修士学位論文

題目

オープンソースソフトウェアにおける重複コードと非重複コードの修正頻度計測とその比較

指導教員
楠本 真二 教授

報告者
佐野　由希子

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

平成 21 年度 修士学位論文

オープンソースソフトウェアにおける重複コードと非重複コードの修正頻度計測とその比較

佐野　由希子

内容梗概

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．その理由としては，ある重複コード片にバグが見つかった場合，そのコード片に対応するすべての重複コードに対して同じ修正を行う必要が生じる可能性があることが挙げられる．この考えに基づき，ソフトウェアの修正作業を容易にするため，重複コードの修正や集約に関する研究が数多く行われている．

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．対象ソフトウェアが少ないと，実験対象に偏った結果となってしまい，一般化できないことがある．

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．調査の結果，開発期間の短いソフトウェアに関しては重複コードと非重複コードの修正頻度に明確な差は見られないが，開発期間の長いソフトウェアでは重複コードの方が修正頻度が低いことがわかった．

主な用語

重複コード
修正頻度
ソフトウェア保守

目 次

1　まえがき

近年，ソフトウェア工学の研究対象の 1 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に問題がある．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危険がある．調査対象についても，門田らの研究では 1 つだけであり，Lozano らの研究では 4 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，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，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 1 箇所で行われるのと，1 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．

調査の結果，開発期間の短いソフトウェアに関しては重複コードと非重複コードの修正頻度に明確な差は見られなかったが，開発期間の長いソフトウェアに関しては重複コードの方が非重複コードよりも修正頻度が低いことがわかった．

以降 2 章では，関連研究を紹介し，3 章では，重複コードの発生原因，重複コード検出ツール CCFinder，バージョン管理システムについて説明する．4 章では，重複コードと非重複コード，ソースコード全体，それぞの修正頻度を計測する手法の提案を行う．5 章では，4 章で提案した手法の実装について説明する．6 章では，4 章で提案した手法を用いて行った調査実験とその結果について説

明し，その考察を 7 章で行う．最後に，8 章で本研究のまとめと今後の課題について述べる．

2　関連研究

門田らは，重複コードとソフトウェア品質との関係について，ファイル単位で分析を行った [2]．その際，ソフトウェア品質の指標の 1 つに保守性を用いた．ここでいう保守性とは，各ファイルの改版数であり，改版数が大きなファイルほど保守性が低い．ある公共機関の業務に用いられる，COBOL で書かれた大規模なレガシーソフトウェアについてソフトウェア品質を調査した．その結果，重複コードを含むファイルは含まないファイルよりも保守性が約 40%低く，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かったと報告している．

Lozano らは，重複コードが有害であるといわれているのが事実か否かを調査するため，CloneTracker[4] というツールを作成した [5]．CloneTracker は，各リビジョンにおいてどのメソッドが重複コードを含んでいて，どのメソッドが変更されたかを計測するツールである．このツールの実験として，準備的なケーススタディを行った．その結果，メソッドは重複コードを含んでいると，より頻繁に変更されるものの，別のメソッドと同時に変更される量は減少した．これは，重複コードは変更を増加させるが，プログラマが重複コードの存在に気付かず，対応する重複コードに対する変更が遅れてしまうという説を裏付けるような結果である．しかし，今回の結果だけでは，確かなこととはいえないと報告している．

更に Lozano らは，ソフトウェア保守に対する重複コードの影響についてメソッド単位で調査を行った [3]．その評価指標として，likelihood（あるメソッドに対して変更の行われる割合），impact（あるメソッドが変更される際，同時に変更されるメソッド数の割合），work（likelihood と impact の積）を定義し，work を保守コストと定義した．4 つのオープンソースの Java プロジェクトについて，同一メソッドで重複コードを含む期間と含まない期間，常に重複コードを含むメソッドと常に含まないメソッド，それぞれの保守コストを比較した結果，likelihood はどちらもあまり変わらなかったが，impact に関しては重複コードを含んでいるほうが大きくなる傾向を示すものがいくつかあった．そして，重複コードの存在期間の割合が高くなると，work が急激に増加したと報告している．

また，Lozano らは，重複コードやメソッドの持つ特徴と，ソースコードの可変性の減少との関係についても調査している [6]．可変性とはソースコードの修正しやすさであり，減少すればソフトウェア保守を妨げる要因となりうる．可変性を評価する尺度にはメソッドの変更された回数とその割合，メソッドが変更される際に同時に変更されるメソッド数の割合と同時に変更されるパッケージの数，同時に変更されるメソッドの分布の 5 つを用いた．調査の結果，重複コードは可変性の減少を招く危険はあるものの，メソッドの長さやファンアウト，複雑さといった特徴の方が可変性の減少に対してより強い影響を与えていた．この結果から，ソフトウェア保守において重複コードを最優先で処理すべきとはいえないだろうと報告している．

Krinke らは，もし重複コードが非重複コードよりも安定性に欠くなら，保守において重複コードにかかるコストが高いと仮定し，システムの進化において，重複コードは非重複コードよりも安定しているかについて調査を行った [7]．調査対象は，5 つの大規模なシステムから，1 週間ごとに区切っ

たバージョンを 200 ずつ抽出して使用した．それらの調査対象において，重複コードと非重複コードのそれぞれに対して行われる追加，削除，変更の行数を計測し，重複コードと非重複コード，それぞれに対する割合を比較した．その結果，重複コードの方が追加，削除，変更の行われる割合の平均が低く，また，追加，削除，変更が行われる割合は非重複コードの方が重複コードより高い週が多かった．この結果から，重複コードは非重複コードよりも安定しており，一般的に，非重複コードの保守よりも重複コードの保守の方がコストが高いとは仮定できないと報告している．

Eick らは，ソースコードを運用・保守していくうちに Code Decay（ソースコードが保守しがたくなること）が引き起こされているのか調査している [8]．彼らは，Code Decay を示す指標として，変更によって増加（または減少）した行数や変更に要した時間，変更に関わった開発者数，ある期間中にあるモジュールに関わる変更を行った数やある変更に関わったファイル数など，様々なものを提案した．そして，15 年にわたって運用された大規模なソフトウェアにつて調査を行った結果，1 つの変更に要するコストが増加していく傾向にあることをつきとめたと報告している．

Nils らは，Type-1 のコードクローンに関して，コードクローンが生成され発展する様子を個々のコード片に着目してモデル化する手法を提案している [9]．Type-1 のコードクローンとは，重複コードの中でも，空白やタブの有無，括弧の位置などのコーディングスタイルを除くと，完全に一致するもののことをいう．また，その提案手法を 9 つのオープンソースソフトウェアに対して適応し，コードクローンの発展の様子を調査している．その結果，コードクローンの割合は時間経過とともに減少していること，コードクローンは平均で約 1 年以上コードクローンとして存在していること，また，コードクローンに一貫性のない変更が加わった場合，その変更がのちのバージョンにおいて一貫性のある変更に修復されることは少ないことなどを報告している．

3 準備

3.1 重複コード

ある系列中に存在する 2 つの部分系列 α，β が等価であるとき，α と β は互いにクローンであるという．それぞれを真に包含する如何なる部分系列も等価でないとき，α，β を極大クローンと呼ぶ．ソースコード中でのクローンを特に重複コードという [10].

重複コードがソフトウェアの中に作りこまれる，もしくは発生する原因として次のようなものがある [11][12][13].

既存コードのコピーとペーストによる再利用

近年のソフトウェア設計手法を利用すれば，構造化や再利用可能な設計が可能である．しかし，コードの再利用が容易になったために，現実にはコピーとペーストによる場当たり的な既存コードの再利用が多く行われるようになった．

コーディングスタイル

規則的に必要なコードはスタイルとして同じように記述される場合がある．例えば，ユーザインターフェース処理を記述するコードなどである．

定型処理

定義上簡単で頻繁に用いられる処理．例えば，給与税の計算や，キューの挿入処理，データ構造アクセス処理などである．

プログラミング言語に適切な機能の欠如

抽象データ型や，ローカル変数を用いられない場合には，同じようなアルゴリズムを持った処理を繰り返し書かなくてはならないことがある．

パフォーマンス改善

リアルタイムシステムなど時間制約のあるシステムにおいて，インライン展開などの機能が提供されていない場合に，特定のコード片を意図的に繰り返し書くことによってパフォーマンスの改善を図ることがある．

コード生成ツールの生成コード

コード生成ツールにおいて，類似した処理を目的としたコードの生成には，識別子名等の違いはあろうとも，あらかじめ決められたコードをベースにして自動的に生成されるため，類似したコードが生成される．

複数のプラットフォームに対応したコード

複数の OS(Linux, FreeBSD, HP-UX や AIX など) や CPU(i386 系, amd64 系, alpha や sparc64 など) に対応したソフトウェアは，各プラットフォーム用のコード部分に重複した処理が存在する傾向が強い．

偶然

偶然に，開発者が同一のコード片を書いてしまう場合もあるが，大きな重複コードになる可能性は低い．

3.2 重複コード検出ツール CCFinder

CCFinder は，単一または複数のファイルのソースコード中から全ての重複コードを検出し，それらの位置情報を出力する [11]．CCFinder の持つ主な特徴は次の通りである．

細粒度の重複コードを検出

字句解析を行うことにより，字句単位での重複コードを検出する．

大規模ソフトウェアを実用的な時間とメモリで解析可能

例えば 10MLOC のソースコードを 68 分 (実行環境 Pentium3 650MHz RAM 1GB) で解析可能である [10].

様々なプログラミング言語に対応可能

言語依存部分を取り替えることで，様々なプログラミング言語に対応できる．現在は，C，C++，Java，COBOL/COBOLS，Fortran，Emacs Lisp に対応している．またプレーンテキストに対しても，分かち書きされた文章として解析可能となっており，未対応の言語に対しても完全一致判定による重複コードは検出可能である．

実用的に意味の持たない重複コードを取り除く

- 重複コードは小さくなればなるほど偶然の一致の可能性が高くなるが，最小一致字句数を指定することができるため，そのような重複コードの検出を防ぐことができる．
- モジュールの区切りを認識し，複数のモジュールにまたがって続くような重複コードについては，各モジュールの区切りごとに分割して検出する．

ある程度の違いは吸収可能

- ソースコード中に含まれるユーザ定義名，定数をパラメータ化することで，その違いを吸収できる．
- クラススコープや名前空間による複雑な名前の正規化を行うことで，その違いを吸収できる．

- その他，テーブル初期化コード，可視性キーワード (protected,public,private 等)，コンパウンド・ブロックの中括弧表記等の違いも吸収できる．

3.3 バージョン管理システム

バージョン管理システムは，主にソフトウェアの開発過程で用いられる，ソースコードなどのソフトウェアを構成するファイルを管理するためのシステムである．利用者はリポジトリと呼ばれるデータ保管場所から必要なファイルを自分の手元のマシンに取り出し（チェックアウト），取り出したファイルを編集する．そして，編集を終えたファイルはリポジトリに戻す（チェックイン，コミット）．リポジトリはチェックインされるごとに，更新差分情報を蓄積していく．更新差分情報には，更新の行われた時刻や更新を行った人物，更新の際に変更されたファイルといった情報が含まれる．また，更新が行われる度にリビジョン番号と呼ばれる，バージョン管理用の番号がインクリメントされる．このような情報を蓄積しておくことにより，利用者は最新のファイルだけではなく，蓄積されている複数のリビジョンから任意のリビジョンを選んで取り出すことができる．

バージョン管理システムには集中型と分散型が存在する．集中型はリポジトリが単一で中央サーバの役割を果たす．それに対し，分散型は開発者それぞれがローカルリポジトリを持ち，それぞれのローカルリポジトリは相互に同期をとることができる．

代表的なバージョン管理システムについて，以下で説明する．

CVS（Concurrent Versions System）

古くから存在し，最も普及していた集中型バージョン管理システムで，現在でも広く利用されている．しかし，ディレクトリの移動やファイル名の変更削除などをうまく扱えない，複数のファイルを同時にコミットしても 1 回のコミットとはみなされず，複数のコミットが短時間に行われたとみなされてしまうといった欠点がある．

Subversion

CVS の問題点を解決すべく開発された集中型バージョン管理システムであり，現在最も広く利用されている．操作性は CVS とよく似ており，移行が容易である．編集したファイルの更新差分情報の送信も効率よく行われ，高速になっている．

Git

Linux カーネルのソースコード管理を目的として開発された分散バージョン管理システムであり，動作速度が速い．代表的なバージョン管理システムのほとんどが CVS と類似の操作性を提供する中，独自の操作性を提供している．

Mercurial

大規模プロジェクトでも利用できるように設計された分散バージョン管理システムであり，機能を抑えるかわりに高いスケーラビリティと高速な動作を実現している．操作性は CVS に似ている．

Bazaar

使いやすさや正確さ，柔軟性に焦点をあてて開発されたバージョン管理システムである．分散型だが，中央サーバあり・なし両方での動作をサポートしており，集中管理から非集中管理までさまざまなワークフローを実現できるという特徴がある．操作性は CVS に似ている．

Monotone

性能よりも整合性の実現に注力して開発された分散バージョン管理システムであり，マージ機能が優れている．操作性は CVS に似ている．

Revision Control System（RCS）

初期に開発されたバージョン管理システムの 1 つだが，個人向けに開発されており，同時に複数のファイルを操作することはできない．そのため，複数のユーザが同時に作業できないため，プロジェクト管理には不向きである．

Source Code Control System（SCCS）

世界初のバージョン管理システムで，RCS が開発されるまでは，ほとんど唯一のバージョン管理システムとして広く利用されていた．現在はあまり利用されていない．

4　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．

4.1　修正頻度

本研究では，修正頻度を 1 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．

4.2　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，1 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)）
    - このとき，変更箇所の位置は変更前のファイルを基準として計測
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
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
9. 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

4.2.1 計測対象リビジョン

4.2 節における手順 1 で検出さ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．

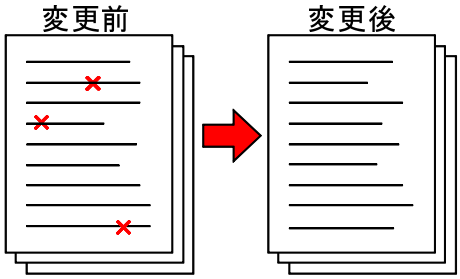
4.2.2 ソースコードの正規化

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．

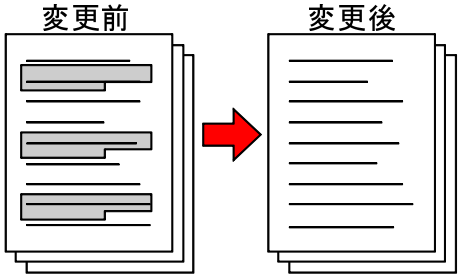
また，ソフトウェアによっては，インデントや中括弧の位置などのコーディングスタイルのみが変更された行がしばしば見つかる．しかし，このような変更はその行が重複コードであるか，あるいは非重複コードであるかには関係がない．

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．

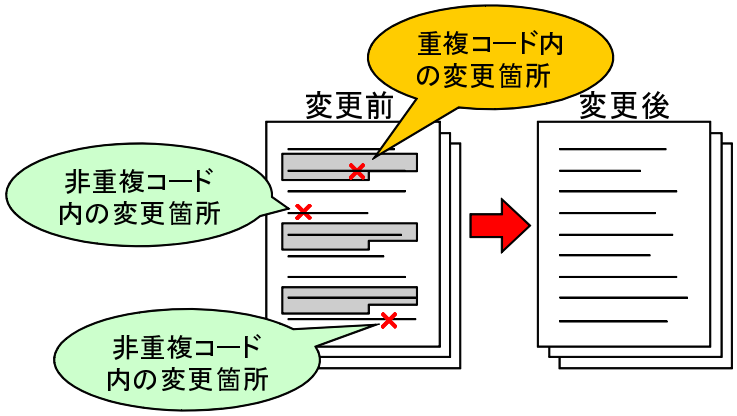
- 空白行の削除
- コメントの削除



(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

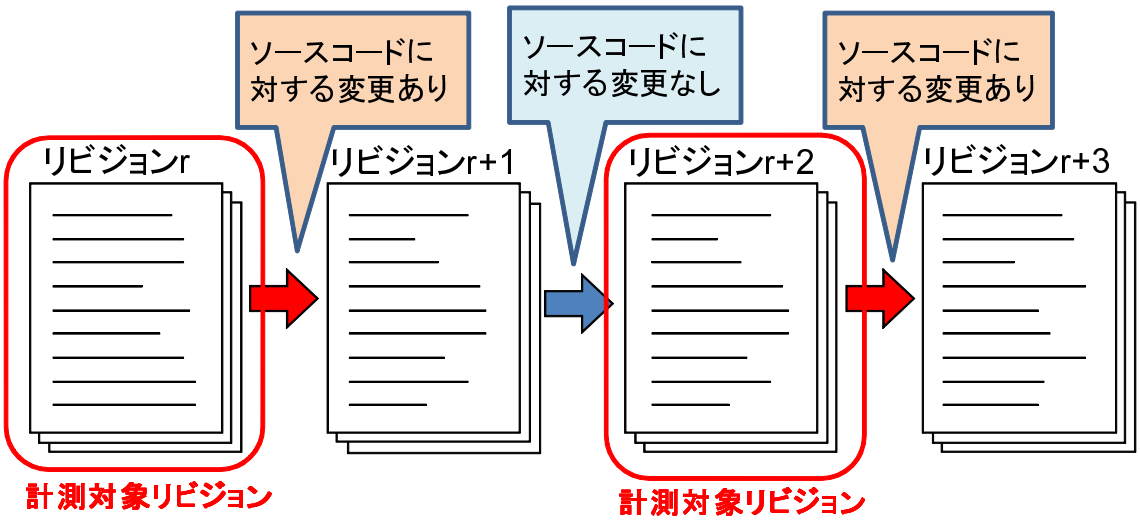


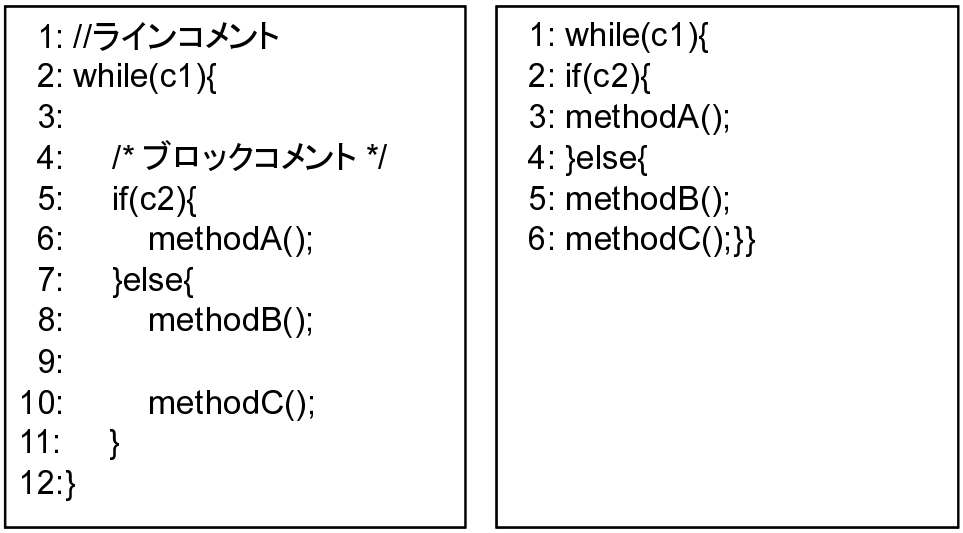
図 2: 計測対象リビジョン

- 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 1 つ上の行に追加

正規化を行った例を図 3 に示す．図 3(a) に対して正規化を行うと，図 3(b) のようになる．

4.2.3 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 1 つ増やし，逆に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 1 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 1 つずつ増やしている．



(a) 正規化前　　(b) 正規化後

図 3: 正規化の例

5 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 2 つに分かれている．

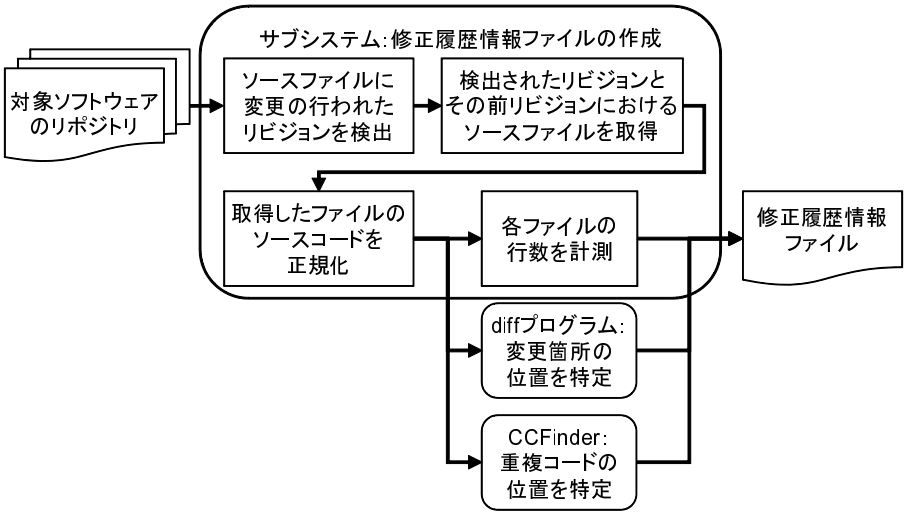
5.1 修正履歴情報ファイル

修正履歴情報ファイルは，1 つの対象ソフトウェアにつき 1 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．

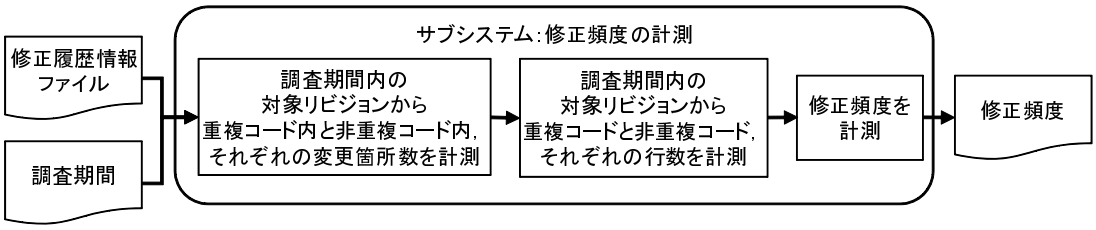
5.2 サブシステム：修正履歴情報ファイルの作成

このサブシステムで行っているのは，4.2 節の手順 1〜6 である．入力は 1 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 4(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[14] 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 を外部ツールとして使用している．



(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 4: ツールの概要

5.3 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，4.2 節の手順 7〜9 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 4(b) に示す.

6 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．

6.1 実験対象

Sourceforge[15] 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 2 つの言語について実験を行った．

6.2 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 3 日，FreeCol と WinMerge は約 5 日，SQuirreL SQL Client は約 1 週間を要した．

6.3 実験方法

6.3.1 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．

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

6.3.2 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．

6.4 実験結果

6.4.1 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 5 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．

6.4.2 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 6〜9 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，9 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．

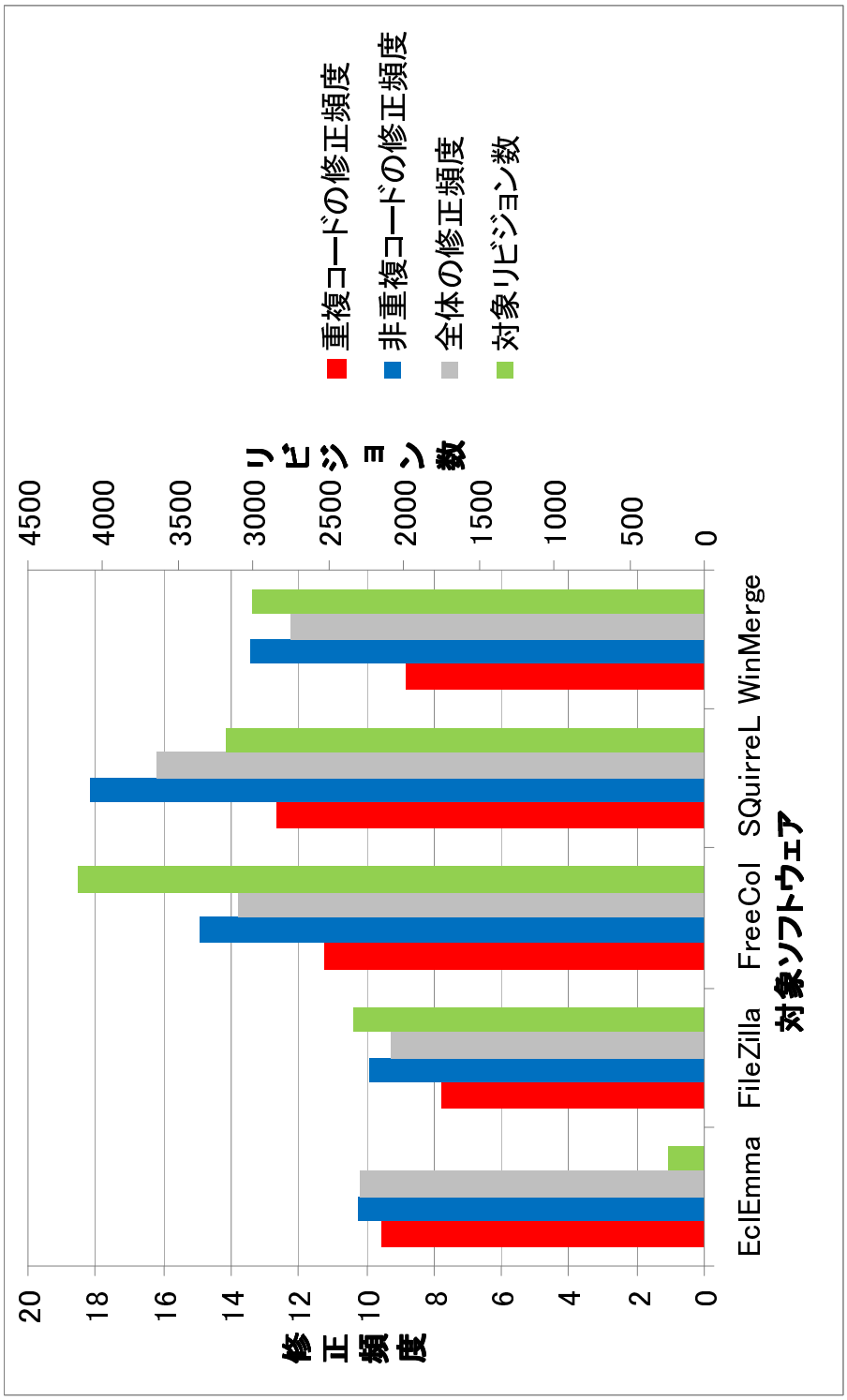


図 5: 各ソフトウェアにおける修正頻度

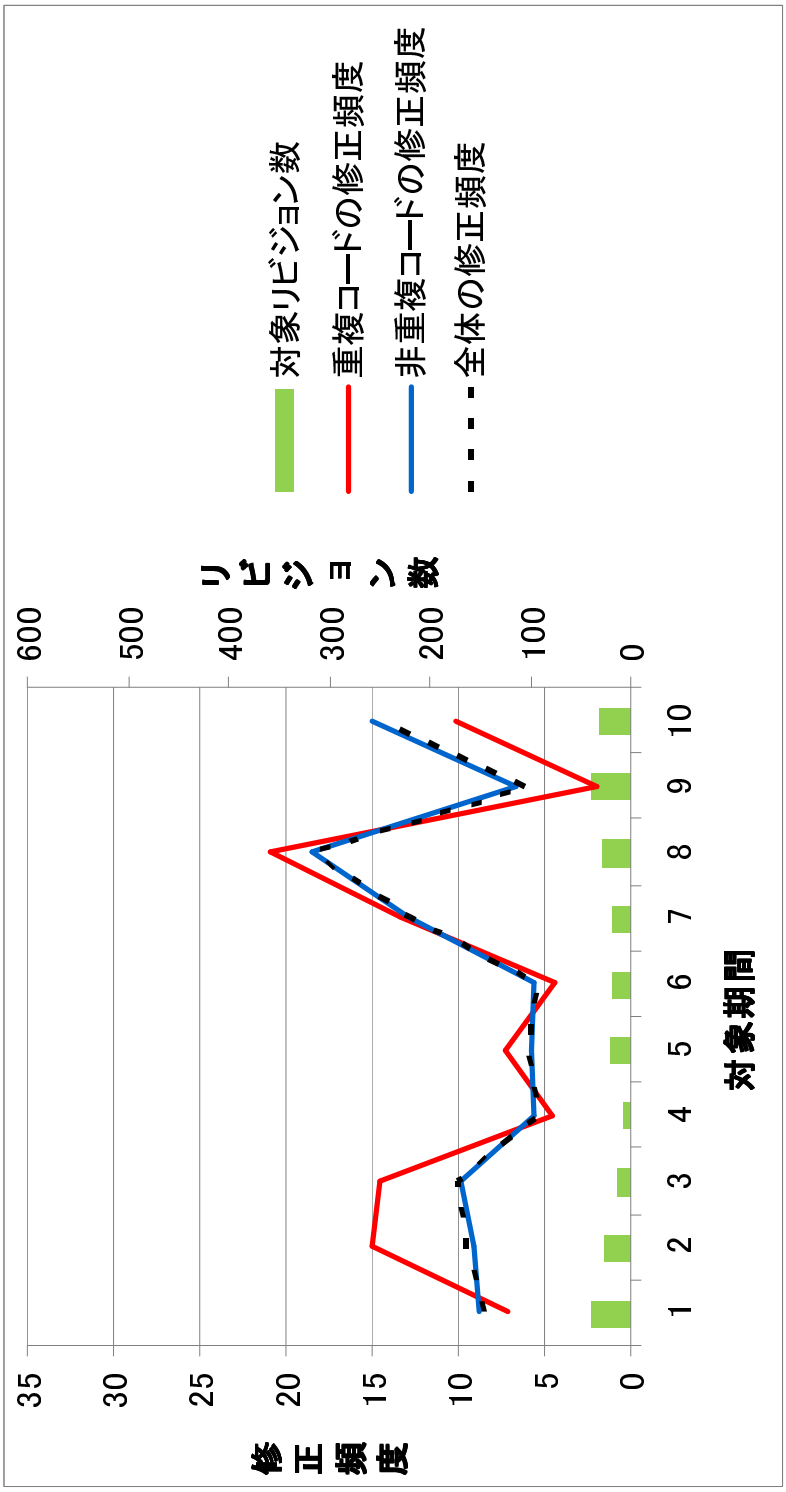


図 6: EclEmma における修正頻度の推移

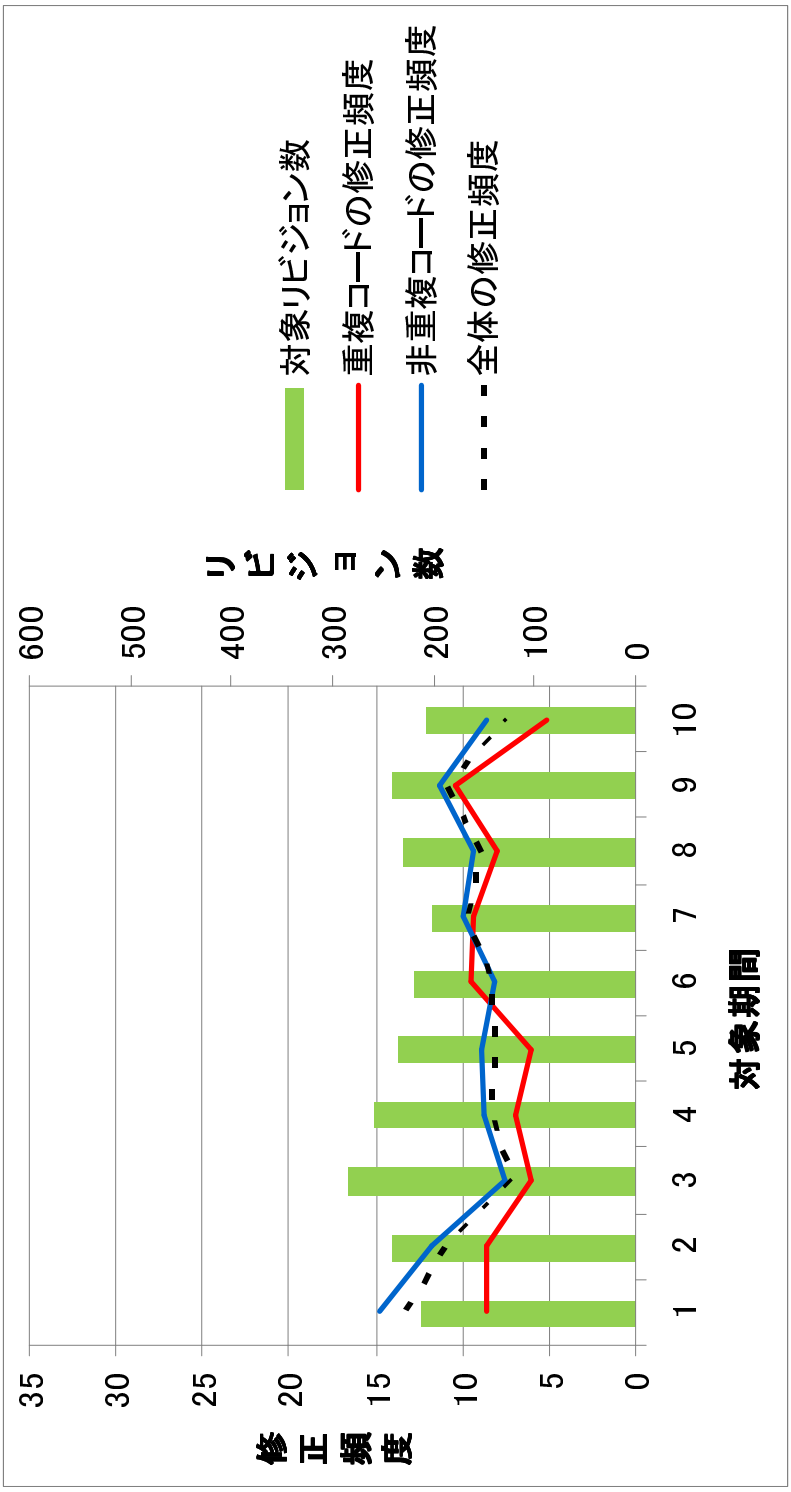


図 7: FileZilla における修正頻度の推移

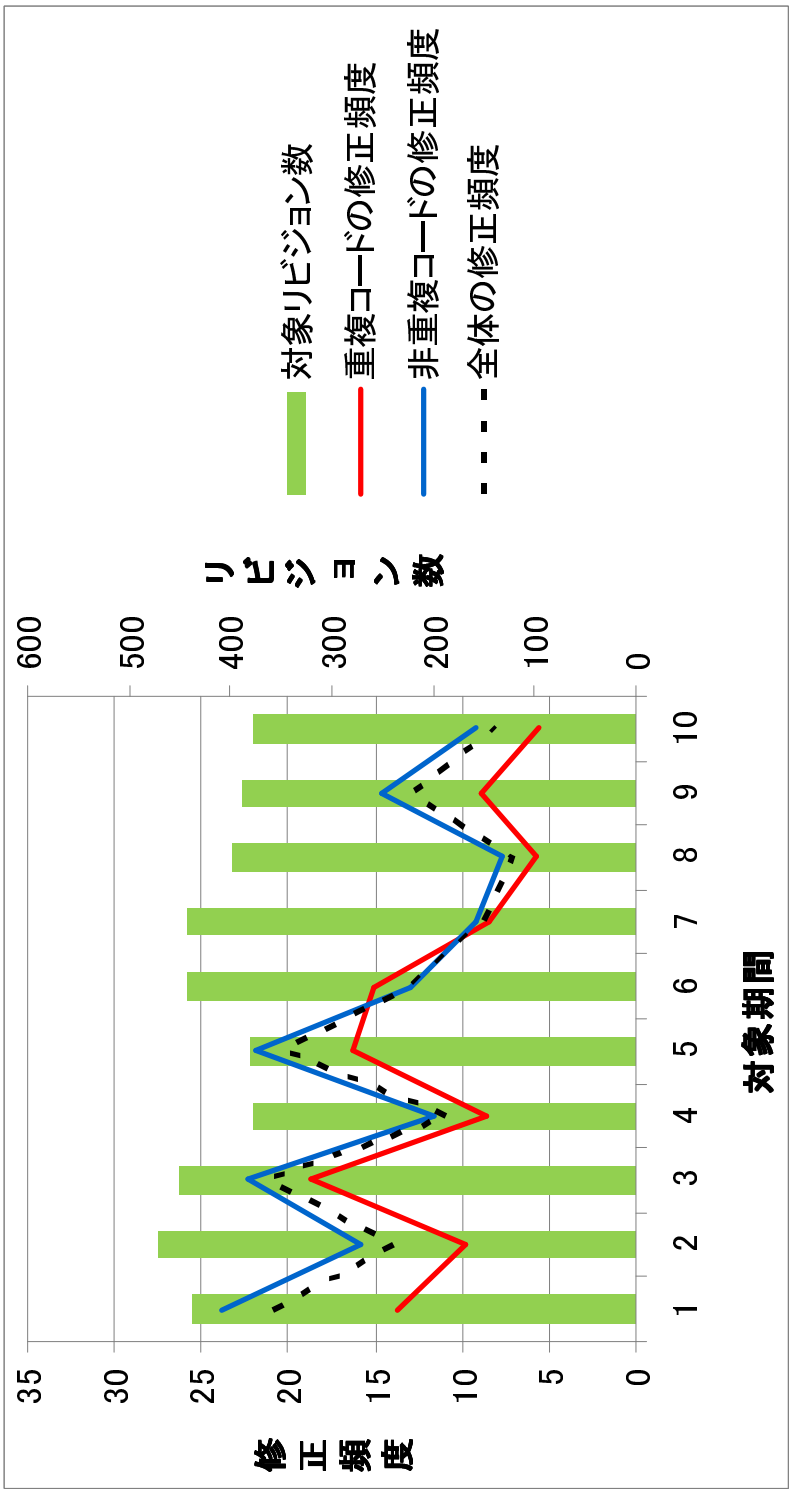


図 8: FreeCol における修正頻度の推移

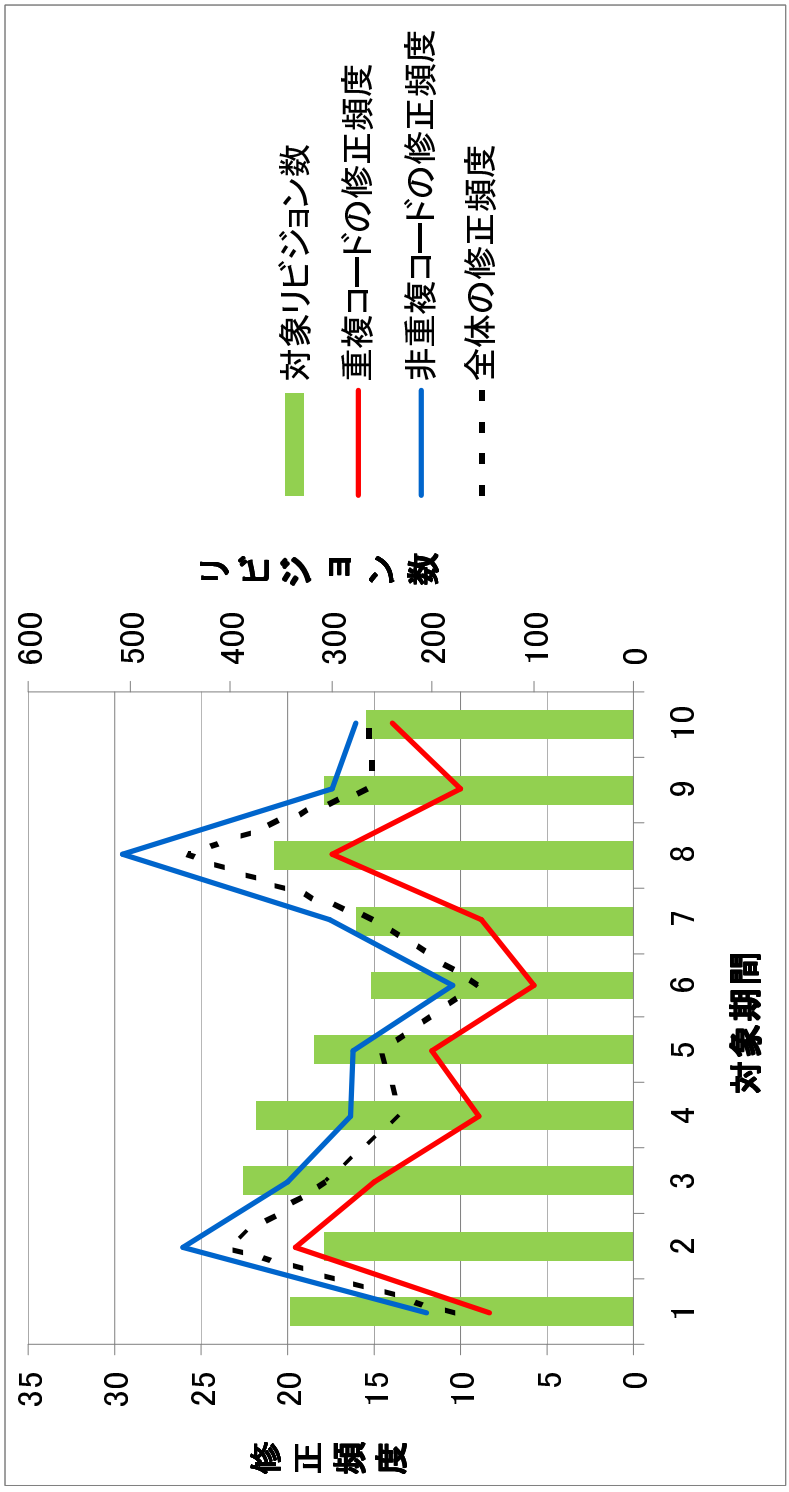


図 9: SQuirreL SQL Client における修正頻度の推移

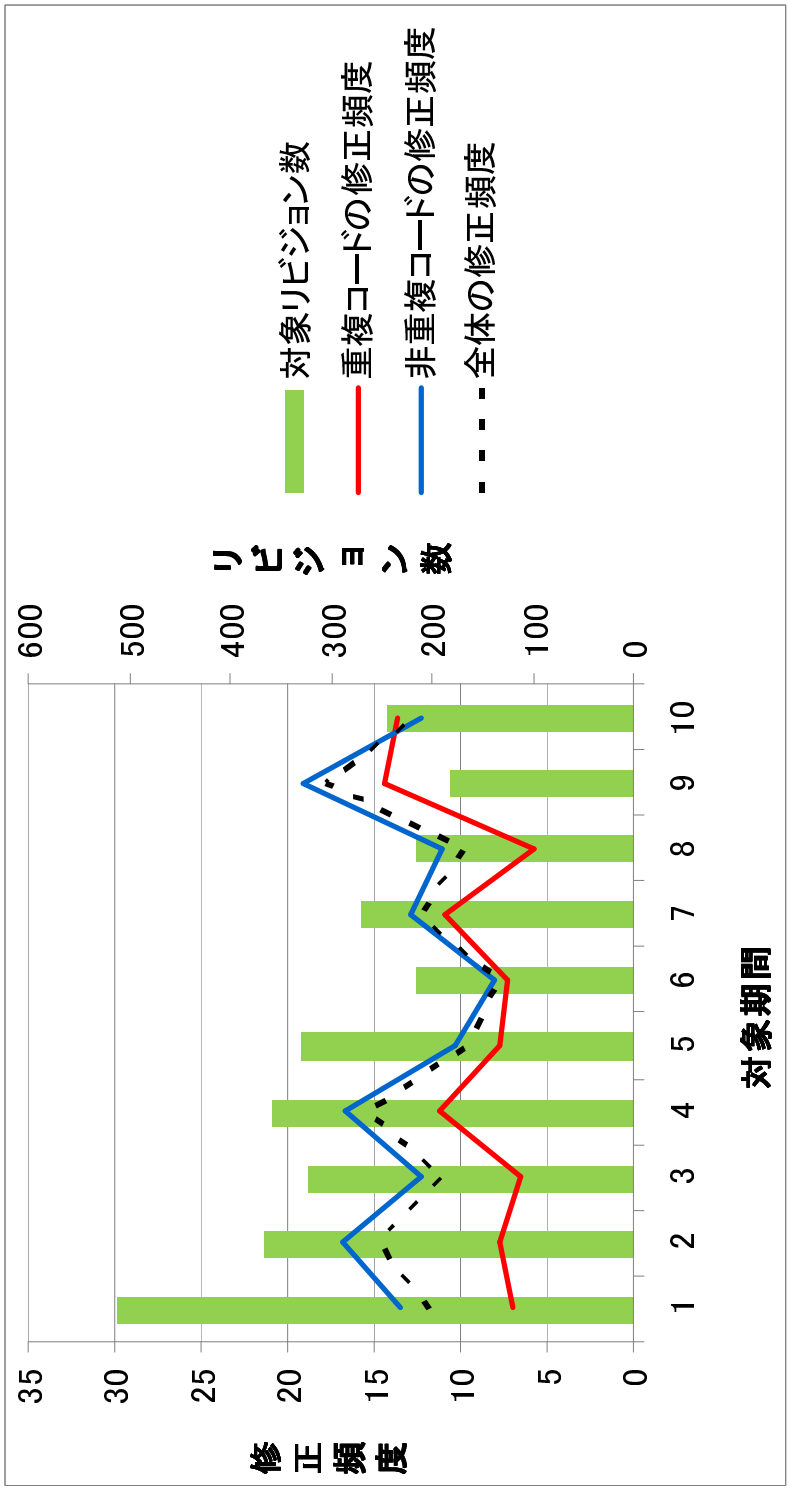


図 10: WinMerge における修正頻度の推移

7　考察

本節では，6.4 節の実験結果について考察していく．

7.1　全体の修正頻度

図 5 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．

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．

7.2　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．

7.3　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．

7.4 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，1 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，1 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，1 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 1 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．逆に，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，1 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 2 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．4.2.2 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．

8　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 5 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

本研究を行うにあたり，理解あるご指導を賜り，常に励まして頂きました 楠本 真二 教授に心から感謝申し上げます.

本研究に関して，有益且つ的確なご助言を頂きました 岡野 浩三 准教授に深く感謝申し上げます.

本研究の全過程を通し，終始熱心かつ丁寧なご指導を頂きました 肥後 芳樹 助教に深く感謝申し上げます.

本研究において，多大なるご助言を頂きました 同 柿元 健 特任助教に深く感謝申し上げます.

本研究に関して，多大なるご協力を頂きました 大阪大学基礎工学部情報科学科 4 年 堀田 圭佑氏に感謝致します.

その他の楠本研究室の皆様のご協力に心より感謝致します.

また，本研究に至るまでに，講義，演習，実験等でお世話になりました情報科学科の諸先生方にこの場を借りて心から御礼申し上げます.

参考文献

[1] 肥後芳樹, 楠本真二, 井上克郎. 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, Vol. J91-D, No. 6, pp. 1465–1481, June. 2008.

[2] 門田暁人, 佐藤慎一, 神谷年洋, 松本健一. 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, Vol. 44, No. 8, pp. 2178–2188, Aug. 2003.

[3] Angela Lozano and Michel Wermelinger. Assessing the effect of clones on changeability. *International Conference on Software Maintenance*, pp. 227–236, Sep. 2008.

[4] CloneTracker. http://mcs.open.ac.uk/alr242/CloneTracker.htm.

[5] Angela Lozano, Michel Wermelinger, and Bashar Nuseibeh. Evaluating the harmfulness of cloning: a change based experiment. *Proceedings of the Fourth International Workshop on Mining Software Repositories*, p. 18, May. 2007.

[6] Angela Lozano, Michel Wermelinger, and Bashar Nuseibeh. Evaluating the relation between changeability decay and the characteristics of clones and methods. *Proceedings of the workshops of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, pp. 100–109, Sep. 2008.

[7] Jens Krinke. Is cloned code more stable than non-cloned code? *Proceedings Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*, pp. 57–66, Sep. 2008.

[8] Stephen G. Eick, Todd L. Graves, Alan F. Karr, J. Steve Marron, and Audris Mockus. Does code decay? assessing the evidence from change management data. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, Vol. 27, No. 1, pp. 1–12, Jan. 2001.

[9] Nils Göde. Evolution of type-1 clones. *Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, pp. 77–86, Sep. 2009.

[10] 井上克郎, 神谷年洋, 楠本真二. コードクローン検出法. コンピュータソフトウェア, Vol. 18, No. 5, pp. 47–54, Sep. 2001.

[11] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. CCFinder: A multi-linguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, Vol. 28, No. 7, pp. 654–670, July. 2002.

[12] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. *Proc. of International Conference on Software Maintenance 98*, pp. 368–377, Mar. 1998.

[13] Shinji Uchida, Akito Monden, Naoki Ohsugi, Toshihiro Kamiya, Ken ichi Matsumoto, and Hideo Kudo. Software analysis by code clones in open source software. *Journal of Computer Information Systems*, Vol. XLV, No. 3, pp. 1–11, Apr. 2005.

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

[15] Sourceforge.net: Find and develop open source software. http://sourceforge.net/.