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.

重複コードと非重複コードにおける修正頻度の比較

佐野由希子†　　肥後　芳樹†　　楠本　真二†

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

あらまし　一般に，重複コードは非重複コードに比べて修正の行われる頻度が高いといわれている．しかし，それが事実かどうかを定量的に調査した研究は少ない．更に，既存研究についても，調査をメソッド単位やファイル単位で行い，重複コードを含むメソッド内やファイル内の修正はすべて重複コードの修正と判定しているために，実際には重複コードと関係ない修正も関係ある修正と判定してしまっている問題がある．加えて，対象ソフトウェアが少ないという問題もある．そこで本研究では，より正確に計測するため，重複コードと非重複コードの修正頻度の調査を行単位で行った．また，より一般的な結果を得るため，様々なソフトウェアに対して実験を行った．
キーワード　重複コード，修正頻度，ソフトウェア保守

Comparison of Fixing Frequency between Duplicate Code and Non-Duplicate Code

Yukiko SANO†, Yoshiki HIGO†, and Shinji KUSUMOTO†

† Graduate School of Information Science and Technology, Osaka University, 1-5 Yamadaoka, Suita-shi, 565-0871 Japan
E-mail: †{y-sano,higo,kusumoto}@ist.osaka-u.ac.jp

Abstract　In general, it is said that duplicate code is modified more frequently than non-duplicate code. However, there are few quantitative studies invstigating whether it is true. Furthermore, the previous studies investigated on the unit of programming language such as file or method, not on duplicate code itself, so that there is a possibility that modifications that are not related to duplicate code are regarded as ones that are related to duplicate code. In addition, only a small number of software systems were investigated. In this paper, therefore, to measure modification frequency more accurately, we investigated modification frequency of duplicate code and non-duplicate code by line. Also, to obtain more general results, experiments conducted on various types of software systems.
Key words　duplicate code, modification frequency, software maintenance

1. ま え が き

近年，ソフトウェア工学の研究対象の一つとして重複コードが注目を集めている．重複コードとは，ソースコード中に存在する同一，または類似したコード片のことであり，コードクローンとも呼ばれる．それらの多くは，既存システムに対する変更や拡張時における「コピーアンドペースト」による安易な機能的再利用の際に発生する [1].

一般に，重複コードの修正頻度は非重複コードの修正頻度に比べて高いといわれている．もし，あるコード片にバグが含まれていた場合，そのコード片に対応する重複コード全てについて修正を行うかどうかを検討しなければならないことが理由に挙げられる．このような作業は，大規模なソフトウェアでは非常に手間のかかる作業である．そこで，重複コードに関連した保守作業を支援するために，重複コードの検出や集約に関する研究が盛んに行われている [1].

しかし，重複コードの修正頻度が非重複コードに比べて高いというのが事実かどうかを定量的に調査した研究は少ない．更に，既存研究についても，調査の粒度や調査対象の量の面において，必ずしも正確には調査できていない場合がある．門田らの研究 [2] はファイル単位で，Lozano らの研究 [3] ではメソッド単位で修正頻度を計測している．門田らの研究は，重複コードを含むファイルは保守性が低く，また，ファイルに含まれる重複コードのサイズが大きいほど保守性が低いという結果となった．Lozano らの研究は，重複コードの存在期間の割合が高くなると，保守コストが急激に増加するという結果となった．

これらの研究では，重複コードを含むファイル内やメソッド内に加えられる修正はすべて重複コードに対する修正と判定してしまうため，実際には重複コードとは関係のない修正が重複コードに関係する修正だとみなされてしまう可能性がある．調査対象についても，門田らの研究では一つだけであり，Lozanoらの研究では四つのソフトウェアを扱っているものの，すべてJava 言語と偏ってしまっている．

そこで，本研究では，修正コストを正確に表す修正頻度を計測するため，重複コードと非重複コードの修正頻度の調査を行単位で行った．また，小規模なものや大規模なもの，Java 言語で書かれたものや C++言語で書かれたものなど，様々なソフトウェアを調査対象として選択した．そのため，より一般的な結果を得られていると期待できる．それから，既存研究とは異なる点として，変更された行数ではなく変更箇所数の観点から修正頻度の計測を行った．変更された行数というのも重要な要素だが，10 行の変更が一箇所で行われるのと，一行の変更が10 箇所で行われるのとでは，後者の方が修正に必要な労力は大きい．そのため，変更箇所数に着目することにより，より適切に修正の労力を評価できる．

2. 修正頻度の計測手法

本研究では，重複コードの修正頻度，非重複コードの修正頻度，ソースコード全体の修正頻度をそれぞれ計測し，比較を行う．本節では，その修正頻度の計測手法について説明する．

2.1 修正頻度

本研究では，修正頻度を**一リビジョン当たりの変更箇所数**として定義した．これを式で表すと，”全変更箇所数/全計測対象リビジョン数”となる．ここで，全計測対象リビジョン数とは，計測を行った期間のうち，ソースコードに対する変更が行われたリビジョン数である．全変更箇所数とは，計測を行った各リビジョンにおけるソースコードの変更箇所数の総和である．

しかし，一般に非重複コードの量と重複コードの量は等しくない．そのため，コード量に対する変更箇所数の割合が重複コードと非重複コードとで等しい場合，上記の式のままではコード量の多い方が修正頻度が高いという結果になってしまう．そこで，重複コードと非重複コードの修正頻度を公平に比較するため，重複コードと非重複コード，それぞれの行数で正規化を行う．

最終的な定義は以下の通りである．

- 重複コードの修正頻度

$$\frac{\text{重複コードの変更箇所数}}{\text{全計測対象リビジョン数}} \times \frac{\text{ソースコードの総行数}}{\text{重複コードの総行数}}$$

- 非重複コードの修正頻度

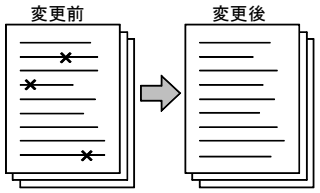
$$\frac{\text{非重複コードの変更箇所数}}{\text{全計測対象リビジョン数}} \times \frac{\text{ソースコードの総行数}}{\text{非重複コードの総行数}}$$

- ソースコード全体の修正頻度

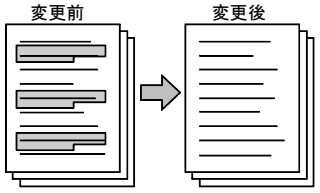
$$\frac{\text{全変更箇所数}}{\text{全計測対象リビジョン数}}$$

ここで，上記の式のソースコードの総行数，重複コードの総行数，非重複コードの総行数とは，全計測対象リビジョンにおける総行数である．すなわち，各計測対象リビジョンにおけるソースコード，重複コード，非重複コードの総行数を計測し，それらの総行数の総和をとったものである．

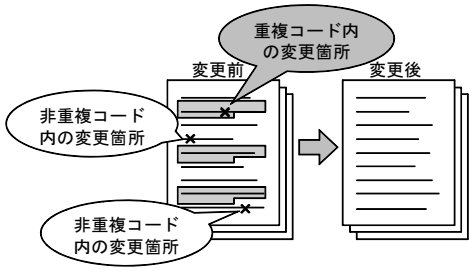
なお，上記の式は重複コードと非重複コードの修正頻度を相対的に比較するためのものであり，値の絶対値そのものには意味がない．



(a) 変更箇所の特定



(b) 重複コードの検出



(c) 変更箇所がどちらのコード内か判定

図 1　変更箇所と重複コードの位置判定

2.2 計測手順

計測手順の概要は以下の通りである．

（1） バージョン管理システムの履歴から，ソースファイルに対して変更の行われたリビジョンを検出

（2） そのリビジョンと，一つ前のリビジョンにおけるすべてのソースファイルのソースコードを取得

（3） 取得したソースコードを正規化

（4） 変更の行われたすべてのソースファイルについて，変更前と変更後の差異を検出し，変更された箇所の位置を行単位で特定（図 1(a)）．なお，変更箇所の位置は変更前のファイルを基準として計測する

（5） 変更前のリビジョンにおけるすべてのソースコードに対して重複コードを検出（図 1(b)）

（6） 変更前のリビジョンにおけるすべてのソースファイルの行数を計測

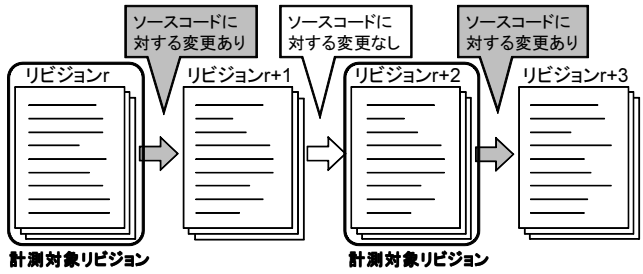


図 2　計測対象リビジョン

（7） 変更箇所が重複コード内に含まれるか非重複コード内に含まれるかを判定し，調査したい期間における重複コード内の変更個所数と非重複コード内の変更箇所数を計測（図 1(c)）

（8） 調査したい期間における，重複コードと非重複コードの行数を計測し，それらの値から調査したい期間における修正頻度を計測

2.2.1　計測対象リビジョン

ソースコードに対して変更が加えられたリビジョンの前リビジョンのみを計測対象としている．例えば，ソースコードに対する変更の状況が図 2 のようになっていた場合は，リビジョン r とリビジョン r+2 が計測対象となる．そのため，次のリビジョンにおいてソースファイルに対して変更の行われていないリビジョンは，修正頻度の式における計測対象リビジョン数には含めていない．同様に，重複コードや非重複コード，ソースコードの総行数にも，計測対象以外のリビジョンにおけるコードの行数は計上していない．

2.2.2　ソースコードの正規化

重複コードの前後にあるコメント行は，重複コードに関する物であっても非重複コードとして計上される．同様に，重複コードの前後に存在する空白行も非重複コードとして計上される．このような行に対する変更を計測対象に含めてしまうと，非重複コードの修正頻度を実際よりも高く計測してしまう可能性がある．

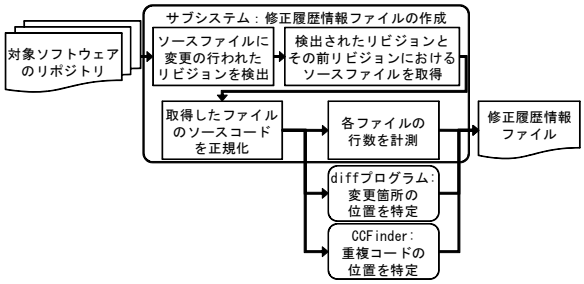
また，ソフトウェアによっては，インデントや中括弧の位置などのコーディングスタイルのみが変更された行がしばしば見つかる．しかし，このような変更はその行が重複コードであるか，あるいは非重複コードであるかには関係がない．

そのような変更を計測してしまうのを防ぐため，計測対象となったリビジョンにおけるソースコードと，その前リビジョンにおけるソースコードに対して，次のような正規化を行っている．

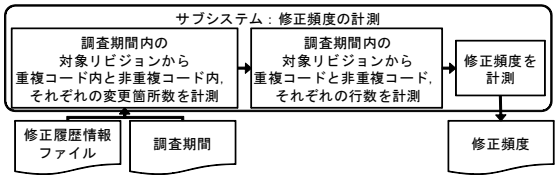
- 空白行，コメント，インデントの削除
- 中括弧のみの行を削除し，その中括弧を一つ上の行に追加

2.2.3　変更箇所数の計測

変更箇所数を計測する際，変更箇所が完全に重複コードに含まれていた場合は重複コードの変更箇所数を一つ増やし，変更箇所が完全に非重複コードに含まれていた場合は非重複コードの変更箇所数を一つ増やしている．しかし，変更箇所が重複コードと非重複コードの両方にまたがって存在している場合もある．そのような場合は，重複コードの変更箇所数と非重複コードの変更箇所数，それぞれを一つずつ増やしている．



(a) サブシステム：修正履歴情報ファイルの作成



(b) サブシステム：修正頻度の計測

図 3　ツールの概要

3. 実　　装

本手法を実現するためにツールを実装した．ツールは主に，修正履歴情報ファイルを作成する部分と，作成された修正履歴情報ファイルから修正頻度を計測する部分の二つに分かれている．

3.1　修正履歴情報ファイル

修正履歴情報ファイルは，一つの対象ソフトウェアにつき一つ生成される．記録されている情報は，各計測対象リビジョンにおける変更箇所の位置情報と重複コードの位置情報，重複コード内と非重複コード内それぞれの変更箇所の数である．

3.2　サブシステム：修正履歴情報ファイルの作成

このサブシステムで行っているのは，2.2 節の手順 (1)〜(6) である．入力は一つのソフトウェアであり，出力はそのソフトウェアに対する修正履歴情報ファイルである．このサブシステムの概要を図 3(a) に示す．また，対象ソフトウェアが使用しているバージョン管理システムは Subversion にのみ対応しており，対象ソフトウェアのプログラミング言語は Java と C/C++ に対応している．

なお，ソースコードの正規化には CommentRemover [4] を，差異の検出には diff プログラムを，重複コードの検出には CCFinder [5] を外部ツールとして使用している．

3.3　サブシステム：修正頻度の計測

このサブシステムで行っているのは，2.2 節の手順 (7)〜(8) である．修正頻度を調査したいソフトウェアの修正履歴情報ファイルと，調査したい期間を入力とし，修正頻度の計測結果を出力とする．このサブシステムの概要を図 3(b) に示す．

4. 実　　験

重複コードの修正頻度が非重複コードの修正頻度に比べて高いのかどうか調査するため，提案手法を実装したツールを用いて実験を行った．本節では，この実験について説明する．

4.1 実験対象

Sourceforge [6] にて公開されているオープンソースソフトウェアで，バージョン管理システム Subversion を使用しているものを実験対象とした．使用したソフトウェアのソフトウェア名と使用言語，実験を行った際のリビジョン数と最終リビジョンの総行数を表 1 に示す．開発期間や規模の小さなソフトウェアから大きなソフトウェアまで対象とした．また，使用言語に関しても Java と C++という代表的な二つの言語について実験を行った．

4.2 実験環境

実験を行った環境は CPU が Intel(R) Xeon(R) E5405 2.00GHz，主記憶容量が 8.00GB，OS が Windows Vista^{TM} Business である．この環境において，EclEmma の修正頻度計測には約半日，FileZilla は約三日，FreeCol と WinMerge は約五日，SQuirreL SQL Client は約一週間を要した．

4.3 実験方法

4.3.1 全体の修正頻度

まず，重複コードの修正頻度が非重複コードの修正頻度に比べて高いというのが事実かどうか調べるため，対象ソフトウェアの修正頻度の全体的な傾向を調査した．具体的には，各対象ソフトウェアについて開発期間中の全リビジョンにおける修正頻度の計測を行った．

4.3.2 修正頻度の推移

開発初期は頻繁に修正が加えられるが，開発が進むにつれだんだんと修正の行われる頻度が低下するなど，開発時期によって修正頻度の計測結果に異なる傾向が出るのではないかと仮定し，各ソフトウェアについて，修正頻度の推移を調査した．この調査では，各ソフトウェアのリビジョン数を 10 等分し，10 等分された期間それぞれについての修正頻度を計測した．例えば，表 1 の FreeCol ならば，1〜597 リビジョン，598〜1194 リビジョン，･･･，5368〜5963 リビジョンというように分割した 10 個の期間について修正頻度を計測することになる．

4.4 実験結果

4.4.1 全体の修正頻度

対象ソフトウェアの全期間についての修正頻度を計測した結果を図 4 に示す．いずれのソフトウェアでも，重複コードの修正頻度の方が非重複コードの修正頻度よりも低いという結果になった．更に，ソフトウェアの開発期間が長いと，重複コードと非重複コードの修正頻度の差も大きかった．

表 1　実験対象のソフトウェア

ソフトウェア名	言語	リビジョン数	最終リビジョンの総行数
EclEmma	Java	788	15,328
FileZilla	C++	3,450	87,282
FreeCol	Java	5,963	89,661
SQuirreL SQL Client	Java	5,351	207,376
WinMerge	C++	7,082	130,283

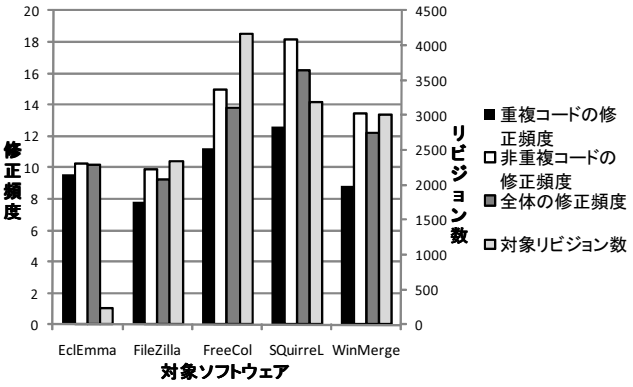


図 4　各ソフトウェアにおける修正頻度

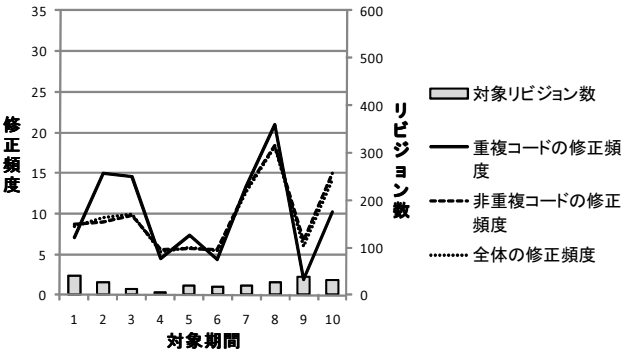


図 5　EclEmma における修正頻度の推移

4.4.2 修正頻度の推移

各ソフトウェアにおける修正頻度の推移を計測した結果を図 5〜8 に示す．x 軸の 1〜10 という数字は，10 等分されたそれぞれの期間を表している．1 が最も開発初期の期間で，10 が最も現在に近い期間である．EclEmma については，重複コードの修正頻度の方が非重複コードよりも高い期間も，逆に重複コードの方が低い期間も半分ずつであった．FileZilla と FreeCol, WinMerge については，10 等分した期間の内，九つの期間において重複コードの修正頻度の方が非重複コードよりも低かった．SQuirreL SQL Client については，すべての期間において重複コードの修正頻度の方が非重複コードよりも低かった．また，開発期間の長いソフトウェアは，10 等分した各期間においても重複コードと非重複コードの修正頻度との差が大きかった．

5. 考　　察

本節では，4.4 節の実験結果について考察していく．

5.1 全体の修正頻度

図 4 より，重複コードの修正頻度の方が非重複コードの修正頻度に比べて低い傾向にあることがわかった．更に，その傾向は開発期間が長いソフトウェアで顕著である．一般には，重複コードの方が非重複コードよりも修正頻度が高いといわれているが，まったく逆の結果となった．

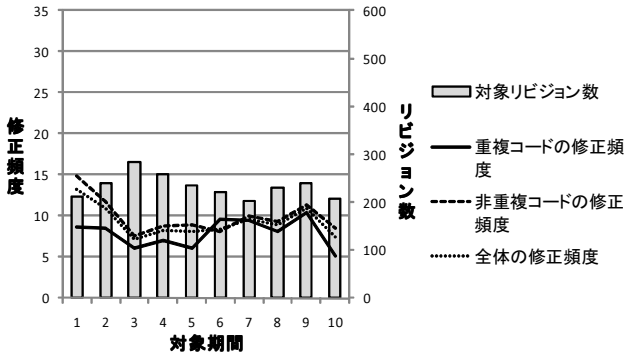


図 6　FileZilla における修正頻度の推移

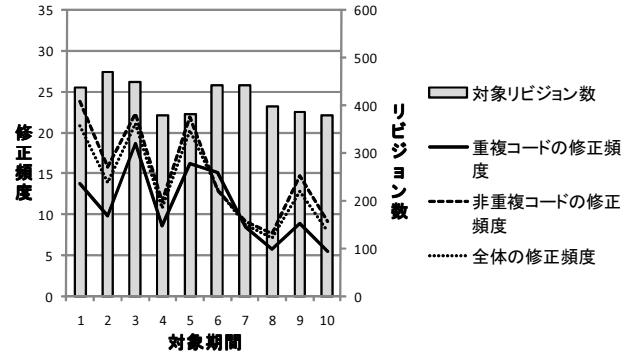


図 7　FreeCol における修正頻度の推移

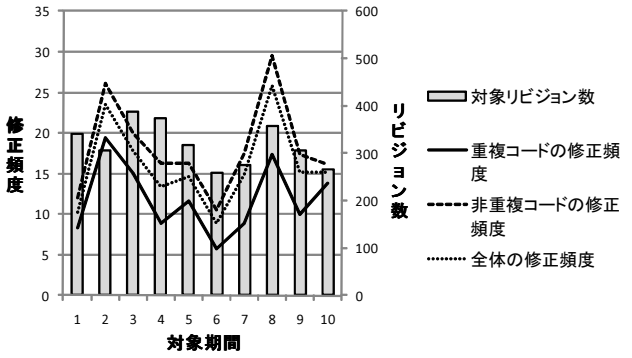


図 8　SQuirreL SQL Client における修正頻度の推移

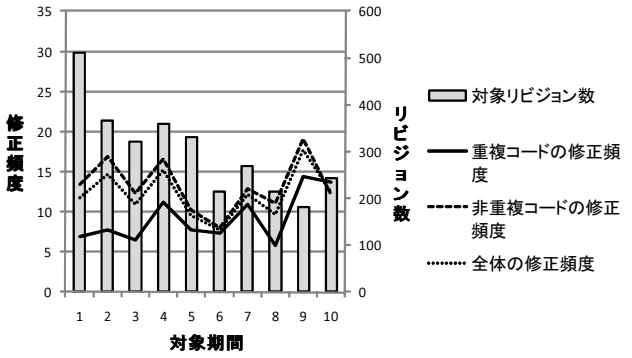


図 9　WinMerge における修正頻度の推移

重複コードの方が修正頻度の低くなった原因としては，機能追加などを行う際に動作の安定したコードを再利用していることが考えられる．新たにコードを書くよりも，既に動作を確認できているコードを再利用した方が，バグの混入する可能性が低い．

その他に考えられる原因としては，コード生成ツールの生成コードである．自動的にコードを生成するため，同じ機能のコードを生成した際は重複コードとなるが，安定性は高い．

5.2　修正頻度の推移

FileZilla や FreeCol，SQuirreL SQL Client，WinMerge の結果から，重複コードの修正頻度が安定して非重複コードの修正頻度より低いことが読み取れる．

なお，EclEmma の結果からはっきりとした傾向を読み取れなかったのは，開発期間が短すぎたせいではないかと推測される．

WinMerge について，最も重複コードと非重複コードの修正頻度の差が大きかった期間 2 と，重複コードの修正頻度が非重複コードの修正頻度を上回っていた期間 10 の調査を行った．その結果，期間 10 ではテストケースに対する修正が多いことがわかった．対象リビジョン数 244 のうち，49 個のリビジョンにおいて，テストケースへの修正が行われていた．一方，期間 2 の対象リビジョン数は 366 だが，テストケースに対する修正は一切行われていなかった．更に，テストケースのソースコードを調査したところ，その 88.3%が重複コードであった．よって，期間 10 で重複コードの修正頻度が非重複コードの修正頻度を上回っていたのはテストケースに対する修正が原因と考えられる．しかし，テストケースに対する修正はソフトウェアの保守作業においてあまり重要ではないと思われる．このことから，期間 10 においても重複コードが保守コストを増加しているとは一概にいえないだろう．

また，開発時期によってソースコード全体の修正頻度に何か違った傾向があるのではないかと予想したが，ソフトウェア間に共通した傾向は特に見られなかった．

5.3　ま と め

全体の修正頻度を計測したところ，重複コードの修正頻度の方が非重複コードより低く，加えて，その傾向は開発期間の長いソフトウェアの方が顕著であった．更に，修正頻度の推移を調査したところ，その傾向は開発期間を通して安定していることがわかった．このことから，一般には重複コードの方が非重複コードよりも修正頻度が高いといわれているが，実際には重複コードの方が修正頻度が低いといえるだろう．

5.4　妥当性への脅威

本研究の妥当性を脅かす要因として，以下の点が挙げられる．

修正に要する作業量の違い

本研究では，一箇所の修正を行うために必要な作業量は全て等しいという仮定の上で調査を行っている．しかし実際には，少ない作業量で修正が可能な箇所もあれば，一箇所を修正するのに多大な作業量を要する箇所も存在すると考えられる．このため，本研究で調査した内容は，厳密に修正の作業量を評価できているとはいえない．また，重複コードへの修正は一貫性を保つことが必要であり，修正に一貫性が無い場合 (修正を加えたコード片と対応する重複コードに対して修正漏れがあった場合)，後のリビジョンで再度同じ内容の修正を検討する必要があ

るため，修正に一貫性がある場合と比べて修正に要する作業量は大きくなると考えられる．しかし，本研究では修正の一貫性の有無に関わらず全て同じ結果となる．そのため，この観点からも，厳密に修正の作業量を評価できているとはいえない．

修正箇所の判別

本研究では，連続した行に修正が加えられた場合，一箇所の修正と判別している．しかし，この判別方法では，本来は一箇所の修正とみなされるべき修正が，途中修正しない行が存在すると，複数の修正とみなされてしまうおそれがある．また，本来は複数箇所の修正とみなされるべき修正が，偶然連続した行に加えられた場合，一箇所の修正とみなされてしまうおそれがある．このため，厳密に修正箇所を調査して測定を行った場合，本研究の結果とは異なる結果となる可能性がある．

修正の内容

本研究では，連続する二つのリビジョン間でソースファイルに修正が加えられた場合，修正内容に関わりなく全ての修正を計測している．しかし，この方法では，例えばソースコードのフォーマット変換など，バグ修正や機能追加などに関係のない修正も修正箇所として計測することになる．2.2.2 節で述べたように，そのような修正の計測を防ぐためソースコードの正規化を行ってはいるが，完全に防ぐことはできない．このような修正はソースコードの内容に本質的な関わりを持たないため，これらを結果に含めることは適切ではないと考えられる．

重複コード検出ツールの種類

本研究では，重複コードの検出には CCFinder のみを使用しているが，別の検出ツールを使用すれば計測結果が異なる可能性がある．なるべく同条件で重複コードを検出できるよう，検出する重複コードのサイズや変数等のパラメータ化の設定を調整していても，検出ツールが異なれば検出結果がまったく等しくなるわけではない．また，CCFinder は字句単位で重複コードを検出を行うが，行単位で検出を行うツールやプログラム依存グラフを用いて検出を行うツールも開発されている．そのような検出ツールを用いれば，検出結果として出力される重複コードが大幅に異なってくることも考えられる．そして，重複コードの検出結果が異なれば修正頻度の計測結果も異なってくるため，どのような検出ツールを使用しても今回のような傾向が見られるとは限らない．また，本研究では，変数名などの名前を正規化した上で 30 字句以上の重複コードを検出するよう CCFinder を設定しているが，このような設定を変更することでも，本研究と異なる結果となる可能性がある．

対象ソフトウェアの種類

本研究では，オープンソースソフトウェアに対してしか実験を行っていないため，商用ソフトウェアでは違った結果が出る可能性がある．一般に，商用ソフトウェアはオープンソースソフトウェアに比べて重複コードの割合が高いといわれている．そのため，オープンソースソフトウェアでは重複コードをうまく活用できていたが，商用ソフトウェアでは重複コードの管理が行き届かず，修正頻度に悪影響を与えているかもしれない．また，オープンソースソフトウェアは商用ソフトウェアに比べて多数の開発者が関わっており，各個人のスキルのばらつきも大きいといわれている．このことからも，重複コードの作られ方や性質にそれぞれ異なる特徴が表れている可能性がある．

期間の分割方法

本研究では，修正頻度の推移を調査する際，リビジョン数で機械的な分割を行ったが，この分割方法は適切ではないかもしれない．異なる分割方法を用いれば，今回の結果とは異なる特徴が表れる可能性がある．例えば，バージョンの区切りごとに分割して調査を行えば，各バージョンごとの修正頻度の特徴を見られるかもしれない．あるいは，各バージョンをいくつかの期間に分割することで，リリースから次期リリースまでの修正頻度の推移について，バージョン間に共通した特徴を発見できるかもしれない．

6. あとがき

本研究では，重複コードと非重複コード，ソースコード全体，それぞれの修正頻度を計測する手法を提案した．本手法では，変更箇所の位置も重複コードの位置も行単位で特定することにより，計測精度を高めた．また，従来の計測手法は変更された行数を重視しているのに対し，本手法では変更の行われた箇所数を重視した．これにより，従来手法よりも修正にかかるコストを反映した値を計測できると思われる．

更に，本手法をツールとして実装し，そのツールを用いて五つのオープンソースソフトウェアに対して修正頻度の調査を行った．その結果，重複コードの修正頻度は非重複コードの修正頻度に比べて低い傾向にあることがわかった．これは，一般に重複コードの修正頻度は非重複コードの修正頻度に比べて高いといわれているのとは逆の結果である．

今後の課題としては，検出された修正とバグ修正との関連を調査するなど，ソフトウェアの保守コストに重複コードがどのような影響を与えているのか調べていきたい．また，より多くのソフトウェアに対して調査を行い，より一般的な結果を得たい．

謝辞　本研究は，一部，文部科学省「次世代 IT 基盤構築のための研究開発」（研究開発領域名：ソフトウェア構築状況の可視化記述の開発普及）の委託に基づいて行われている．また，文部科学省科学研究費補助金基盤研究 (C)(課題番号:20500033) の助成を得て行われている．

文　献

[1] 肥後, 楠本, 井上：“コードクローン検出とその関連技術”, 電子情報通信学会論文誌, **J91-D**, 6, pp. 1465–1481 (2008).

[2] 門田, 佐藤, 神谷, 松本：“コードクローンに基づくレガシーソフトウェアの品質の分析”, 情報処理学会論文誌, **44**, 8, pp. 2178–2188 (2003).

[3] A. Lozano and M. Wermelinger: “Assessing the effect of clones on changeability”, International Conference on Software Maintenance, pp. 227–236 (2008).

[4] “commentremover”. http://www-sdl.ist.osaka-u.ac.jp/~higo/cgi-bin/moin.cgi/commentremover.

[5] T. Kamiya, S. Kusumoto and K. Inoue: “CCFinder: A multi-linguistic token-based code clone detection system for large scale source code”, IEEE Transactions on Software Engineering, **28**, 7, pp. 654–670 (2002).

[6] “Sourceforge.net: Find and develop open source software”. http://sourceforge.net/.