, pp. 398–401 (2019).

[9] Tang, X., Wang, S. and Mao, K.: Will This Bug-Fixing Change Break Regression Testing?, *In Proc. of the 2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pp. 215–224 (2015).

[10] Tsantalis, N., Guana, V., Stroulia, E. and Hindle, A.: A multidimensional empirical study on refactoring activity, *In Proc. of the 2013 Center for Advanced Studies on Collaborative Research*, pp. 132–146 (2013).

[11] Tsantalis, N., Ketkar, A. and Dig, D.: RefactoringMiner 2.0, *IEEE Transactions on Software Engineering* (2020).

[12] Vahabzadeh, A., Fard, A. M. and Mesbah, A.: An empirical study of bugs in test code, *In Proc. of the 2015 IEEE International Conference on Software Maintenance and Evolution*, pp. 101–110 (2015).

[13] Vassallo, C., Zampetti, F., Romano, D., Beller, M., Panichella, A., Penta, M. D. and Zaidman, A.: Continuous Delivery Practices in a Large Financial Organization, *In Proc. of the 2016 IEEE International Conference on Software Maintenance and Evolution*, pp. 519–528 (2016).