特別研究報告

題目

ソースコード修正時におけるコードクローンの影響に関する調査 -複数の検出ツールを用いて-

指導教員
楠本 真二 教授

報告者
堀田 圭佑

平成 22 年 2 月 16 日

大阪大学 基礎工学部 情報科学科

平成 21 年度 特別研究報告

ソースコード修正時におけるコードクローンの影響に関する調査
-複数の検出ツールを用いて-

堀田 圭佑

内容梗概

コードクローン (ソースコード中の同一，あるいは類似するコード片) の存在は，あるコード片に修正すべき箇所が見つかった場合，そのコード片とコードクローン関係にあるすべてのコード片について同様の修正を検討しなければならないため，ソフトウェアの修正に要する作業量を増大させるおそれがあると考えられている．しかし，実際にコードクローンがソフトウェアの修正に要する作業量をどの程度増大させているのかを定量的に調査した研究は少ない．また，既存の研究は特定のコードクローン検出ツールのみを用いて調査を行っている．しかし，コードクローンに厳密で普遍的な定義が存在しないため，ある検出ツールではコードクローンとして検出されたコード片が，別の検出ツールではコードクローンとして検出されない可能性がある．このため，ある検出ツールを用いて導かれた結果がコードクローン一般に成り立つとは限らない，という問題点がある．

本研究では既存研究の問題点を改善するため，複数の検出ツールを用いてコードクローンが修正の作業量に与える影響を調査した．修正の作業量の基準として，他のコードとコードクローン関係になっている部分とコードクローン関係になっていない部分にどの程度の頻度で修正が加えられるか，という点に着目した．もし，コードクローンが修正の作業量を増大させているのであれば，より高い頻度で修正が加えられていると考えられる．

いくつかのオープンソースソフトウェアを対象に調査を行ったところ，コードクローンに加えられる修正の頻度はコードクローン関係になっていない部分に加えられる修正の頻度よりやや低いが，統計的に有意な差はみられないという結果を得た．

主な用語

コードクローン
ソフトウェア保守
バージョン管理システム

目 次

1　まえがき

近年，ソフトウェアの大規模化，複雑化に伴い，ソフトウェアの保守に要するコストが増加している．ソフトウェアの保守を困難にさせる要因の 1 つとしてコードクローンへの関心が高まっており，これまでにコードクローンに関する様々な研究が盛んに行われている [1].

コードクローンとは，ソースコード中に存在する同一，あるいは類似したコード片のことであり，主にコピーアンドペーストの操作等によって発生するといわれている．あるコード片に修正すべき箇所が見つかった場合，そのコード片とコードクローン関係にある他のすべてのコード片に対して同様の修正の是非を検討しなければならない．このため，コードクローンの存在はソフトウェアの修正に要する作業量を増大させるおそれがあると考えられている．

この問題に対処するため，コードクローンの情報を開発者が把握することは有益であると考えられているが，人間が全てのコードクローンを認識しておくことは現実的ではないため，これまでにコードクローンを自動的に検出する様々な手法が提案されている．またそれらの手法を実装したコードクローンを自動的に検出するツールも多数開発されている [1][2][3][4][5][6]. しかし，コードクローンに厳密で普遍的な定義が存在しないため，検出ツールによって異なるコードクローンの定義が用いられている．よって，ある検出ツールではコードクローンとして検出されたコード片が別の検出ツールを用いた際にはコードクローンとして検出されないという可能性がある．

コードクローンの検出，集約等に関する研究は盛んに行われているが，コードクローンがソフトウェアの修正に要する作業量に対して実際にどの程度影響を与えているのかを定量的に調査した研究は少ない．さらに既存研究 [7][8][9] には，特定の検出ツールのみを用いて調査を行っている，という問題点がある. 上述の通り，検出ツールによって検出されるコードクローンに差があるため，ある検出ツールを用いて導かれた結果が，別の検出ツールを用いた時も同様に導かれるとは限らない．

そこで本研究ではより一般的な結果を得るため，複数の検出ツールを用いて，コードクローンがソフトウェアの修正の作業量にどの程度影響を与えているのかを定量的に調査する. また本研究では，修正の作業量を計測するための基準として，ソースコードに加えられる修正の頻度に着目する．バージョン管理システム *Subversion*[10] で管理されているソフトウェアの開発履歴情報を元に，過去の全ての開発期間において，1 リビジョンあたりのコードクローンに加えられた修正の箇所数と，コードクローン以外の部分に加えられた修正の箇所数をそれぞれ計測し，比較する．もしコードクローンが修正の作業量を増大させているのであれば，コードクローンへの修正頻度がコードクローン以外の部分への修正頻度よりも高くなると考えられる．

いくつかのオープンソースソフトウェアに対して，*CCFinder*[2]，*CCFinderX*[3]，*Simian*[4]，*Scorpio*[5] の 4 種類の検出ツールを用いて計測を行った．その結果，いずれの検出ツールを用いた場合でも，コードクローンへの修正頻度とコードクローン以外の部分への修正頻度では，コードクローンへの修正頻度のほうがやや低いが統計的に有意な差はみられない，という結果を得た．

以降 2 節では，コードクローン，本研究に用いた検出ツール，バージョン管理システム *Subversion*，及びソースコード正規化ツール *CommentRemover*[11] について述べる．3 節では修正頻度の計測方法について説明する．4 節では，オープンソースソフトウェアを対象に修正頻度を計測した計測結果について述べ，その考察を 5 節で行う．6 節では本研究で得られた結果の妥当性について述べる．7 節で関連研究について述べ，最後に 8 節で本研究のまとめと今後の課題について述べる．

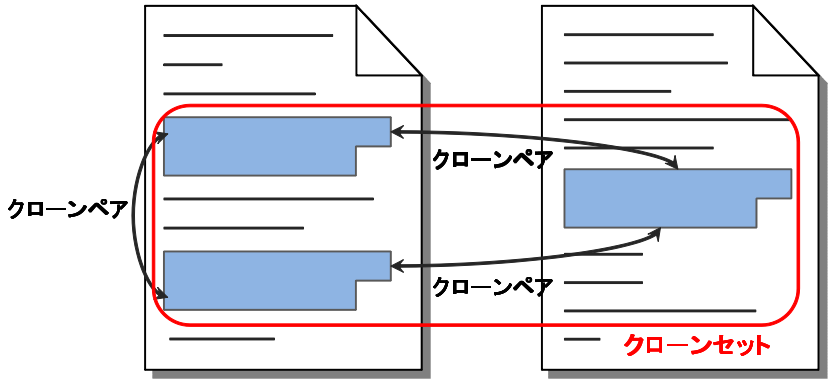


図 1: クローンペアとクローンセット

2 準備

2.1 コードクローン

2.1.1 定義

コードクローンとはソースコード中に存在する同一，あるいは類似するコード片のことである．図 1 に示すように，ソースコード中に存在する 2 つのコード片 α，β が類似しているとき，α と β は互いにクローンであるという．またペア（α，β）をクローンペアと呼ぶ．α，β それぞれを真に包含する如何なるコード片も類似していないとき，α，β を極大クローンと呼ぶ．また，互いにクローンであるコード片を同値としたときの同値類をクローンセットと呼ぶ [12]．

ただし，どのような基準で類似していると判断するかは検出手法や検出ツールによって異なる．

また，コードクローン間の類似の度合に基づきコードクローンを次の 3 つのタイプに分類することができる [13][14]．

Type-1

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に依存する箇所を除いて，

完全に一致するコードクローン.

Type-2

変数名や関数名などのユーザ定義名，また変数の型などの一部の予約語のみが異なるコードクローン.

Type-3

Type-2 における変更に加えて，文の挿入や削除，変更が行われたコードクローン.

2.1.2 発生の原因

コードクローン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挙げられる [2][15][16].

既存コードのコピーアンド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ることにより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，現実にはコピーアンド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．コピーアンドペーストによって生成されたコード片は，コピー元のコード片とコードクローン関係になる.

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，所得税の計算や，キューの挿入処理，データ構造アクセス処理などである.

適切な機能の欠如

抽象データ型やローカル変数を用いることができないプログラミング言語を開発に用いている場合，同じようなアルゴリズムを用い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よって生成されるコードは，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．このため，類似した処理を目的としたコードを生成した場合，識別子名等の違いを除き，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 や CPU に対応したソフトウェアは，各プラットフォームを対象に生成された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，大きなコードクローンになる可能性は低い．

2.2 コードクローン検出ツール

2.2.1 コードクローン検出ツールの分類

コードクローンを検出する手法はこれまでに多数提案されている．またそれらを実装した，コードクローンを自動的に検出するツールも多数開発されている．これらの検出技術はコードクローンをどの単位で検出するかによって，大まかに以下の 5 つに分類することができる [1].

行単位の検出

行単位の検出は，ソースコードを行単位で比較してコードクローンを検出する手法であり，閾値以上連続して一致する行をコードクローンとして検出する．他の手法と異なり，ソースコードに対する事前処理を必要としない．このため，他の手法と比べて高速にコードクローンを検出可能である．しかし，同じ処理を行っているコードであっても，例えば長い行を複数行に分割した場合と分割しなかった場合など，コーディングスタイルが違う場合はコードクローンとして検出できないという弱点を持つ．

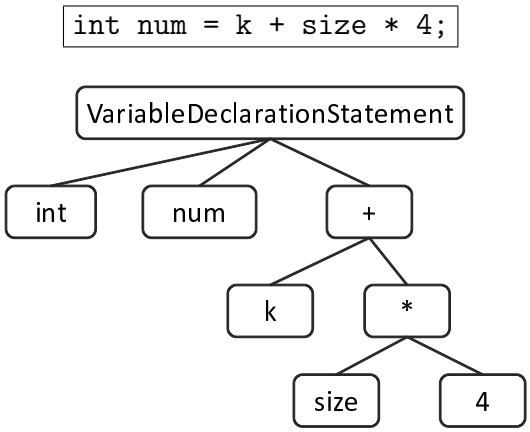


図 2: 抽象構文木の例

字句単位の検出

字句単位の検出は，ソースコードを字句単位に分割し，閾値以上連続して一致する字句の部分列をコードクローンとして検出する手法である．行単位の検出と異なり，コーディングスタイルのみ違う場合などもコードクローンとして検出することが可能である．ソースコードを検出用の中間表現に変換する必要がないため，高速にコードクローン検出を行うことができるという利点もある．また，字句に事前処理を行うことで変数名などのユーザ定義名のみ異なるコードクローンなども検出可能となる．

抽象構文木を用いた検出

抽象構文木 (図 2) を用いた検出は，ソースコードに対して構文解析を行い，抽象構文木を構築した後，その抽象構文木を用いてコードクローンを検出する手法であり，抽象構文木上の同形の部分木がコードクローンとして検出される．抽象構文木を構築するという事前処理を要するため，行単位の検出や字句単位の検出と比べ，時間的，空間的コストが高くなるという欠点がある．ある関数定義の終わりから次の関数定義の先頭までの類似部分など，プログラムの構造を無視したコードクローンを検出しないという特徴を持つ．

プログラム依存グラフを用いた検出

プログラム依存グラフ (図 3，[17]) を用いた検出は，ソースコードに対して意味解析を行い，ソースコードの要素間の依存関係を表すプログラム依存グラフを構築した後，そのプログラム依存グラフを用いてコードクローン検出を行う手法である．プログラム依存グラフ

```
1: void sample() {
2:   for ( int i = 0 ; i < 10 ; i++ ) {
3:     System.out.println(i);
4:   }
5: }
```

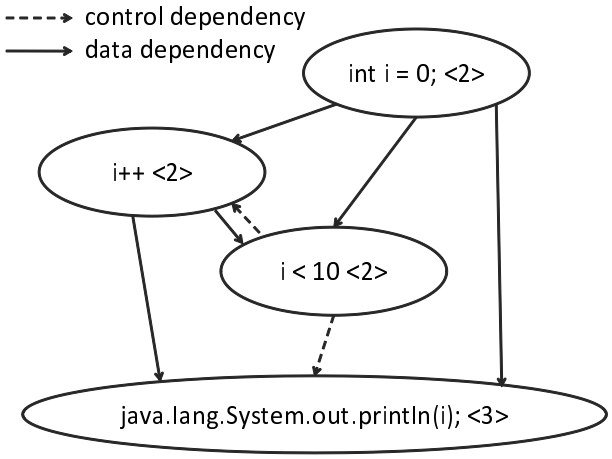


図 3: プログラム依存グラフの例

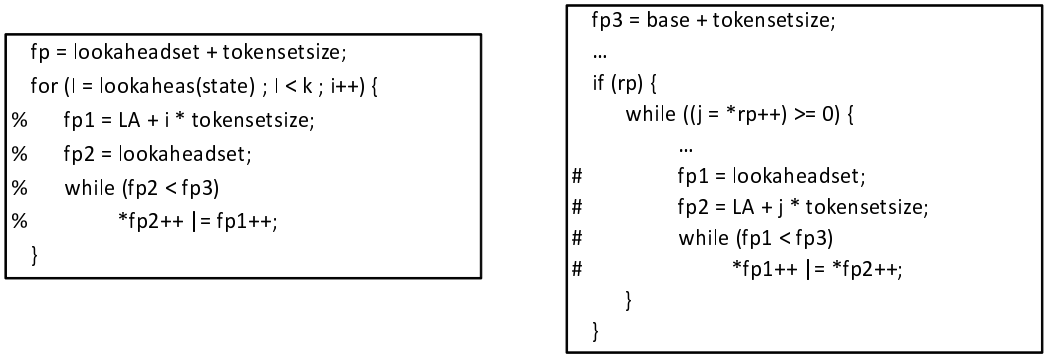
上の同形部分がコードクローンとして検出される．抽象構文木を用いた検出と同様に事前処理を必要とするため，時間的，空間的コストが高くなるという欠点を持つ．ソースコードの順番が入れ替わっていても意味的に同一であるコードクローン (順序入れ替わりコードクローン) などは意味的な処理を考慮しなければ検出できないが，この手法はこれらのコードクローンを検出することができるという点が特徴として挙げられる．

順序入れ替わりコードクローンの例を図 4 に示す．この例の場合，%で表されているコード片と，#で表されているコード片が順序入れ替わりコードクローンとなる．

その他の技術を用いた検出

その他の技術を用いた検出手法として，プログラムのモジュール (ファイル，クラス，メソッドなど) に対してメトリクスを計測し，その値の一致または近似の度合いを検査することによって，そのモジュール単位でのコードクローンを検出する手法であるメトリクスを用いた検出や，プログラムの盗用の検出やプログラムの作者を特定することを目的とした，フィンガープリントやバースマークを用いた検出手法などがある．

以下の小節では，本研究に用いたコードクローン検出ツールについて述べる．



(a) コード片 1　　(b) コード片 2

図 4: 順序入れ替わりコードクローン

2.2.2 *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極大クローンを検出し，それをクローンペアの位置情報として出力する．*CCFinder* は字句単位の検出に分類される．*CCFinder* の持つ主な特徴は次の通りである．

細粒度のコードクローンを検出

検出対象ソースコードに事前処理として字句解析を行うことにより，字句単位でのコードクローンを検出する．

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 [12].

細かい設定が可能

- 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コードクローンは検出可能である．
- コードクローン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，最少一致トークン数を指定することでそのようなコードクローンの検出を防ぐことができる．

ある程度の違いは吸収可能

- ソースコード中に含まれる変数などのユーザ定義名，定数をパラメータ化することで，その違いを吸収することができる．
- 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，その違いを吸収することができる．
- 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することができる．

2.2.3 *CCFinderX*

CCFinderX は *CCFinder* のメジャーバージョンアップであり，*CCFinder* と同様に字句単位の検出に分類される．*CCFinder* からの改良点として以下のような点が挙げられる [3].

- マルチコア CPU 向けにマルチスレッド化.
- 抽象構文木ベースの前処理を行う.
- 検索機能を追加.
- 対応しているすべての言語を可能な限り等しくサポートする.
- 利用者によるプログラミング言語の方言，新しいプログラミング言語への対応を可能にした.
- コードクローンに関するメトリクスを使った分析に対応.
- 他のツールとの連携のために，TSV(タブ区切り形式) でデータを入出力.
- 複数のビューによる対話的な分析が可能.

2.2.4 *Simian*

Simian は，行単位の検出に分類されるコードクローン検出ツールである．*Simian* の特徴として，以下のような点が挙げられる．

様々な言語，及びテキストにも対応

Simian は Java，C#，C++，C，Objective-C，JavaScript(ECMAScript)，COBOL，Ruby，Lisp，SQL，Visual Basic，Groovy に対応している．また，これらの言語以外のファイルについても，テキスト形式のファイルであればテキストファイルとみなして類似する部分の検出を行うことができる．

少ないメモリで，高速に検出

390,309 LOC，ファイル数 4242 のソフトウェアに対して実行した結果，使用メモリ 46MB，実行時間 10 秒以内でコードクローンを検出した [4].

検出対象を細かく設定できる

コメント，空白，import 文，include 文，パッケージ宣言等を無視するように設定することで，実用的でないコードクローンの検出数を軽減することができる．この他に，コードクローンの最小行数の設定や，文字の大文字小文字の違い，変数名の違い，数値の違い等を無視するかどうかなども設定することが可能である．

2.2.5 *Scorpio*

Scorpio はプログラム依存グラフを用いた検出に分類されるコードクローン検出ツールであり，Java に対応している．*Scorpio* の特徴として以下の点が挙げられる [5][17].

非連続コードクローンが検出可能

Scorpio は Type-1，Type-2，Type-3 の全てのタイプのコードクローンを検出することが可能である．またプログラム依存グラフを用いて検出を行うため，行単位の検出手法や字句単位の検出手法では検出できない順序入れ替わりコードクローンなどの非連続コードクローンを検出することが可能である．

グラフ探索方法の併用

プログラム依存グラフをたどって同形部分グラフを検出する際，グラフを探索する方法として，グラフを順方向に探索する方法 (フォワードスライス) と，グラフを逆方向に探索する方法 (バックワードスライス) の 2 種類が存在する．フォワードスライスでは検出できないコードクローンがバックワードスライスで検出できる場合や，バックワードスライスでは検出できないコードクローンがフォワードスライスでは検出できる場合が起こり得るが，*Scorpio* では両方の探索方法を併用するため，双方で共通して検出できるコードクローンに加え，どちらか一方の探索方法でしか検出できないコードクローンを検出することが可能である．

実用的な時間で検出可能

プログラム依存グラフを用いた検出ではソースコードの意味解析を行いプログラム依存グラフを構築するという事前処理を必要とするため，他の検出手法と比べて検出に要するコス

トが高いという欠点がある．*Scorpio* では，プログラム依存グラフを探索する際に基点として用いる頂点に制限を設けることで，検出コストの削減を行っている．

2.3 バージョン管理システム

バージョン管理システムとは，主にプログラム開発においてソースコードやその他のデータを管理するために用いるシステムである．バージョン管理システムは主に開発情報 (ソースコード，リソースなど) を開発者間で共有する機能，及び開発履歴情報を保持する機能を提供する．バージョン管理システムを用いることで離れた場所にいる開発者間で開発情報の共有が容易になるという利点があるため，商用ソフトウェア開発やオープンソースソフトウェアの開発など，多数の開発者によってソフトウェア開発が行われる際に一般的に使用されている．

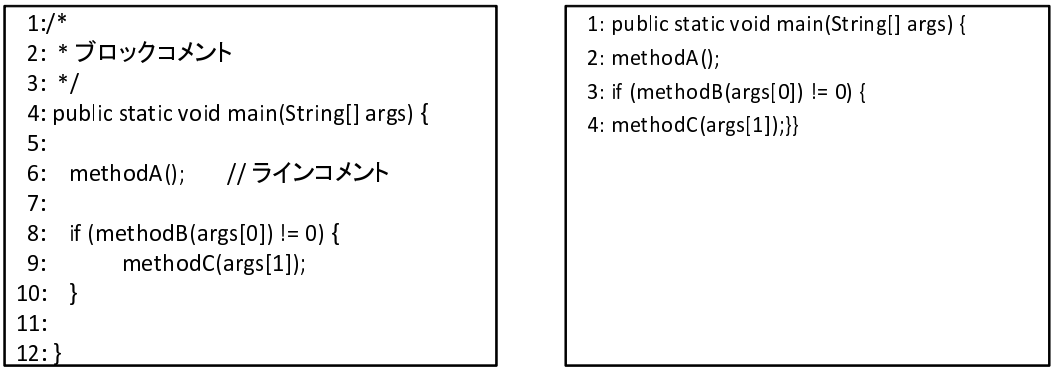
本研究で用いたバージョン管理システム *Subversion*[10] における上述の機能の概要は以下の通りである．

開発情報の共有

Subversion では，ソフトウェアの開発情報 (ソースコード，リソースなど) はリポジトリと呼ばれるデータ格納庫に保存される．開発者はソフトウェアの開発を行う際，リポジトリから開発情報を手元にコピーし (この動作をチェックアウトと呼ぶ)，修正，変更を加えた後，リポジトリに反映させる (この動作をコミット，あるいはチェックインと呼ぶ)．コミットしたとき，別の開発者が同じファイルに変更を加えていた場合は，ファイルが別の開発者によって変更されていることが通知されるので，開発者は 2 つのファイルをマージし，コミットする．これにより別の開発者が行った変更を誤って上書きすることを防ぐことができる．

開発履歴情報の保持

Subversion では，開発履歴情報をリビジョンと呼ばれる単位で保持している．リビジョンとは開発の状態を表す単位で，リポジトリがコミットを受け付けるたびに生成され，それぞれのリビジョンにはユニークな自然数 (リビジョン番号) が割り当てられる．開発履歴情報にはソースコードなどの開発情報の他，変更を加えた開発者名や日時，変更が加えられたファイル名，変更を加えた開発者の変更に関するメッセージ等のログも含まれている．開発履歴情報を利用すれば，ソースコードに誤った変更を加えてしまった場合などに過去の状態に復元することが可能となる．



(a) 正規化前　　(b) 正規化後

図 5: ソースコードの正規化

2.4 ソースコード正規化ツール *CommentRemover*

CommentRemover は，ソースコードの正規化を行うツールであり，Java，C#，C/C++ に対応している．*CommentRemover* により下記の正規化を行うことができる．

- 空白行を削除.
- ブロックコメント (/* ··· */) を削除.
- ラインコメント (// ···) を削除.
- 中括弧 ({ }) のみの行を削除し，1 つ上の行に追加.
- インデントを削除.

コードクローンの前後にあるコメント行は，コードクローンに関するものであってもコードクローン以外の部分とみなされる．同様にコードクローンの前後に存在する空白行もコードクローン以外の部分とみなされる．このような行に対する変更を計測対象に含めると，コードクローン以外への修正頻度が実際よりも高い値となるおそれがある．

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存在する．しかし，このような変更はソースコードの意味的な内容と本質的な関わりを持たないため，これらを計測に含めるのは適切ではないと考えられる．これらの問題を解決するために，本研究では，*CommentRemover* を用いて上記の正規化を行い，正規化後のソースコードに対して計測を行う．図 5 に，*CommentRemover* を用いてソースコードの正規化を行った例を示す.

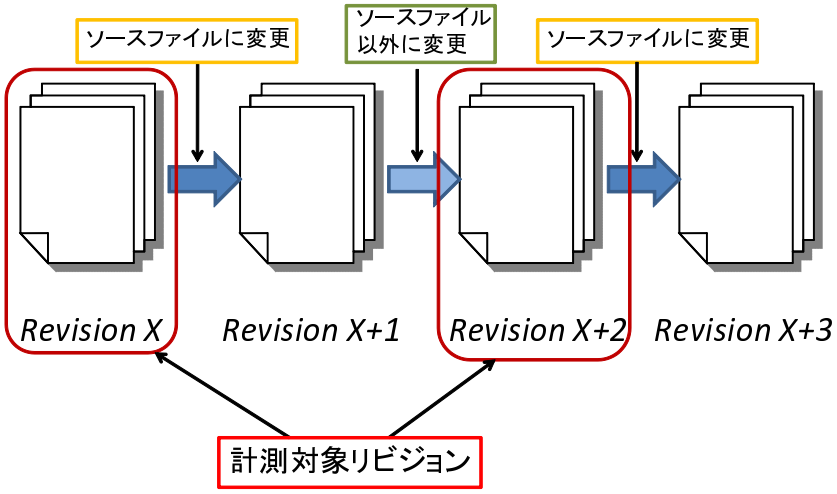


図 6: 計測対象リビジョン

3 修正頻度の計測方法

本節では修正頻度の計測方法について述べる．

3.1 計測対象リビジョン

全リビジョンのうち，ソースファイルに変更が加えられたリビジョンを計測対象リビジョンと呼ぶ．図 6 の例の場合，リビジョン X とリビジョン $X+2$ が計測対象リビジョンとなる．

3.2 修正箇所数

本研究では修正頻度の計測に修正箇所数を用いる．修正箇所数の計測には外部ツール *diff-win*[18] を用いる．*diff-win* は，2 つのファイルの差分を計算し，連続する行に関する差分については 1 つに結合して出力する．*diff-win* の出力例を図 7 に示す．この例の場合，変更前のファイルの 58 行目から 71 行目が変更後のファイルの 61 行目から 81 行目に変更されていることを示している．本研究ではこの *diff-win* の出力をもとに修正箇所数を計測する．

変更箇所 (前述の例の場合 58 行目から 71 行目) のすべての行がコードクローンの場合コードクローンへの修正とみなし，すべての行がコードクローンでない場合コードクローン以外

```
58,71c61,81
<public void show() {
<display.setCurrent(list);}
(中略)
<private String intToTwoDigit(int i) {
<return ((i < 10) ?  "0" :  "") + i; }}
---
>public void show(MIDlet m) {
>this.parent = m;
(中略)
>this.display.setCurrent(alert);}}
```

図 7: *diff-win* の出力例

への修正とみなす．また変更箇所にコードクローンとコードクローン以外の部分の両方が含まれる場合，コードクローンとコードクローン以外へそれぞれ 1 度ずつ修正が加わったものとみなす．

なお，既存の研究 [7][8][9] では修正が加えられた行数を用いて計測を行っている．しかしこの計測手法では，例えば 1 行の変更が 10 箇所に加えられた場合と，10 行の変更が 1 箇所に加えられた場合の計測結果が等しくなる．ソースコードに修正を加える場合，その作業に要する作業量の大部分は，修正すべき箇所の特定など，実際の修正作業の前段階が占めると考えられる．上述の例の場合，1 行の変更が 10 箇所にある場合と 10 行の変更が 1 箇所にある場合では，1 行の変更が 10 箇所にある場合の方がより修正に要する作業量は多いと考えられる．本研究ではこの考えに基づき，修正箇所数に基づいた修正頻度を計測する．

3.3 修正頻度

計測対象リビジョンの集合を R とし，ある計測対象リビジョン $r \in R$ のソースコードの行数を $l(r)$，コードクローン関係にある行数を $lc(r)$，コードクローン関係にない行数を $ln(r)$ とする．また，r について，コードクローンに加えられた修正箇所数を $mc(r)$，コードクローン以外に加えられた修正箇所数を $mn(r)$ とする．

このとき，次式で算出される値を**コードクローンへの修正頻度**と定義する．

$$\frac{\sum_{r \in R} mc(r)}{|R|} \cdot \frac{\sum_{r \in R} l(r)}{\sum_{r \in R} lc(r)}$$

同様に，次式で算出される値を**コードクローン以外の部分への修正頻度**と定義する．

$$\frac{\sum_{r \in R} mn(r)}{|R|} \cdot \frac{\sum_{r \in R} l(r)}{\sum_{r \in R} ln(r)}$$

これらの値は，1 リビジョンあたりのコードクローンに加えられた修正箇所数，及びコードクローン以外に加えられた修正箇所数に，コードクローンが占める行数の割合 (いずれかのソースファイルとコードクローン関係にある行数の割合)$\sum_{r \in R} lc(r) / \sum_{r \in R} l(r)$，及びコードクローン以外の部分が占める行数の割合 $\sum_{r \in R} ln(r) / \sum_{r \in R} l(r)$ の逆数をかけあわせて正規化した値である．

リビジョン r のコードクローンが占める行数の割合と $mc(r)$ の間には強い正の相関がある．すなわち，コードクローンが占める行数の割合が高ければ，コードクローンに加えられる修正箇所数は大きくなる．同様に，リビジョン r のコードクローン以外の部分が占める行数の割合と $mn(r)$ の間には強い正の相関がある．このため，正規化を行う前の値は，コードクローンが占める行数の割合とコードクローン以外の部分が占める行数の割合の影響を強く受けると考えられる．したがって，単純に 2 つの値の大小を比較することができない．

そこで，2 つの計測値を直接比較するため，コードクローンが占める行数の割合とコードクローン以外の部分が占める行数の割合を用いて上述の正規化を行う．

3.4 計測手順

修正頻度の計測方法の概要は以下の通りである．

1. *Subversion* の開発履歴情報を用いて計測対象リビジョンを特定し，それぞれの計測対象リビジョンにおいてどのファイルが変更されたのかを特定する．
2. 計測対象リビジョンのソースファイルをリポジトリから取得する．
3. ソースファイルに対し *CommentRemover* を使用して，ソースコードを正規化する．
4. 正規化したソースファイルに対してコードクローン検出ツールを適用し，いずれかのコードとコードクローン関係にある行を特定する．
5. 計測対象リビジョンのソースファイルとその次のリビジョンのソースファイルとの差分を取り，それぞれの計測対象リビジョンにおいてどのファイルのどの部分が変更されたのかを特定する．
6. 4. と 5. で得た情報を照合し，コードクローンに加えられた修正箇所数とコードクローン以外に加えられた修正箇所数をそれぞれ計測する．

4 計測

4.1 計測対象

本研究では *SourceForge*[19] で公開されているオープンソースソフトウェアの中から，11 のソフトウェアを対象に計測を行った．対象としたソフトウェアを表 1 に示す．これらのソフトウェアを選択した基準は以下の通りである．

1. バージョン管理システム *Subversion* を用いて開発を行っている．
2. 開発に Java もしくは C，C++を用いている．
3. リビジョン数が 10〜500 程度である．

ただし，Squirrel-SQL に関しては開発期間の長いソフトウェアに対する計測を目的に選択したため，リビジョン数が上記の範囲内に収まっていない．また，開発に Java，C，C++以外のプログラミング言語を用いているソフトウェアの場合，これらの言語以外のソースファイルは計測対象としない．すなわち，これらのプログラミング言語以外のソースファイルのみに修正が加えられたリビジョンは計測対象リビジョンには含めない．

次に，それぞれのソフトウェアに対して 4 種類の検出ツールを適用して計測した，コードクローン含有率 (コードクローン行数/総行数) を表 2 に示す．なお，*Scorpio* が Java のみに対応しているため，C，C++ソフトウェアに対しては 3 種類の検出ツールを用いて計測している．またこのコードクローン含有率は，全計測対象リビジョンのコードクローン行数の和を全計測対象リビジョンの総行数の和で除算して算出した値である．また，Squirrel-SQL の *Scorpio* を用いた計測については，*Scorpio* によるコードクローン検出に大きな時間を要したため処理を中断した．

表 1: 計測対象

プロジェクト名	言語	総リビジョン数	ソースファイル数 (最新リビジョン)	総行数 (最新リビジョン)
ThreeCAM	Java	14	40	3,854
DatabaseToUML	Java	59	100	19,695
AdServerBeans	Java	98	103	7,406
NatMonitor	Java	128	8	1,139
OpenYMSG	Java	141	865	130,072
Squirrel-SQL	Java	5,351	3,246	481,368
QMailAdmin	C	312	579	173,688
Tritonn	C,C++	100	92	45,368
Newsstar	C	165	1,018	192,716
Hamachi-GUI	C	190	420	65,790
GameScanner	C,C++	420	2,163	1,214,570

表 2: コードクローン含有率

プロジェクト名	コードクローン含有率 (%)			
	CCFinder	*CCFinderX*	Simian	Scorpio
ThreeCAM	29.82	10.49	4.09	26.17
DatabaseToUML	21.35	25.12	7.63	11.80
AdServerBeans	22.69	18.24	20.25	15.89
NatMonitor	9.04	7.70	0.66	6.63
OpenYMSG	17.37	9.94	5.76	9.94
Squirrel-SQL	28.92	33.43	31.18	—
QMailAdmin	34.28	19.59	8.81	—
Tritonn	13.84	7.46	5.48	—
Newsstar	7.88	4.80	1.52	—
Hamachi-GUI	36.52	23.05	18.51	—
GameScanner	23.06	13.14	6.57	—

4.2 計測結果

本節では上述のオープンソースソフトウェアに対して修正頻度の計測を行った結果を述べる．なお計測結果を示した図において，検出ツール名を表 3 のように略記する．

4.2.1 プログラミング言語別の計測結果

Java を使用しているソフトウェアに対して計測を行った結果を図 8 に，C，C++を使用しているソフトウェアに対して計測を行った結果を図 9 にそれぞれ示す．なお，*Scorpio* が Java のみに対応しているため，C，C++を使用しているソフトウェアに対する計測は 3 種類のツールを用いている．また Squirrel-SQL における *Scorpio* を用いた計測に関しては，途中で処理を中断したため未計測となっている．図の縦軸は計測した修正頻度，横軸は計測対象ソフトウェア及び計測時に使用した検出ツールをそれぞれ表している．全 38 のデータのうち 25 のデータにおいて，コードクローンへの修正頻度がコードクローンでない部分への修正頻度よりも低い，という結果が得られた．また．全 11 のソフトウェアのうち，6 つのソフトウェアについてはいずれの検出ツールを用いた場合でもコードクローンへの修正頻度の方が低く，3 つのソフトウェアについては検出ツールによってコードクローンへの修正頻度が低い場合とコードクローンへの修正頻度が高い場合が存在している．

プログラミング言語ごとの修正頻度の平均値，および全体の修正頻度の平均値を表 4 に示す．いずれの場合においても，コードクローンへの修正頻度は，コードクローン以外への修正頻度より低い，という結果が得られた．

表 3: 検出ツール名の略記

検出ツール名	略記
CCFinder	**C**
CCFinderX	**X**
Simian	**Si**
Scorpio	**Sc**

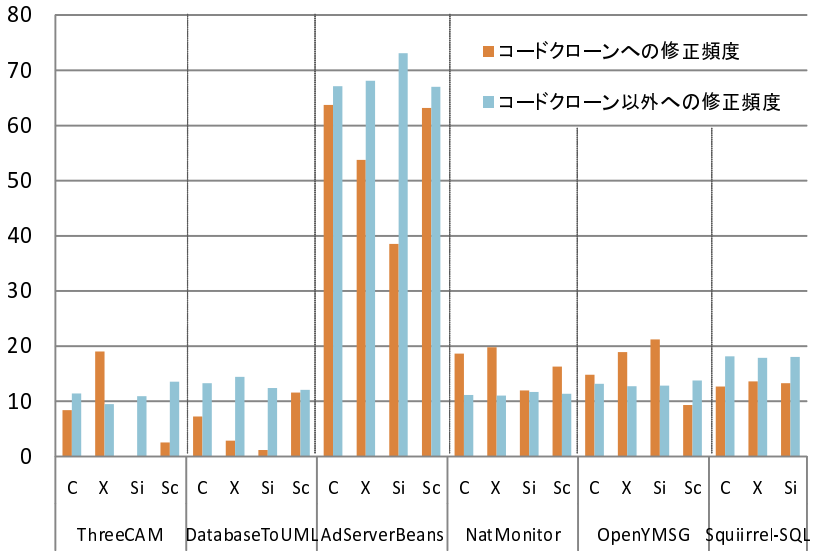


図 8: 計測結果 -Java-

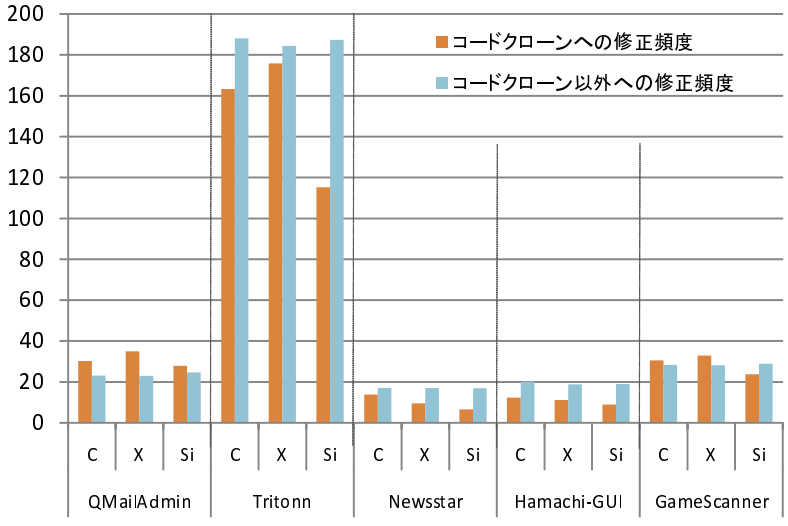


図 9: 計測結果 -C,C++-

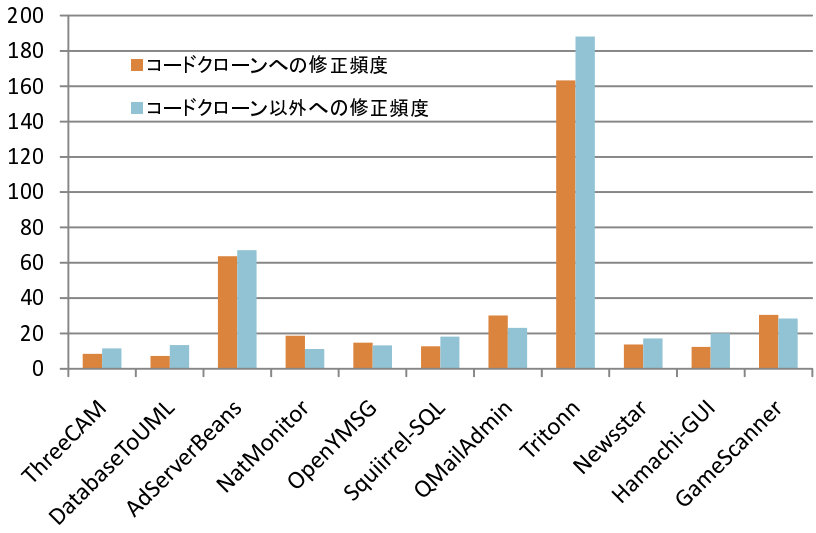


図 10: 計測結果 -*CCFinder*-

4.2.2 検出ツール別の計測結果

計測結果を検出ツール別にグラフ化した図を図 10，図 11，図 12，図 13 にそれぞれ示す．

次に，検出ツールごとのコードクローンへの修正頻度の平均値，およびコードクローン以外への修正頻度の平均値を表 5 に示す．

いずれの検出ツールを用いた場合でも，コードクローンへの修正頻度の平均値はコードクローン以外への修正頻度の平均値より低い，という結果が得られた．また *Simian* を用いた結果において，他の検出ツールを用いた結果よりも 2 つの値の平均値の差が大きくなった．

表 4: 言語ごとの平均値

言語	コードクローン	コードクローン以外
Java	19.2463	22.8141
C,C++	46.4531	55.0157
全体	29.9858	35.5253

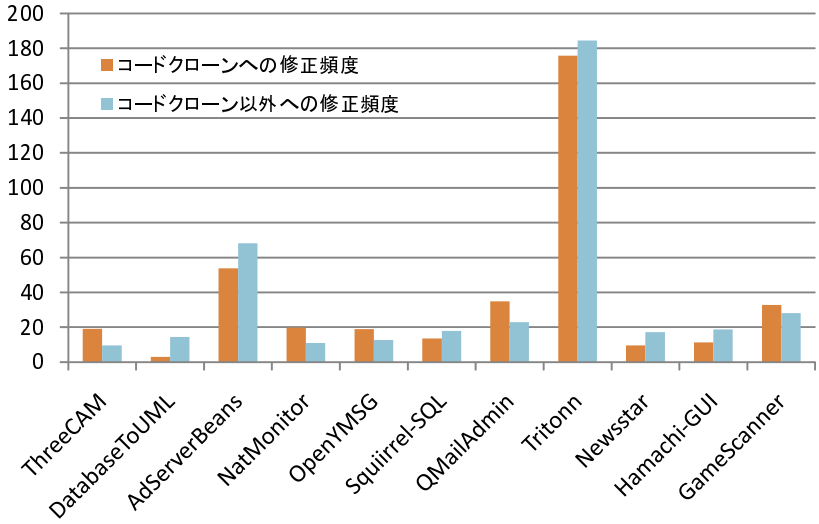


図 11: 計測結果 -*CCFinderX*-

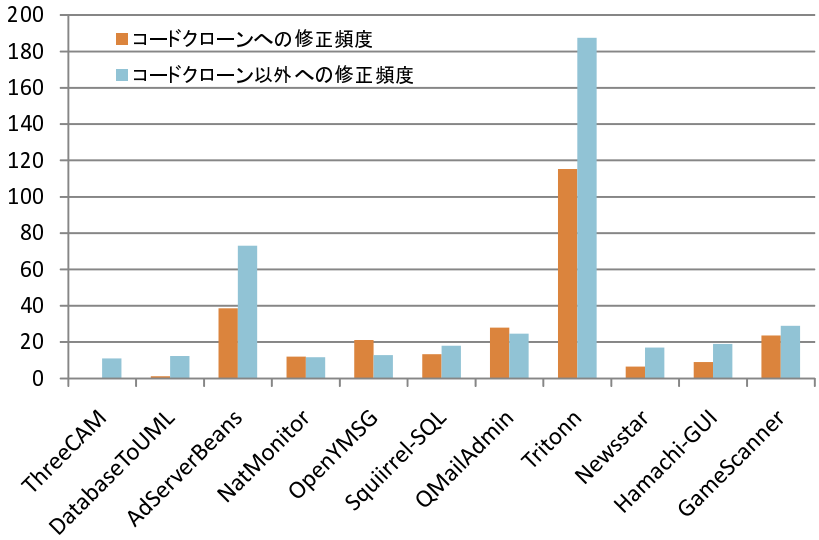


図 12: 計測結果 -*Simian*-

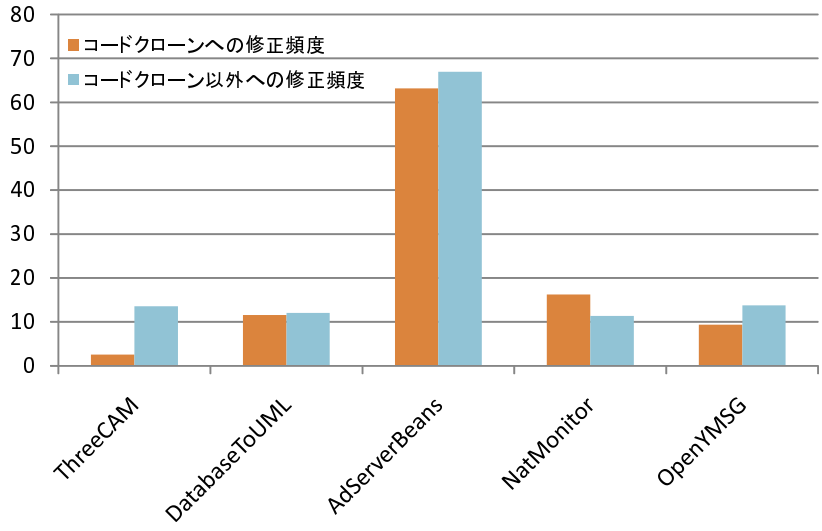


図 13: 計測結果 -*Scorpio*-

表 5: 検出ツールごとの平均値

検出ツール	コードクローン	コードクローン以外
CCFinder	34.1389	37.3632
CCFinderX	35.6745	36.8383
Simian	24.4172	37.8184
Scorpio	20.5852	23.5483

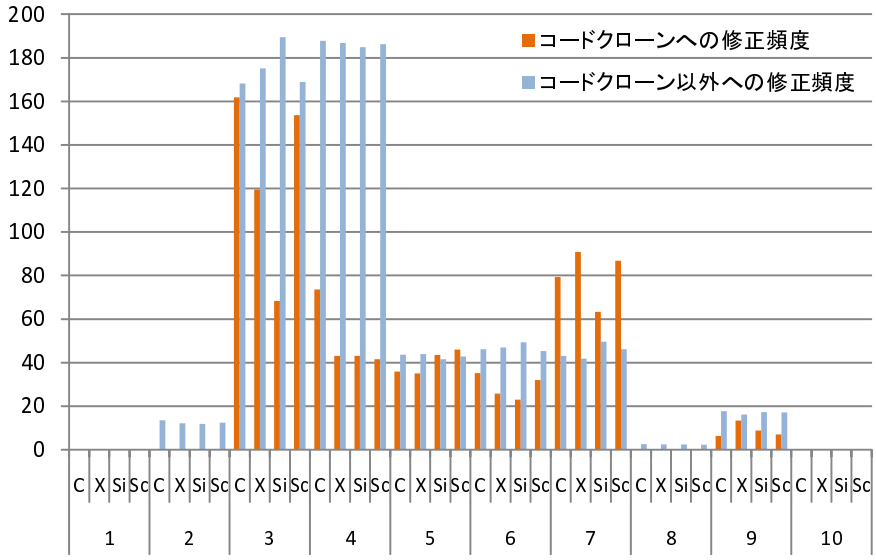


図 14: 計測値の推移 -AdServerBeans-

4.2.3 計測値の推移

本節ではリビジョンによってソフトウェアを 10 の区間に等分割し，それぞれの区間において修正頻度がどのように推移しているのかを調査した結果を述べる．

AdServerBeans，NatMonitor，OpenYMSG，Tritonn に対して調査を行った結果を図 14，図 15，図 16，図 17 にそれぞれ示す．図の縦軸が計測した修正頻度，横軸が開発期間と検出ツールを表しており，開発期間は数字が小さいものほど開発の早い段階であることを示している．

AdServerBeans では，7 番目の区間ではすべての検出ツールにおいてコードクローンへの修正頻度が高くなっており，5 番目の区間では *Simian* と *Scorpio* ではコードクローンへの修正頻度が高いが，*CCFinder* と *CCFinderX* ではコードクローン以外への修正頻度が高くなっている．その他の区間ではコードクローン以外への修正頻度が高くなっている．

NatMonitor では，開発期間の前半ではコードクローンへの修正頻度が低くなっているが，開発期間の後半ではコードクローン以外への修正頻度が高くなっている．OpenYMSG では，6 番目の区間まではコードクローンへの修正頻度が高くなっているが，7 番目の区間以降はすべての検出ツールにおいてコードクローン以外への修正頻度が低くなっている．

Tritonn では，3 つの検出ツールにおいて，*Simian* の計測結果は 3 番目の区間を除き，コー

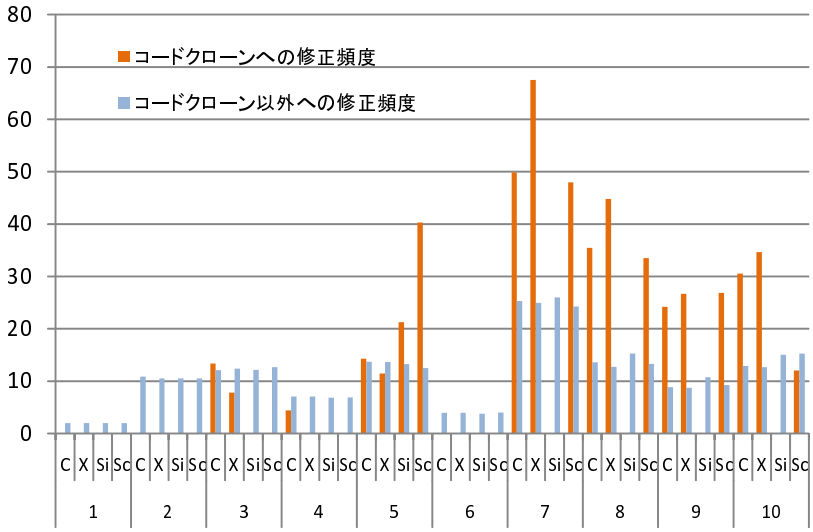


図 15: 計測値の推移 -NatMonitor-

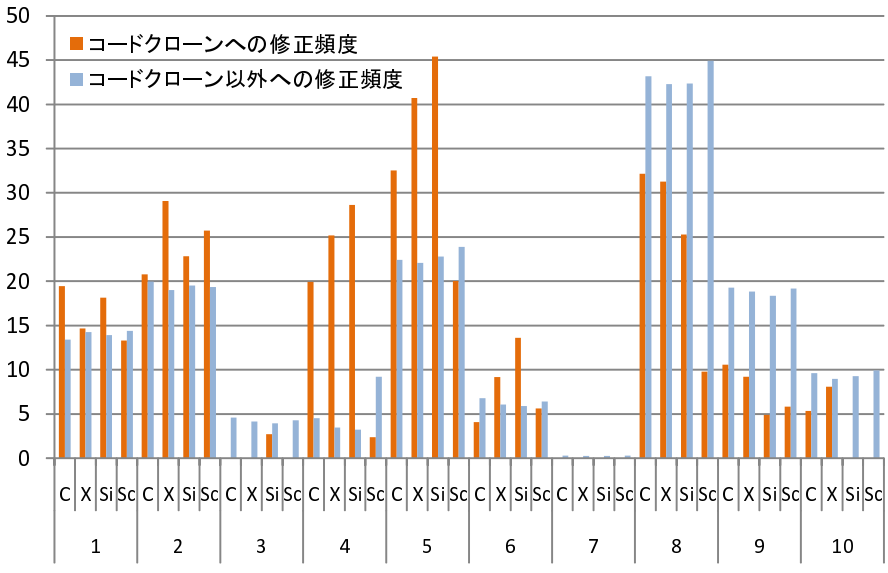


図 16: 計測値の推移 -OpenYMSG-

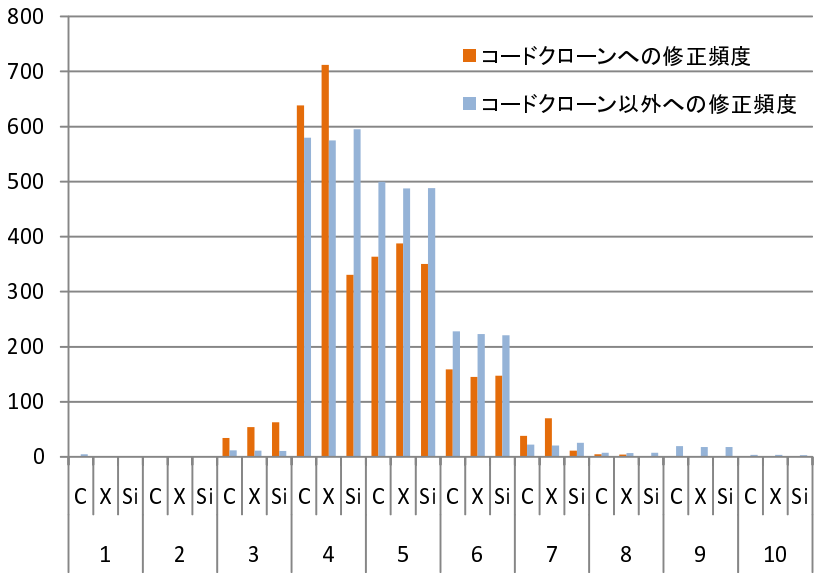


図 17: 計測値の推移 -Tritonn-

ドクローンへの修正頻度がコードクローン以外の部分への修正頻度を下回っている．しかし，他の 2 つの検出ツールについては 4 番目の区間，及び 7 番目の区間でもコードクローンへの修正頻度がコードクローン以外の部分への修正頻度より高い値となっている．特に 4 番目の区間におけるコードクローンへの修正頻度の差が顕著となっている．

4.2.4 計測結果のまとめ

コードクローンへの修正頻度とコードクローン以外への修正頻度をオープンソースソフトウェアを対象として計測したところ，コードクローンへの修正頻度の平均値がコードクローン以外への修正頻度の平均値と比べて低い，という結果を得た．またプログラミング言語別，及びコードクローン検出ツール別に平均値を計測したところ，すべての場合においてコードクローンへの修正頻度がコードクローン以外の修正頻度よりも低い，という結果となった．

次に，リビジョンによってソフトウェアを等分割し，それぞれの区間において修正頻度がどのように推移しているかを調査した．調査を行ったところ，コードクローンへの修正頻度がコードクローン以外への修正頻度より低い区間が多いが，一部区間ではコードクローンへの修正頻度が高くなっており，また検出ツールによって同じ区間でもコードクローンへの修正頻度に差が生じている，という結果を得た．

5　考察

5.1　修正頻度について

計測結果について，コードクローンへの修正頻度の平均値とコードクローン以外への修正頻度の平均値に有意な差があるかを，t 検定を行うことで評価した．その結果，2 つの値には有意水準 5%で有意な差はみられなかった．また，プログラミング言語ごとの計測結果，検出ツールごとの計測結果のそれぞれについても，有意水準 5%で有意な差はみられなかった．一般的にコードクローンはソフトウェアの修正に要する作業量を増大させると考えられているが，計測結果からはこのような傾向はみられなかった．

このような結果が得られた要因として，安定したコードの再利用を行っているという可能性がある．機能追加などを行う際，すでに動作を確認できている安定したコードを再利用した場合，新たにコードを記述するよりもバグが混入する可能性が低くなる．また，開発にコード生成ツールを利用している場合，コード生成ツールによって生成されたコードは安定したコードであると考えられる．

安定したコードの例を図 18 に示す．このコードは DatabaseToUML で記述されている関数であり，引数で受け取ったオブジェクトをコレクションに追加する処理を記述した関数である．この関数と同様の処理を行う `add***PropertyDescripter` という名前の関数が，4 つのソースファイルにわたって計 17 存在しており，これらの関数はすべて 1 つのクローンセットとして検出された．また，これらの関数はリビジョン 10 で同時に生成されてから最新のリビジョン 59 に至るまでの間に一切修正が加えられていなかった．これらの関数すべてに対してコメントに ”`@generated`” と記述されていることから，コード生成ツールを使用して生成されたコードであると考えられる．この例のようなコード生成ツールで生成されたコードや，動作の確認が行われており機能的に安定しているコードなどの安定したコードは修正が加えられにくく，これらのコードがコードクローンとして再利用されている割合が高ければ，本研究の計測結果のような結果が得られると考えられる．

5.2　修正頻度の推移について

5.2.1　AdServerBeans

AdServerBeans における修正頻度の推移について調査したところ，7 番目の期間ではすべての検出ツールにおいてコードクローンへの修正頻度が高くなっているが，その他の期間では一部を除きコードクローン以外への修正頻度が高くなっているという結果が得られた．

次に，コードクローンへの修正頻度とコードクローン以外への修正頻度の差が顕著な 4 番目の期間におけるソースコードと，コードクローンへの修正頻度が高くなっている 7 番目の

```
protected void addNamePropertyDescriptor(Object object) {
   itemPropertyDescriptors.add
       (createItemPropertyDescriptor
           (((ComposeableAdapterFactory)adapterFactory)
                  .getRootAdapterFactory(),
               getResourceLocator(),
               getString("_UI_NamedElement_name_feature"),
               getString("_UI_PropertyDescriptor_description",
                  "_UI_NamedElement_name_feature", "_UI_NamedElement_type"),
               MetadataPackage.Literals.NAMED_ELEMENT__NAME,
               true,
               false,
               false,
               ItemPropertyDescriptor.GENERIC_VALUE_IMAGE,
               null,
               null));
}
```

図 18: 安定したコードの例

期間におけるソースコードをそれぞれ調査した．4 番目の期間ではファイル単位でのコピーやメソッド単位でのコピーなどが頻繁に行われており，コピーされたソースファイルのほとんどがこの期間のうちに削除されていたため，コードクローン含有率が他の期間と比べ高くなっていた．なおかつ，それらのソースファイルやメソッドにあまり変更が加えられなかったため，コードクローンへの修正頻度が低くなったと考えられる．7 番目の期間ではクローンセット内のコード片に対して同様の修正が行われている場合が他の期間と比べ多く存在したため，コードクローンへの修正頻度が高くなったと考えられる．

このような不安定なコードクローンと加えられた修正の例を図 19 に示す．このコード片が属するクローンセットの要素数は 7 番目の期間においては 6 であった．1 度目の修正はオブジェクト `dataGridDisplayCriteria` のフィールドを取得するためのメソッドが廃止されたことによる修正であり (図中の%で示した箇所)，2 度目の修正は `offsetTmp` が 0 より大きいときの処理を削除する修正 (図中の#で示した箇所) である．これらの 2 つの修正はともに 7 番目の期間に加えられていた．このように短期間に変更が加えられる不安定なコードクローンは，ソフトウェアの修正に要する作業量を増大させる恐れがあると考えられる．また 2 度目の修正に関しては，クローンセット内のある 1 つのコード片に対しては修正が加えられていなかった．修正されなかったコード片はその後最新リビジョンに至るまでの間修正が加えられていなかったため，この修正を行わなかったことによるバグ等は発見されていないと考えられる．しかし，もし今後この修正を行わなかったことによるバグが発見された場合，

同様の処理を行うコードに対して修正が必要かを検討する必要が生じる．このような場合は修正に要する作業量は非常に大きいと考えられる．

5.2.2 NatMonitor

NatMonitor における修正頻度の推移について調査したところ，開発期間の前半ではコードクローン以外への修正頻度が高くなっており，開発期間の後半ではコードクローンへの修正頻度が高くなっているという結果が得られた．

NatMonitor のソースコードを調査したところ，開発期間の前半では各リビジョンにおけるクローンセット数が非常に少なかったため，コードクローンにほとんど修正が加えられていなかった．開発期間の後半では，各リビジョンにおけるクローンセット数は前半と比較して増加しているが，他のソフトウェアと比較するとコードクローン含有率が低く，なおかつクローンセット内のすべてのコード片に対して同じ修正が行われているリビジョンがいくつか存在した．コードクローン含有率が低いことと，クローンセット内のコード片に同様の修正が行われていたことから，開発期間の後半においてコードクローンへの修正頻度が高くなっていると考えられる．

なお，NatMonitor における *Simian* を用いた調査では，期間 5 以外のすべての期間においてコードクローンへの修正頻度が 0 となっている．ソースコードに対して調査を行ったところ，期間 5 では Type-1 のコードクローンが存在しているが，それ以外の期間では Type-1 のコードクローンが存在せずに Type-2 のコードクローンのみが存在していた．*Simian* のデフォルトの設定では他の検出ツールで検出できた Type-2 のコードクローンを検出できなかったため，*Simian* では期間 5 以外の期間ではコードクローンを検出できなかった．このため，期間 5 以外のすべての期間でコードクローンへの修正頻度が 0 となっていた．*Simian* のデフォルト設定で検出できなかった Type-2 のコードクローンと，そのコード片に加えられた修正の例を図 20 に示す．

5.2.3 OpenYMSG

OpenYMSG における修正頻度の推移について調査したところ，開発期間の前半ではコードクローンへの修正頻度が高くなっており，開発期間の後半ではコードクローン以外への修正頻度が高くなっているという NatMonitor とは反対の結果が得られた．

OpenYMSG における計測結果を解析すると，開発期間の前半では`***Test.java` のようなある機能の動作をテストするためのコードに高い頻度で修正が加えられていた．これらのコードは，同じ処理を変数の値や関数に与える引数などを変化させながら意図的に繰り返し記述されている場合や，他のファイルに記述されたコードの処理の内容を変えずに変数の型

```
  int offsetTmp = dataGridDisplayCriteria
         .getItemsPerPage() * (dataGridDisplayCriteria.getPage() - 1);
  if (offsetTmp > 0) --offsetTmp;
  if (offsetTmp < 0) offsetTmp = 0;
  final int offset = offsetTmp;
  String sortColumn = dataGridDisplayCriteria.getSortColumn();
  Order orderTmp =
         dataGridDisplayCriteria.getOrder()
                 .equals(AdServerBeansConstants.ASC) ?
                 Order.asc(sortColumn) : Order.desc(sortColumn);
```

(a) 修正前

```
  int offsetTmp = dataGridDisplayCriteria
         .getItemsPerPage() * (dataGridDisplayCriteria.getPage() - 1);
  if (offsetTmp > 0) --offsetTmp;
  if (offsetTmp < 0) offsetTmp = 0;
  final int offset = offsetTmp;
% String sortColumn = dataGridDisplayCriteria.sortColumn;
  Order orderTmp =
%         dataGridDisplayCriteria.order
                 .equals(AdServerBeansConstants.ASC) ?
                 Order.asc(sortColumn) : Order.desc(sortColumn);
```

(b) 1 度目の修正後

```
  int offsetTmp = dataGridDisplayCriteria
         .getItemsPerPage() * (dataGridDisplayCriteria.getPage() - 1);
#
  if (offsetTmp < 0) offsetTmp = 0;
  final int offset = offsetTmp;
  String sortColumn = dataGridDisplayCriteria.sortColumn;
  Order orderTmp =
         dataGridDisplayCriteria.order
                  .equals(AdServerBeansConstants.ASC) ?
                 Order.asc(sortColumn) : Order.desc(sortColumn);
```

(c) 2 度目の修正後

図 19: 不安定なコードクローンの例

```
  cb = new JCheckBox("Graph");
  cb.setSelected(NATMonitorPrefs.isGraphEnabled());
  cb.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
                NATMonitorPrefs.setGraphEnabled(((JCheckBox)
                        e.getSource()).isSelected());
                popup.setVisible(false);
        }
  });
```

```
  rb = new JRadioButton("Auto");
  rb.setSelected(NATMonitorPrefs.getZoomType()
                == NATMonitorPrefs.ZoomType.ZOOM_AUTO);
  rb.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
                NATMonitorPrefs.setZoomType(NATMonitorPrefs
                        .ZoomType.ZOOM_AUTO);
                popup.setVisible(false);
        }
  });
```

(a) 修正前のコードクローン

```
% cb = new JCheckBox("Resolve hostname",
%                NATMonitorPrefs.isResolveEnabled());
  cb.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
%                NATMonitorPrefs.setResolveEnabled(((JCheckBox)
%                        e.getSource()).isSelected());
                popup.setVisible(false);
        }
  });
```

```
# rb = new JRadioButton("Auto",
#                NATMonitorPrefs.getZoomType() ==
#                        NATMonitorPrefs.ZoomType.ZOOM_AUTO);
  rb.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
                NATMonitorPrefs.setZoomType(NATMonitorPrefs
                        .ZoomType.ZOOM_AUTO);
                popup.setVisible(false);
        }
  });
```

(b) 修正後のコードクローン

図 20: Type-2 のコードクローン

```
sender.login(USERNAME, PASSWORD);
SessionFriendEvent event =
       listener.waitForNextEvent(MAX_WAIT_IN_MILLIS);
assertNotNull(event);
assertEquals(Status.AVAILABLE,
       event.getFriend().getStatus());
assertEquals(null,
       event.getFriend().getCustomStatusMessage());

sender.setStatus(Status.BUSY);
event = listener.waitForNextEvent(MAX_WAIT_IN_MILLIS);
assertNotNull(event);
assertEquals(event.getFriend().toString(),
       Status.BUSY, event.getFriend().getStatus());
assertEquals(null,
       event.getFriend().getCustomStatusMessage());

...
```

図 21: 意図的に繰り返されているコードの例

などのみを変更して利用している場合が多いため，これらのコードから多くのコードクローンが検出された (図 21)．テスト用のコードの条件などを変化させる修正がたびたび加えられており，かつ同様の処理を行うコードに対して同様の条件変更が行われていたため，期間の前半においてコードクローンへの修正頻度が高い値となったと考えられる．しかし，これらの修正はソフトウェアの機能に直接影響を与えないため，ソフトウェアの修正に要する作業量に与える影響は小さいと考えられる．

5.2.4 Tritonn

Tritonn における修正頻度の推移について調査したところ，*CCFinder* もしくは *CCFinderX* を用いた場合はある時期のコードクローンへの修正頻度が高くなっているが，*Simian* を用いた場合は同じ時期のコードクローンへの修正頻度は他の 2 つのツールを用いた場合と比べて著しく低いという結果が得られた．

この期間における Tritonn のソースコードを調査したところ，この期間ではソースファイルの削除が他の期間と比べ頻繁に行われていた．ソースファイルの削除が行われた場合，削除したソースファイルにコードクローンが含まれていればコードクローンとコードクローン以外の部分にそれぞれ 1 箇所ずつ修正が加えられたとみなされる．このため，ソースファイルの削除が多く，かつコードクローン含有率が低い場合，コードクローン含有率を用いた正

規化によってコードクローンへの修正頻度が高くなる．Tritonn の 4 番目の期間においては，ソースファイルの削除が頻繁に行われ，かつ *Simian* では検出できなかったコードクローンが削除したソースファイルに含まれていたため，このような結果が得られたと考えられる．

6　結果の妥当性

本研究の結果の妥当性に関して，以下で挙げる点に留意する必要がある．

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仮定して調査を行っている．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，コードクローンへの修正は一貫性が必要であり，修正に一貫性が無い場合 (クローンセット中のあるコード片に修正を加え忘れ，次のリビジョンで修正を加えた場合など)，後のリビジョンで再度クローンセット中の全てのコード片に対して修正漏れがないか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．反対に，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 (図 22) 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が，このような修正はソースコードの意味的な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．

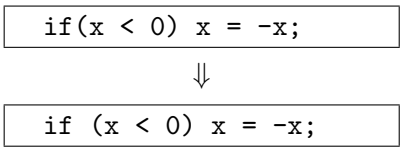


図 22: フォーマット変換

対象ソフトウェアのリビジョン数

本研究で対象としたソフトウェアは Squirrel-SQL を除き，比較的リビジョン数の小さいソフトウェアである．しかし，よりリビジョン数の大きい複数のソフトウェアに対して修正頻度を計測した場合，今回導かれた結果とは異なる結果となる可能性がある．

検出ツールの設定

本研究では，検出ツールを用いてコードクローンを検出する際，全てデフォルトの設定で検出を行っている．検出ツールの設定を変え，検出するコードクローンの種類 (コードクローン関係にあるコード片がどの程度一致するのかや，最少コード片の大きさなど) を変更することで，本研究と異なる結果となる可能性がある．

計測対象

本研究では，計測対象をオープンソースソフトウェアのみに限定している．しかし，一般的に商用ソフトウェアはオープンソースソフトウェアと比べてコードクローン含有率が高いといわれている．このため，商用ソフトウェアに対して計測を行った場合，本研究とは異なる結果が導かれる可能性がある．

期間分割の方法

本研究では，修正頻度の開発期間による推移を調査する際，リビジョン数で機械的に分割を行った．しかし，バージョンの区切りごとの分割など，分割の方法を変更して調査を行った場合，本研究で得られた結果とは異なる結果となる可能性がある．

7　関連研究

門田ら [7] は，COBOL で記述された大規模なレガシーソフトウェアに対し，信頼性・保守性とコードクローンとの関係を分析している．コードクローンを含むモジュールは含まないモジュールより信頼性が約 40%高いが，200 行を超える大きなコードクローンを含むモジュールは逆に信頼性が低下し，またコードクローンを含むモジュールは含まないモジュールより改版数が約 40 %高い (保守性が低い)，と報告している．

Lozano ら [8] は，ソフトウェア保守に対するコードクローンの影響について，メソッド単位で調査を行った．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．

Krinke ら [9] は，もしコードクローンがコードクローン以外の部分より安定性に欠くなら，保守においてコードクローンに要するコストが高いと仮定し，システムの進化において，コードクローンの安定性について調査を行った．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対してコードクローンとコードクローン以外の部分のそれぞれに対して行われる追加，変更，削除の行数を計測し，コードクローンとコードクローン以外の部分のそれぞれに対する割合を比較した．その結果，コードクローンの方が追加，変更，削除が行われる割合の平均が低く，また，追加，変更，削除が行われる割合がコードクローンの方が低い週が多かった．この結果から，コードクローンの方がコードクローン以外の部分より安定しており，一般的にコードクローンの保守に要するコストの方がコードクローン以外の部分の保守に要するコストより高いとは仮定できない，と報告している．

Eick ら [20] 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難くなること) が引き起こされているのか調査している．Eick 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い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と報告している．

Nils ら [21] 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．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用し，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．

Lozano ら [22] は，バージョン管理システム *CVS* で管理されているソフトウェアに対して，メソッドが変更された期間，ある期間においてメソッドが含んでいるコードクローンの割合，及びある期間において複数のメソッドが共有しているコードクローンの数を算出するツール *CloneTracker*[23] を作成した．また作成したツールをある Java ソフトウェアに適用した結果，全てあるいは一部の期間でコードクローンを含んでいたメソッドは，全ての期間でコードクローンを含んでいないメソッドと比較して，より高い頻度で変更が加えられていると報告している．

8 あとがき

本研究では，コードクローンがソフトウェアの修正作業量にどの程度影響を与えているのかをソースコードに加えられる修正の頻度に着目して定量的に調査した．また，コードクローン検出ツールによらないより一般的な結果を得るために，*CCFinder*，*CCFinderX*，*Simian*，*Scorpio* の 4 種類のコードクローン検出ツールを用いて調査を行った．

調査を行ったところ，コードクローンに加えられる修正頻度とコードクローン以外の部分に加えられる修正頻度に統計的に有意な差は見られない，という結果を得た．

本研究の今後の課題は以下の通りである．

- よりリビジョン数の大きいソフトウェアに対して計測を行う．
- 検出ツールの設定を変化させ，コードクローンとして検出するコード片の大きさなどを変化させて計測を行い，どのようなコードクローンが修正されやすいかなどを調査する．
- それぞれの修正箇所の作業量の違いや，コードクローンへの一貫性のない修正による手戻りの作業量などを考慮に入れた修正頻度の定義，計測を行う．

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，有益か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 柿元 健 特任助教に深く感謝申し上げます.

本研究に用いたツールの大部分を設計, 実装してくださり，また本研究に関して多大なるご助言，ご助力を頂きました，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博士前期課程 2 年の 佐野 由希子 氏に深く感謝申し上げます.

本報告を行うにあたり，多大なるご助言，ご助力を頂きました，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博士前期課程 1 年の 山田 慎也 氏に深く感謝申し上げます.

その他の楠本研究室の皆様のご助言，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大阪大学基礎工学部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, 2008.

[2] 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*, Vol. 28, No. 7, pp. 654–670, July 2002.

[3] CCFinderX. http://www.ccfinder.net/ccfinderx-j.html.

[4] Simian - similarity analyser. http://www.redhillconsulting.com.au/products/simian/.

[5] Scorpio. http://www-sdl.ist.osaka-u.ac.jp/˜higo/cgi-bin/moin.cgi/scorpio.

[6] J.H. Johnson. Substring matching for clone detection tools. *Proc. International Conference on Software Maintenance 94*, pp. 120–126, Sep. 1994.

[7] 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, 2003.

[8] A. Lozano and M. Wermelinger. Asessing the effect of clones on changeability. *International Conference on Software Maintenance*, pp. 227–236, 2008.

[9] J. Krinke. Is cloned code more stable than non-cloned code? *Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*, 2008.

[10] Subversion. http://subversion.apache.org/.

[11] ソースコード正規化ツール CommentRemover. http://www-sdl.ist.osaka-u.ac.jp/˜higo/cgi-bin/moin.cgi/commentremover.

[12] 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, 2001.

[13] S. Bellon. Detection of software clones. *Technical Report, Institute for Software Technology, University of Stuttugart*, 2003. available at http://www.bauhaus-stuttgart.de/clones/.

[14] S. Bellon, R. Koschke, G. Antniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Trans. Software Engineering*, Vol. 31, No. 10, pp. 804–818, Oct. 2007.

[15] I. Baxter, A. Yahin, M. Anna L. Moura, and L. Bier. Clone detection using abstract syntax trees. *Proc. of International Conference on Software Maintenance 98*, pp. 368–377, Mar. 1998.

[16] S. Uchida, A. Monden, N. Ohsugi, T. Kamiya, K. Matsumoto, and H. Kudo. Software analysis by code clones in open source software. *Journal of Computer Information Systems*, Vol. XLV, No. 3, pp. 1–11, Apr. 2005.

[17] 肥後芳樹, 楠本真二. 実規模ソフトウェアへの適用を目的としたプログラム依存グラフに基づくコードクローン検出法. ソフトウェアエンジニアリング最前線 2008, 2009.

[18] diff-win. http://www.gfd-dennou.org/library/cc-env/diff-win/SIGEN.htm.

[19] SourceForge.net : Find and Develop Open Source Software. http://sourceforge.net/.

[20] S. G. Eick, T. L. Graves, A. F. Karr, J.S. Marron, and A. Mockus. Does code decay? assessing the evidence from change management data. *IEEE TRANSCATIONS ON SOFTWARE ENGINEERING*, Vol. 27, No. 1, pp. 1–12, Jan. 2001.

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

[22] A. Lozano, M. Wermelinger, and B. Nuseibeh. Evaluating the harmfulness of cloning: a change based experiment. *IEEE Fourth International Workshop on Mining Software Repositories*, 2007.

[23] Clone Tracker. http://mcs.open.ac.uk/alr242/CloneTracker.htm.