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

信学技報
TECHNICAL REPORT OF IEICE.

リポジトリマイニングを用いた リファクタリングが開発に与える影響の測定

木村　秀平† 　肥後　芳樹† 　楠本　真二†

† 大阪大学大学院情報科学研究科
E-mail: †{s-kimura,higo,kusumoto}@ist.osaka-u.ac.jp

あらまし　リファクタリングは，ソフトウェアの品質を保持する上で重要な技術である．しかし，リファクタリングが開発に与える長期的な影響を定量的に測定する手法は確立されていない．本論文では，リファクタリングの影響として考えられる以下の 4 項目について測定する手法を提案した．(1) 依存関係を簡潔にする．(2) バグ発生数が減少する．(3) 変更行数が減少する．(4) 変更回数が減少する．本手法ではまず，同一とみなされるメソッド対をリポジトリ中から検出し，それぞれの進化を辿る．次に，その中から，一方はリファクタリングされ，他方はリファクタリングされなかったメソッドの進化の対を抽出し，上記項目を測定する．4 種の Java で記述されたオープンソースプロジェクトに対して実験を行い，リファクタリングには (1) および (2) の効果があることを確認できた．これは，企業のプロジェクトに対して行われた既存研究と同一の結果である．一方，(3) および (4) の効果は実験対象プロジェクトによって異なることがわかった．
キーワード　リファクタリング，リポジトリマイニング，ソフトウェア解析，ソフトウェア保守

Evaluation for the Value of Refactoring with Mining Software Repositories

Shuhei KIMURA†, Yoshiki HIGO†, and Shinji KUSUMOTO†

† Graduate School of Information Science and Technology, Osaka University
E-mail: †{s-kimura,higo,kusumoto}@ist.osaka-u.ac.jp

Abstract　Developers consider refactoring as an important technique in software maintenance. However, any technique to evaluate long-term effect of refactoring has not been established yet. This paper proposes a technique to evaluate the following four items as effects of refactoring, (1) simplifying dependencies among modules, (2) reducing the number of bug occurrences, (3) reducing the number of modified lines, and (4) reducing the number of modifications. In order to evaluate those items, firstly, we found pairs of methods that previously had the same body. Each of the pairs has to consist of two methods: one was refactored and the other was not refactored. After that, we compared the evolution of each method pair, and investigated whether there was a significant difference. The experiments conducted on the version history of four Java open source projects showed that the refactored methods had better effects than the non-refactored methods on (1) and (2). This was the same result as the previous research conducted on industrial software. On the other hand, the experimental results on (3) and (4) were different from project to project.
Key words　refactoring; mining software repositories; software analysis; maintenance

1. はじめに

コードの品質は，多くの開発者が様々な仕様を満たすように実装を行うため，徐々に低下してゆく [1]．このような品質の低下を抑えるために，リファクタリングという技術が用いられる．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ながら，内部構造を改善する技術のことである [2]．一般に，リファクタリングは重要な技術であると認識されており，様々なリファクタリング手法が提案されている [3]．また，エクストリームプログラミング (XP) では，リファクタリングを行うこ

とで開発コストを削減できるとしており，開発サイクルの一部としてリファクタリングが導入されている [4]．企業においてもその重要性は認識されており，マイクロソフト社では，開発中にリファクタリングを専門に行うチームを編成するなどの取り組みが行われている [5]．

このように，リファクタリングはプログラム保守の面から重要視されている．しかし，リファクタリングが及ぼす長期的な効果について定量的に測定を行った研究は少なく，リファクタリングの効果を測定する手法は確立されていない．

そこで，本論文では，オープンソースプロジェクトを対象に，リファクタリングの長期的な効果を定量的に測定する手法を提案する．本論文では，リファクタリングの効果として一般に述べられる以下の事柄について測定する手法を提案した．

効果 **1:** 依存関係が簡潔になる
効果 **2:** バグ発生数が減少する
効果 **3:** 変更行数が減少する
効果 **4:** 変更回数が減少する

リファクタリングの目的として第一に挙げられるのは，設計品質の向上である．設計品質が向上することで，以下の効果があると考えられる．

- Pull Up Method リファクタリングなどにより，重複しているコードがまとめられ，変更すべき箇所が減少する（依存関係が簡潔になる）
- コードに加えるべき変更の量が減少する
- 変更すべき箇所が明確となり，少ない変更回数で必要な機能が実装できる
- 将来発生するバグが減少する

他にも，リファクタリングの目的として可読性の向上などがあり，上記の効果に貢献していると考えられる．本論文では，リファクタリングよって生じると考えられる効果の中でも，定量的な測定が行える効果に対して測定を行うこととし，上述した 4 つの効果について測定を行う手法を提案した．

本論文で用いた手法は，擬似的に，同一のメソッドが「リファクタリングされた場合の進化」と「リファクタリングされなかった場合の進化」を比較する．リファクタリングの効果を測定する理想的な手法として，以下の手順が挙げられる．(1) 対象プロジェクトの複製を用意する．(2) 一方はリファクタリングを行って開発を進め，他方はリファクタリングをせずに開発を進める．(3) 開発終了後に両プロジェクトで比較を行う．このような手法であれば，完全に同一なメソッドの異なる進化を比較できる．しかし，リファクタリング以外の条件を同一にして両プロジェクトの開発を行う必要があることや，長い時間を要するなどの問題点がある．そのため，実験に要するコストが非常に高く，多くの実験対象に対して測定を行うことが困難である．そこで，本論文では，クローン関係にある 2 つのメソッドを同一とみなし，それらの一方がリファクタリングされ，もう他方がリファクタリングされなかった，というペアを見つけ，比較を行った．

評価実験の結果，本手法により「依存関係が簡潔になる（効果 1）」,「バグ発生数が減少する（効果 2）」効果が，オープンソースプロジェクトに対して定量的に示された．これは，企業のプロジェクトに対して測定を行った Kim らの研究 [5] と同様の結果である．また,「変更行数が減少する（効果 3）」「変更回数が減少する（効果 4）」効果はプロジェクトによってばらつきがあることがわかった．

以降，本論文は次のように構成されている．2 章では，関連する研究について述べる．3 章では，本論文で行った実験の設定について説明する．4 章では，本実験の結果を示す．5 章では，その結果を考察し，本手法の有効性について議論する．6 章では，本論文をまとめる．

2. 関 連 研 究

Kataoka らは，結合度の観点からリファクタリングの効果を測定した [6]．この研究は企業のプロジェクトを対象に，リファクタリング前後のソースコードを定量的に比較している．しかし，被験者がソースコードをリファクタリングし，その前後でメトリクス値の計測を行っているため，リファクタリングによって生じる長期的な効果は測定していない．

リファクタリングによる長期的な影響を測定した研究として，Kim らの研究が挙げられる [5]．Kim らは，開発者がリファクタリングをどのように捉えているのか，また，リファクタリングの「目に見える」利点は何か，ということについて調査を行った．この研究の中で，定量的にリファクタリングの利点を測定するため，Windows OS のリポジトリを対象に実験を行っている．リファクタリングの効果を示す指標として挙げられているのは，モジュール間の依存関係と，リリース後の不具合発生数である．これらの指標は，リファクタリングの効果として挙げられる,「依存関係が簡潔になる（効果 1）」,「バグ発生数が減少する（効果 2）」という事柄に対応していると考えられる．この研究では，依存関係の複雑さ，およびリリース後の不具合発生数は共に減少し，リファクタリングは定量的にみて効果がある，と結論づけている．

本論文は，この研究と以下の点で異なる．

オープンソースプロジェクトに対する測定: 上述した研究の実験対象は Windows OS という特殊なプロジェクトであった．Windows プロジェクトを他のプロジェクトと比較した際の特殊な点として，開発元のマイクロソフトでは，リファクタリング専門のチームを作るなど，リファクタリングを効果の高いものと捉え積極的に導入していることが挙げられる．このような環境では，リファクタリングが長期的，計画的かつ多くの労力をかけて行われるため，リファクタリングの効果が出やすいのではないかと考えられる．また，企業のプロジェクトであるため，開発自体が管理され，開発プロセスにリファクタリングが組み込まれている（定期的に行われる）こと，開発の規模が非常に大きいこと，などの特徴がある．

一方，オープンソースプロジェクトでは，リファクタリングは散発的に行われることが多く [7]，開発者のリファクタリングに対する意識も様々である．また，貢献者の数も幅広い．これらの特徴により，オープンソースプロジェクトに対する実験では結果が異なる可能性がある．

異なる測定手法: 上述した研究では，MaX [8] というツールを用いて依存関係を解析し，また，バイナリごとに不具合発生数を計測していた．本論文では，異なる手法を用いて，依存関係および不具合発生数の測定を行う．依存関係は，同一のコミットで変更されたメソッド数で表し，不具合発生数は，リファクタリングが適用された後のバグ修正コミット数で表した（詳細は 3.2 節に記載）．このような測定手法では，特殊なツールを用いず測定が行えるため，汎用性が高く，様々なプロジェクトに適用できると考えられる．また，既存研究では，リファクタリングによる影響は，バイナリレベル（COM，EXE，DLL など）で測定している．本論文では，メソッド単位で測定を行っているが，クラス単位やファイル単位での測定も可能である．

さらに，本論文では，上述の研究に加えて 2 つの「リファクタリングの効果」を定義し，より幅広くリファクタリングの効果を測定している．

変更量，変更回数: リファクタリングの効果として，将来の変更量が減少する，また，将来の変更回数が減少する，ということが考えられる．本論文では，これらの効果が本当に生じるのかについて調査を行うため，提案手法で計測する項目に含めた．将来の変更量は，リファクタリングが適用されてから変更された行数で表し，また，将来の変更回数はリファクタリングが適用されてから変更が加わった回数で表した（詳細は 3.2 節に記載）．

3. 実験設定

実験は，4 種類のオープンソースプロジェクト，log4j, commons-io，maven，および jEdit に対して行った．各プロジェクトの行数（LOC）とリビジョン数を表 1 に掲載する．本実験の入力は，Git で管理され，Java 言語で記述されたプロジェクトである．また，出力は，リファクタリングの効果を示すメトリクス値（ 3.2 節に示す）である．

3.1 実験の手順

リファクタリングの効果を測定するため，本手法では，次の 3 条件を満たすメソッドの進化のペアを特定する．(1) あるリビジョンでクローンであったメソッドの組．(2) 組の一方は，進化中にリファクタリングが一度だけ行われたメソッド．(3) 組の他方は，進化中にリファクタリングが行われなかったメソッド．

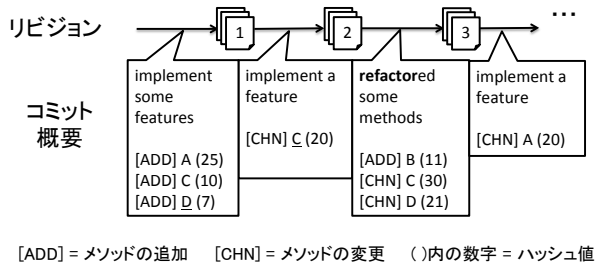
この進化のペアを，以降，実験対象の進化対と呼ぶ．以下の手順で，この進化対を見つける．

STEP 1: 対象リポジトリを Historage リポジトリに変換する

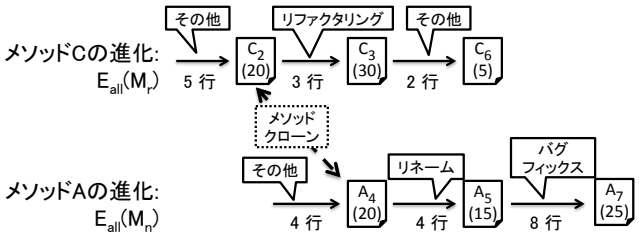
STEP 2: リファクタリングコミットを特定する

表 1: 実験対象のプロジェクトとその規模

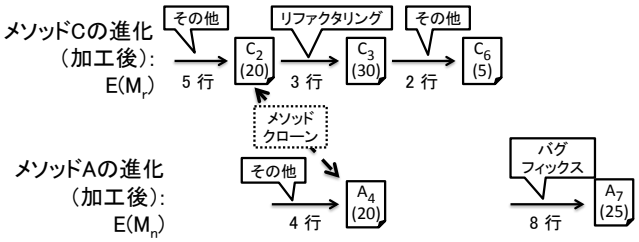
プロジェクト名	LOC（最新）	リビジョン数
log4j	30,010	3,226
commons-io	25,031	1,526
maven	72,201	9,312
jEdit	115,842	6,221



(a) STEP 3 の出力例



(b) STEP 5 の出力例



(c) STEP 6 の出力例

図 1: 実験の出力例

STEP 3: リファクタリング直前のメソッドを特定する

STEP 4: 同一メソッドとなるペアを特定する

STEP 5: メソッドの進化を取得する

STEP 6: 実験に適さないメソッドの進化を加工または削除する

この手順を行うことで得られた進化対を用いて，リファクタリングの効果を表すメトリクス値を算出する．以下，各手順を説明する．

STEP 1) 対象となるリポジトリを，**Historage** リポジトリ [9] に変換する．Historage リポジトリとは，各メソッドの変化を追従できるよう，各メソッドが 1 ファイルに変換された Git リポジトリである．正確には，各メソッドはシグネチャを表すファイルと，メソッドの中身を表すファイルの 2 ファイルに変換されるが，本論文ではシグネチャを表すファイルは除外し，メソッドの本文のみ対象としている．なお，クラス名やメソッド名などの情報は，フォルダ名として保持される．

STEP 2) リファクタリングを行ったコミットを特定する．本論文では，コミットメッセージに大文字小文字を区別せず “refactor*” を含むコミットを，リファクタリングを目的としたコミットと判断した．この手法は，Kim らの論文 [5] でも用いられている．

STEP 3) 各リファクタリングコミットに含まれるメソッド

それぞれに対し，そのコミットによって変更される前のメソッド本文を取得する．この時，リファクタリングコミットで新たに追加されたメソッドは除外する．

図 1(a) に例を示す．ファイルアイコンのまとまりはリビジョン，その中にある番号はリビジョン番号を表している．また，矢印はコミット，矢印を指している吹き出しは，そのコミットの概要を表している．吹き出し内の上部はそのコミットのコミットメッセージを表し，空行をはさみ，変更されたメソッド名とその変更内容（[ADD] は追加，[CHN] は変更）を表している．以降，リビジョン番号 n にあるメソッド A を A_n，コミット後のリビジョン番号が n になるコミットをコミット番号 n のコミットとする．この例では，コミット番号 3 でリファクタリングが行われている．コミット番号 3 で変更されたメソッドは，B，C，D である．STEP 3 では，それぞれのメソッドごとにリファクタリング前の本文を取得する．メソッド B はリファクタリングコミットで新たに追加されたメソッドなので，本文の特定は行わない．メソッド C は，C_2 がリファクタリング前の本文である．メソッド D も同様に特定し，D_1 を取得する．結果として，下線を付加したメソッドの，変更前の本文を取得することとなる．

STEP 4) STEP 3 で取得したメソッドそれぞれについて，メソッドクローンを特定する．本論文では，コードクローンを形成するメソッドのペアをメソッドクローンとする．なお，それぞれのメソッドの変数名や定数などを考慮せず完全一致した場合にコードクローンとみなす．これは，Type-2 と呼ばれるコードクローンである [10]．以降，ペアの中で，リファクタリングされたメソッドを M_r，他方を M_n と記述する．言い換えると，あるリビジョンでの M_r と M_n はメソッドクローンを形成する．

STEP 5) M_r，M_n のそれぞれに対し，メソッドの進化（変更履歴）を取得する．この進化は，リネームを含めた全変更である．この処理により，リファクタリングコミットを 1 個以上含む M_r の進化（以降，$E_{all}(M_r)$ と記述）と，リファクタリングコミットを 0 個以上含む M_n の進化（以降，$E_{all}(M_n)$ と記述）のペアが得られる．以降，このペアを進化対と呼ぶ．この時，同一メソッドの異なるリビジョンをメソッドクローンとみなしていた場合は除外する．

なお，進化対の中には，リファクタリングコミットの数が $E_{all}(M_r)$ で 1 より大きい，または $E_{all}(M_n)$ で 0 より大きい場合が存在する．このような場合は測定対象以外のリファクタリングが進化中に含まれているため，正しく測定を行えない．次の手順でこのような進化を加工または除外する．

ここで，加工または除外のため，コミットを以下の 4 種類に区分する．

リファクタリング：コミットメッセージに”refactor*”を含むコミット．これは STEP 2 で特定したコミットである

バグフィックス：コミットメッセージに”bug*”を含むコミット

リネーム：100 以上のメソッドが修正され，そのうちの 80%がリネームのコミット．これはパッケージ移動によるリネームを想定している

その他：その他のコミット

図 1(b) に，STEP 5 までを実行した例を示す．矢印はコミットを，ファイルのアイコンはある 1 つのリビジョンの 1 つのメソッドを表す．ファイルアイコン内の英数字はメソッド名とリビジョン番号を表し，括弧で囲まれた数字はそのメソッドのハッシュ値を表す．また，矢印の下にある数字は，直前のリビジョンから変更された行数を，吹き出しはコミットの種類を表している．上の図はリファクタリングされるメソッド C の進化であり（$E_{all}(M_r)$），下の図はリファクタリングされないメソッド A の進化（$E_{all}(M_n)$）である．

STEP 4 では，STEP 3 で特定したリファクタリングコミット前のメソッド（C_2）とクローンとなるメソッドを全リビジョンから探す．ここでは，A_4 が同じハッシュ値を持ち，C_2 とメソッドクローンを形成する（メソッド C は M_r となり，メソッド A は M_n となる）．

STEP 5 では，メソッドクローンとなった以降の進化を取得する．STEP 5 の結果として，図に示した進化対が得られ，リファクタリングされたメソッドの進化として 3 回変更されたメソッド C の履歴と，リファクタリングされていないメソッドの進化として 3 回変更されたメソッド A の履歴が得られる．

STEP 6) 以下の場合に進化を加工，または除外する．

（1） 進化対で 2 回以上リファクタリングが行われた場合，その進化対を除外する

（2）「リネーム」に属するコミットを含む場合，そのコミットに含まれる変更を除外する

1 番目の条件による除外は，「1 度だけリファクタリングが行われたメソッドの進化」と「リファクタリングされていないメソッドの進化」に限定するためである．さらに，2 番目の条件による除外は，パッケージ移動などによって大量のリネームが発生してしまうコミットは，機能追加やバグ修正でないにも関わらず，変更行数や同時変更数に大きな影響を及ぼしてしまうためである．以降，条件を満たしていた，または満たすよう加工された $E_{all}(M_r)$，$E_{all}(M_n)$ を，それぞれ $E(M_r)$，$E(M_n)$ と記述する．

図 1(c) に，最終的に得られた進化対の例を示す．図 1(b) との変更点として，“リネーム” という吹き出しの付いていた，上述した 2 番目の条件を満たすコミットに含まれる変更を除外している．この例には存在しなかったが，上述した 1 番目の条件を満たす場合は，進化対ごと除外する．

3.2 メトリクス値の計測

上記の手順を行うことで，リファクタリングされたメソッドと，リファクタリングされていないメソッドの進化対が複数得られる．これらを利用し，リファクタリングの効果を測定する．以下，各効果を表すメトリクスと，その計測方法を示す．

- 効果 **1**: 同一コミットで変更されるメソッド数の平均を計測した．これは，進化に含まれる各変更における，同時に変更されたメソッド数の合計を，変更回数で割った値である．
- 効果 **2**: バグ修正数を計測した．これは，進化内で「バグフィックス」に属するコミットの個数である．
- 効果 **3**: 変更行数の平均を計測した．これは，進化に含ま

れる各変更における，変更行数の合計（メソッド追加または削除の場合，変更行数は 0 とする）を変更回数で割った値である．

- 効果 **4**: 変更回数を計測した．これは，進化に含まれる，そのメソッドを変更したコミット数である．

図 1(c) を用いてメトリクス計測の例を示す．$E(M_r)$ に含まれる各リビジョンを（rm_1，rm_2，rm_3），$E(M_n)$ に含まれる各リビジョンを（nrm_1，nrm_2）とする．

効果 1 のためのメトリクス値（同一コミットで変更されるメソッド数の平均）は，各コミットで変更されたメソッド数の合計を，変更回数で割った値である．$E(M_r)$ を例に取ると，rm_1 の直前にある矢印（コミット）で変更されたメソッド数をまず求める．同様に，rm_2，rm_3 においても求め，それらの値を合計する．その合計値を変更回数（3 回）で割り，平均を求める．$E(M_n)$ についても同様にして求める．

効果 2 のためのメトリクスは，バグ修正数である．$E(M_n)$ の最終コミットのみがバグ修正なので，$E(M_r)$ では 0，$E(M_n)$ では 1 となる．

効果 3 のためのメトリクスは，進化における変更行数の平均である．$E(M_r)$ の変更行数は，各コミットで加わった変更の行数を合計したものである．なお，新しく追加または削除されたメソッドが存在する場合は，変更行数を 0 とみなす．rm_1 では 5 行，rm_2 では 3 行，rm_3 では 2 行変更されているため，これらの合計を変更された回数（3 回）で割り，変更行数の平均を求める（$(5+3+2)/3 = 3.33$）．$E(M_n)$ における変更行数の平均も，同様にして求める（$(4+8)/2 = 6.0$）．

最後に，効果 4 のためのメトリクス（変更回数）は，コミット数で表されるので，$E(M_r)$ の変更回数は 3，$E(M_n)$ の変更回数は 2 である．

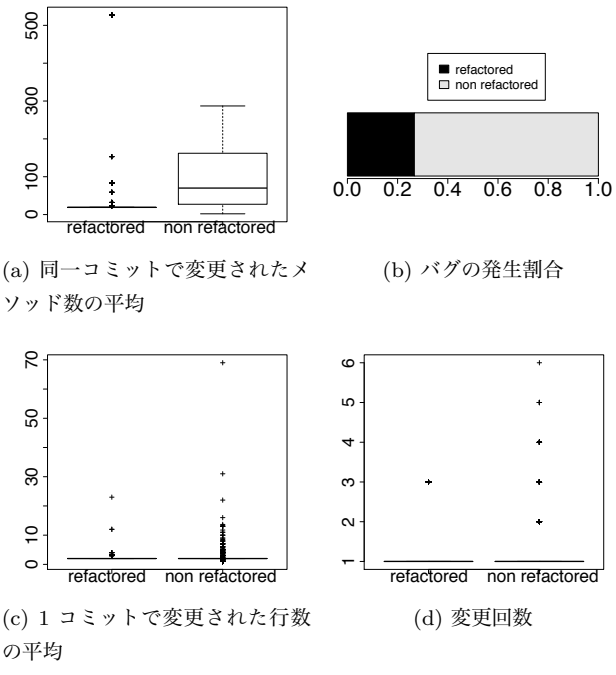
このような計測を全ての進化対に対して行う．本実験では，得られた結果を検定し，どの効果が生じるのかを測定した．

4. 実験結果

log4j，commons-io，maven, jEdit に対して上記手順を適用し，表 2 の実験結果を得た．表では 4 種類のメトリクスに対してそれぞれ検定を行った結果を示している．リファクタリングしたメソッドが有意にそのメトリクス値が低いのであれば “refactored”，反対に，リファクタリングしないメソッドが低いのであれば “non refactored”，有意差が出ない場合は “–” と表している．本実験では，二群間に差があるか否かは，マンホイットニーの U 検定を用いて，有意水準 1%で検定を行った．また，log4j に対する実験結果を図 2 に示している．以下，log4j を例に取り，結果について述べる．

表 2: 実験結果

プロジェクト （進化対数）	RQ1	RQ2	RQ3	RQ4
log4j (735)	ref	ref	ref	ref
commons-io (68)	–	ref	non ref	non ref
maven (1,617)	ref	ref	ref	ref
jEdit (2.267)	ref	ref	ref	ref



(a) 同一コミットで変更されたメソッド数の平均

(b) バグの発生割合

(c) 1 コミットで変更された行数の平均

(d) 変更回数

図 2: log4j の実験結果

図 2(a) は，あるメソッドと同時に変更されたメソッド数を，箱ひげ図として表している．左側の箱ひげ図は，リファクタリングが行われているメソッド，右側の箱ひげ図は，リファクタリングが行われていないメソッドである．U 検定の結果，$p < 2.2 \times 10^{-16}$ となり，二群間に差がある，となった．リファクタリングされているメソッド群は，中央値は 19.00，平均値は 38.48 となった．一方，リファクタリングされていないメソッド群では，中央値は 70.00，平均値 91.04 はとなった．結果として，リファクタリングによって共に変更されるメソッド数が減少する，となった．

図 2(b) は，バグ修正数の合計をそれぞれの群で算出し，割合で比較を行った帯グラフである．右側の灰色箇所が 50%を超えており，リファクタリングされていないメソッド群の方が，バグ修正数が多いことがわかる．U 検定を行ったところ，$p = 1.989 \times 10^{-5}$ となり，有意な差があるとわかった．リファクタリングされているメソッド群では，中央値は 0，平均値は 0.0119 だった．一方，リファクタリングされていないメソッド群では，中央値は 0，平均値は 0.0449 だった．結果として，リファクタリングによってバグ修正数が減少する，となった．

図 2(c) は，1 度のコミットで変更された行数を，箱ひげ図として表している．U 検定を行った結果，$p < 3.772 \times 10^{-4}$ となり，有意な差があるとわかった．リファクタリングされているメソッド群では，中央値は 2，平均値は 2.211 だった．一方，リファクタリングされていないメソッド群では，中央値は 2，平均値は 2.568 だった．結果として，リファクタリングによって変更行数が減る，となった．

図 2(d) は，何度変更されたかを箱ひげ図として表している．同様に U 検定を行い，$p < 8.207 \times 10^{-8}$ となった．リファクタリングされているメソッド群は，中央値は 1.000，平均値は

1.22 となった．一方，リファクタリングされていないメソッド群では，中央値は 1.000，平均値は 1.335 となった．結果として，リファクタリングによって変更回数が減少する，となった．

commons-io，maven，jEdit に対しても同様の手順で検定を行い，表 2 に示した結果を得た．

5. 考　　察

表 2 により，同時に変更されるメソッド数は，3 プロジェクトがリファクタリングによって減少，1 プロジェクトが有意差なしとなった．また，バグ修正数は，実験対象の全プロジェクトでリファクタリングによって減少することを示した．これらのメトリクスは，リファクタリングによって値が減少する可能性が高いといえる．一方で，変更行数，および変更回数はリファクタリングによって減少する場合と，リファクタリングによって増加する場合の両方が存在し，プロジェクトによって結果が異なった．よって，本実験で判明したリファクタリングによる効果は以下となる．

効果 **1:** 依存関係は多くのプロジェクトで簡潔になる

効果 **2:** バグの発生数は減少する

効果 **3:** 変更量が減少するかはプロジェクトによって異なる

効果 **4:** 変更回数が減少するかはプロジェクトによって異なる

効果 1 と効果 2 により．本手法が既存研究 [5] と異なった手法で同様の結果が得られたことを示している．

なお，リファクタリングされることにより，メソッドに対する変更は小さく，少なくなり，安定したコードとなると考えられる．しかし，この効果はプロジェクトによって差があり，反対に commons-io ではリファクタリングしない方がよい結果が得られた．プロジェクトごとに差が生じており，効果 3 と効果 4 に関するさらなる研究が必要だと考えられる．

6. 妥当性の脅威

進化対を生成する際，リネームコミットによる変更を取り除いている．リファクタリングの一種と考えられるリネームを取り除いているため，正しくリファクタリングによる影響を測定できない可能性がある．しかし，リネームは発生回数が多く，他のリファクタリングと同等に扱うことはできないと考えられるため，除外を行うことが適切だと考えられる．加えて，本論文ではパッケージ移動と考えられる，大量にリネームが行われた場合のみ除外を行っているため，本手法により適切にリファクタリングの効果を測定できるといえる．なお，Historage リポジトリに変換したため，クラス名やメソッド名の変更もリネームとみなされる．

本論文では，実験対象は 4 種類のみで，実験対象の増加によって結果が異なる場合がある．より多くの実験対象を用いて，プロジェクトごとの結果を総合することで，より信頼性の高い結論が得られると考えられる．

7. まとめと今後の課題

本論文では，オープンソースプロジェクトに対してリファクタリングの長期的な効果を定量的に測定する手法を提案した．評価実験を行い，「依存関係が簡潔になる」「バグ発生数が減少する」という効果が示された．この結果は，既存研究 [5] と同様である．また，オープンソースプロジェクトに対しても，企業のプロジェクトと同様にリファクタリングの効果を示すことができた．しかし，「変更量が減少する」「変更回数が減少する」という効果についてはプロジェクトごとに結果が異なったため，さらなる研究を要するといえる．

今後の課題として，より多くのプロジェクトを対象に実験を行うことが挙げられる．これにより，結果の妥当性を高めるとともに，プロジェクト間で差が発生した要因についても考察が行えると考えられる．また，メソッドクローンの判定基準を追加し，より正確に同一（とみなす）メソッドの進化を比較できるようにする．具体的には，各メソッドの入次数と出次数をカウントし，それぞれが一致する，という基準を追加する．加えて，リファクタリングの種類による分類を行うことで，より詳細なリファクタリングによる効果を考察する予定である．さらに，今回はメソッド単位で効果の測定を行ったが，モジュール単位や，プロジェクトにまたがったクローン検出を行うことで，より大きな単位でのリファクタリングによる長期的な効果を測定する予定である．

謝辞　本研究は，日本学術振興会科学研究費補助金基盤研究 (S)(課題番号：25220003)，挑戦的萌芽研究 (課題番号：24650011)，および文部科学省科学研究費補助金若手研究 (A)(課題番号：24680002) の助成を得た．

文　　献

[1] S. Eick, T. Graves, A. Karr, J. Marron, and A. Mockus, “Does code decay? assessing the evidence from change management data,” *IEEE Trans. Software Eng.*, vol. 27, no. 1, pp. 1–12, Jan/Feb 2001.

[2] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code.* Pearson Education, 1999.

[3] T. Mens and T. Tourwe, “A survey of software refactoring,” *IEEE Trans. Software Eng.*, vol. 30, no. 2, pp. 126–139, Feb. 2004.

[4] K. Beck and C. Andres, *Extreme Programming Explained: Embrace Change (2nd Edition).* Addison-Wesley Professional, 2004.

[5] M. Kim, T. Zimmermann, and N. Nagappan, “A field study of refactoring challenges and benefits,” in *FSE*, 2012, pp. 1–11.

[6] Y. Kataoka, T. Imai, H. Andou, and T. Fukaya, “A quantitative evaluation of maintainability enhancement by refactoring,” in *ICSM*, 2002, pp. 576–585.

[7] E. Murphy-Hill, C. Parnin, and A. P. Black, “How we refactor, and how we know it,” *IEEE Trans. Softw. Eng.*, vol. 38, no. 1, pp. 5–18, Jan. 2012.

[8] S. Amitabh, T. Jay, and S. Craig, “Efficient integration testing using dependency analysis,” Microsoft Research, Tech. Rep. MSR-TR, 2005.

[9] H. Hata, O. Mizuno, and T. Kikuno, “Historage: fine-grained version control system for java,” in *IWPSE-EVOL*, 2011, pp. 96–100.

[10] C. K. Roy, J. R. Cordy, and R. Koschke, “Comparison and evaluation of code clone detection techniques and tools: A qualitative approach,” *Sci. Comput. Program.*, vol. 74, no. 7, pp. 470–495, May 2009.