情報処理学会論文誌 Vol. 52 No. 9 2788–2798 (Sep. 2011)

修正頻度の比較に基づくソフトウェア修正作業量に対する重複コードの影響に関する調査

堀田圭佑[†1] 佐野由希子[†1]
肥後芳樹[†1] 楠本真二[†1]

近年，重複コードへの関心が高まっている．一般的に重複コードはソフトウェアの修正作業量を増大させるおそれがあると考えられており，重複コードの検出や集約に関する研究がさかんに行われている．しかし，重複コードと修正作業量の関係を定量的に調査した研究はあまり行われていない．そこで本論文では，重複コードが非重複コードと比較して修正されやすければ重複コードが修正作業量を増大させているという考えに基づき，ソースコードに加えられる修正の頻度を計測，比較することで，重複コードと修正作業量の関係を調査した．15 のオープンソースソフトウェアに対して実験を行った結果，非重複コードと比較して重複コードは修正されにくく，重複コードがソフトウェアの修正作業量を増大させているとは必ずしもいえないという結果を得た．

An Empirical Study of Influence of Duplicate Code on Software Maintenance Based on Modification Frequency Comparison

Keisuke Hotta,[†1] Yukiko Sano,[†1] Yoshiki Higo[†1]
and Shinji Kusumoto[†1]

Recently, duplicate code has received much attention. In general, duplicate code is considered as one of the factors that makes software maintenance more difficult. And there are many research efforts on detection or removal of duplicate code. However, there are only a few research efforts on the influence of existence of duplicate code on software maintenance effort. In this study, we research the maintenance effort on duplicated code and non-duplicated code based on the modification frequency under the assumption that if duplicate code is modified frequently, duplicate code has a negative impact on software maintenance. We experimented on 15 open source software systems, and the result showed that the modification frequency of duplicated code was smaller than the modification frequency of non-duplicated code. Consequently, we conclude that the existence of duplicate code does not necessarily have a negative impact to software maintenance effort.

1. まえがき

近年，ソフトウェアの保守を困難にする要素の 1 つとして，重複コードに対する関心が高まっている．重複コードは主にコピーアンドペーストにより生成され，その存在はソースコードの修正作業量を増大させるおそれがあると考えられている[1)]．これは，あるコード片に修正を加えた際，そのコード片と重複しているすべてのコード片に対して，同様の修正を検討しなければならないという理由からである．この定説に基づき，これまでに重複コードの検出や集約に関する研究がさかんに行われている[1)–6)]．しかし，実際に重複コードが修正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した研究はあまり行われていない．

また，既存研究の課題として，調査の粒度が大きいという点と，特定の重複コード検出ツールのみを用いて調査を行っているという点があげられる[7)–9)]．本論文では，重複コードの方が非重複コードと比較して不安定である，すなわち重複コードが非重複コードと比較して修正されやすければ重複コードが修正作業量を増大させているという考えに基づき，重複コードと非重複コードの修正のされやすさを比較することで，重複コードと修正作業量の関係を調査する．本論文における調査の特徴は次のとおりである．

- 修正されやすさを測るための指標として，**修正頻度**と呼ぶ値を定義する．この値は修正行数ではなく，修正箇所数に基づいている．ソースコードの修正に要する作業量の大部分は修正箇所の特定などの実際に修正を加える前の段階に要する可能性が高く，修正箇所数を用いる方がより正確に修正作業量を表現できると考え，本論文では修正箇所数に基づく指標を用いた．
- ソースコードの各行が重複コードであるかを調査する．これにより，ソースコードに加えられた修正が重複コードへの修正か否かをより正確に判別できる．
- 特定の重複コード検出ツールによらない結果を得るために，複数の重複コード検出ツールを利用する．

†1 大阪大学大学院情報科学研究科
Graduate School of Information and Science Technology, Osaka University

表 1　関連研究
Table 1　Related works.

手法	重複コードの判別単位	計測の指標	対象の数	検出ツール
門田らの手法[7]	モジュール	モジュールの改版数	1	*CCFinder*
Krinke の手法[8]	行	修正行数/行数	5	*Simian*
Lozano らの手法[9]	メソッド	メソッドの変更割合 × 同時に変更されるメソッドの割合	4	*CCFinder*
Göde の手法[10]	トークン	重複コードの割合，存在期間など	9	*iClones* [11]
Lozano らの手法[12]	メソッド	メソッドの修正回数と重複コードの割合	1	*CCFinder*
本論文の手法	行	修正の頻度	15	*CCFinder* など 4 種類

以降 2 章で関連する研究について触れ，3 章で重複コードの定義と重複コード検出ツールについて述べる．4 章で調査手法の概要と手順を説明する．5 章で実験結果とその考察について述べ，最後にまとめと今後の課題を 6 章に記す．

2. 関 連 研 究

本章では，重複コードとソフトウェアの保守性の関係を調査した関連研究について述べる．関連研究および本論文の手法の特性を**表 1** に示す．

門田らは，COBOL で記述された大規模なソフトウェアにおける，重複コードと信頼性，保守性の関係を分析した[7]．その結果，重複コード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 行を超える大きな重複コードを含むモジュールは逆に信頼性が低下し，また重複コードを含むモジュールは含まないモジュールより改版数が 40%高い（保守性が低い）と報告している．

Krinke は，ソースコードの安定性が低ければ保守に要するコストが高いと仮定し，重複コードの安定性について調査を行った[8]．5 つの大規模なソフトウェアを対象とし，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，変更，削除の行数を計測し，その割合を比較した．その結果，重複コードの方が追加，変更，削除の行われる割合が低く，重複コードの方が非重複コードと比較して安定性が高いと報告している．

Lozano らは，ソフトウェア保守に対する重複コードの影響をメソッド単位で調査した[9]．その評価指標として，likelihood（あるメソッドに対して変更が加えらえる割合），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ソフトウェアに対して実験を行った結果，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．

Göde は，類似度がきわめて高い重複コードに関して，それら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た[10]．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，重複コードの発展の様子を調査した．その結果，重複コードの割合は時間経過とともに減少していること，重複コードは平均で約 1 年以上重複コードとして存在していること，重複コードに一貫性のない修正（あるコード片を修正し，そのコード片と重複コード関係にある他のコード片に同様の修正を行っていない修正）が加わった場合，それらの修正がのちのバージョンにおいて一貫性のある修正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重複コードの割合，およびある期間において複数のメソッドが共有している重複コードの数を算出するツール *CloneTracker* [13] を作成した[12]．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，すべてあるいは一部の期間で重複コードを含んでいたメソッドは，すべての期間重複コード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．

これらの既存研究と比較して，本論文では 4 つの重複コード検出ツールを用いている点と，対象としたソフトウェアの数が多いという点が特長としてあげられる．また，本論文では Krinke の手法と同様に，ソースコードの各行が重複コードか否かを判別している．ファイルやメソッド，モジュール単位での計測手法では，一定の割合以上の重複コードを含むメソッドなどに対して修正が加えられたか否かを計測指標として用いている．しかしこの手法には，実際の修正が重複コード以外の箇所に加えられた場合でも，修正されたメソッドなどが一定の割合以上の重複コードを含んでいれば，重複コードが修正作業に影響を与えたと誤判定されるおそれがあるという課題点が存在する．行単位で重複コードか否かを判別す

ることでこの課題点の改善が期待できると考え，本論文では行単位での判別を行っている．また，Krinke の手法と本論文の手法では，重複コードの判別を行単位で行っているという点は共通しているが，Krinke の手法では行ベースの評価指標を用いているのに対し，本論文の手法では修正箇所ベースの評価指標を用いているという点が異なる．評価指標の違いに関する詳細は 4.2 節において述べる．

3. 重複コード

3.1 定　　義

重複コードとは，ソースコード中において類似している，もしくはまったく同一のコードを表すが，類似や同一の厳密で普遍的な定義はない．これまでに，数多くの重複コード検出ツールが開発されているが，各ツールは類似や同一に関する独自の定義を持ち，その定義に基づき重複コードの検出を行っている．そのため，同じソースコードから検出を行った場合でも，検出ツールが異なれば検出される重複コードは異なる[1]．

3.2 重複コード検出ツール

本節では本論文で用いる重複コード検出ツールについて述べる．

3.2.1 *CCFinder*

CCFinder[2] はソースコードを字句単位で比較することで解析を行う検出ツールである．*CCFinder* の持つ主な特徴は以下のとおりである．

- 変数名などのユーザ定義名を特殊な文字列に置き換える事前処理を行う．この処理により，ユーザ定義名のみが異なる重複コードを検出することができる．
- 大規模なソフトウェアから実用的な時間とメモリ使用量で解析を行うことができる．たとえば，約 500MLOC のソースコードを約 15 分（Xeon 2.67 GHz，RAM 6 GB）で解析可能である．
- 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．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解析は可能である．

3.2.2 *CCFinderX*

CCFinderX[3] 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比較に基づき重複コードを検出する．*CCFinderX* は重複コード検出に用いるアルゴリズムを *CCFinder* の *suffix tree* から *bucket sort* に変更している．また，*CCFinder* と比較してより多くのプログラミング言語に対応している．さらに，マルチコア CPU 環境においてより効率良く検出を行うためにマルチスレッド化を行っている．

3.2.3 *Simian*

Simian[4] はソースコードを行単位で比較することで重複コードの検出を行うツールである．*Simian* は様々なプログラミング言語に対応しており，プレーンテキストに対しても解析可能である．また，ソースコードに対する事前処理を必要としないため，少ないメモリ消費量で非常に高速に検出を行うことができるという特徴がある．さらに，コメント，空白，import 文，大文字小文字の違い，数値の違いを無視するかどうかなどの様々な設定を行うことが可能となっており，検出対象を細かく設定することが可能である．

3.2.4 *Scorpio*

Scorpio[5] はソースコードから特殊なプログラム依存グラフを構築し，同形部分グラフを重複コードとして検出するツールである．プログラム依存グラフの構築にともなう時間的，空間的コストが高いという欠点があるが，行単位や字句単位の検出手法では検出できない，順序が入れ替わっている重複コードや，重複コードが生成された後修正が加えられ，途中に非重複コードが含まれ分断されているような重複コードを検出することができるという特徴がある．また，*Scorpio* は，プログラム依存グラフを探索する際に探索の基点として用いる頂点に制限を設けることで，検出に要する時間的，空間的コストの削減を行っている．

4. 修正頻度の計測

本論文では，修正されやすさを修正頻度と呼ぶ指標を用いて定量化し，重複コードと非重複コードの修正頻度を比較することで調査を行う．本章では修正頻度の定義，およびその計測手順について述べる．なお，これ以降は重複コードの修正頻度を $\boldsymbol{MF_d}$，非重複コードの修正頻度を $\boldsymbol{MF_n}$ と表記する．

4.1 計測対象リビジョン

本論文ではバージョン管理システムによって管理されている開発履歴情報を利用する．管理されているリビジョンのうち，ソースファイルに修正が加えられたリビジョンを計測対象リビジョンと定義し，修正頻度の計測は計測対象リビジョンに対してのみ行う．すなわち，ドキュメントや画像ファイルなど，ソースファイル以外のファイルにのみ変更が加えられたリビジョンは計測の対象としない．

4.2 修 正 箇 所

本論文では調査単位として修正箇所数を用いる．既存の研究は修正行数を基にした調査を行っているが，この手法では『1 行の修正が 10 カ所にある場合』と『10 行の修正が 1 カ所

にある場合』の調査結果が等しくなる．しかし，ソースコードに修正を加える際，その作業量の大部分は修正すべき箇所の特定など，実際にコードに修正を加える前の段階に要すると我々は考えている．すなわち先の例の場合，『1 行の修正が 10 カ所にある場合』の方がより修正に要する作業量は大きいと考えている．本論文ではこの考えに基づき，修正箇所に基づいた調査を行う．

本論文では，連続して修正が加えられた行を 1 カ所の修正と見なす．それぞれの修正箇所について，以下の基準に基づいてその修正が重複コードと非重複コードのいずれに対する修正かを判別する．

- 修正が重複コードの領域内であれば，重複コードに対して 1 度修正が加えられたものとする．
- 修正が非重複コードの領域内であれば，非重複コードに対して 1 度修正が加えられたものとする．
- 修正が重複コードの領域と非重複コードの領域にまたがって存在している場合は，重複コードと非重複コードに対してそれぞれ 1 度ずつ修正が加えられたものとする．

4.3　修正頻度の定義

R を計測対象リビジョンの集合，$|R|$ を R の要素数，$MC_d(r)$，$MC_n(r)$ をそれぞれリビジョン r から次のリビジョンまでの期間において，重複コードおよび非重複コードに加えられた修正箇所数とする．このとき，MF_d，MF_n を次式で定義する．

$$MF_d = \frac{\sum_{r \in R} MC_d(r)}{|R|} \tag{1}$$

$$MF_n = \frac{\sum_{r \in R} MC_n(r)}{|R|} \tag{2}$$

これらは，重複コードおよび非重複コードへの修正箇所数を計測対象リビジョン数で除算した値である．これらの値はソースコード中に含まれる重複コードの割合に強く影響を受ける．たとえば，重複コードの割合が非常に低いソフトウェアに対して計測を行うと，重複コードへの修正箇所数が小さくなり，結果として MF_d が小さくなる可能性が高い．また，これらの値を用いた場合『修正されやすい重複コードが少数存在する場合』と『修正されにくい重複コードが多数存在する場合』の判別ができない．我々は不安定な重複コードの存在がソフトウェアの修正作業量増大につながるおそれがあるという考えに基づき，重複コードの修正されやすさを評価したいと考えている．このため，この定義をそのまま使用することができない．

重複コードの割合による影響を除去し，重複コードの安定性を評価するために，重複コードの総行数とソースコードの総行数を用いた正規化を行う．$LOC_d(r)$ をリビジョン r における重複コードの総行数，$LOC_n(r)$ をリビジョン r における非重複コードの総行数，$LOC(r)$ をリビジョン r におけるソースコードの総行数とする．このとき，定義より次式が成立する．

$$LOC(r) = LOC_d(r) + LOC_n(r)$$

正規化後の MF_d，MF_n を次式で定義する．

$$MF_d = \frac{\sum_{r \in R} \left(MC_d(r) \times \frac{LOC(r)}{LOC_d(r)} \right)}{|R|} \tag{3}$$

$$MF_n = \frac{\sum_{r \in R} \left(MC_n(r) \times \frac{LOC(r)}{LOC_n(r)} \right)}{|R|} \tag{4}$$

以降，本論文では正規化後の値を MF_d，MF_n として用いる．

4.4　計 測 手 順

本節では，修正頻度の計測手順について述べる．

手順 1：計測対象リビジョンの特定

バージョン管理システムで管理されている開発履歴情報をもとに，計測対象リビジョンを特定する．

手順 2：ソースコードの正規化

計測対象リビジョンのすべてのソースファイルに対し，次に示す正規化を行う．

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

正規化の例を**図 1** に示す．これらの正規化を行う目的は，インデントや中括弧の位置の変更など，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修正を計測から除外することである．

手順 3：重複コードの検出

それぞれの計測対象リビジョンのソースファイルに対し，重複コード検出ツールを適用する．検出結果をもとに，ソースファイルの各行が重複コードか否かを特定する．

```
 1: // this is a sample
 2: doSomething1();
 3: if (x == 0) {
 4:   doSomething2();
 5: } else {
 6:   doSomething3();
 7: }
 8:
 9: for(int i = 0; i < 10; i++){
10:   doSomething4(i);
11: }
```

(a) 正規化前

```
1: doSomething1();
2: if (x == 0) {
3: doSomething2();
4: } else {
5: doSomething3();}
6: for(int i = 0; i < 10; i++) {
7: doSomething4(i);}
```

(b) 正規化後

図 1　ソースコード正規化の例
Fig. 1　An example of normalization of source code.

手順 4：修正箇所の特定

すべての計測対象リビジョンについて，計測対象リビジョンとその次のリビジョンにおけるソースファイルの差分を計測し，それぞれのソースファイルのどの行からどの行までが計測対象リビジョンにおいて修正されたかを特定する．

手順 5：修正箇所数の計測

手順 3 で計測された重複コードの位置情報と手順 4 で計測された修正箇所の位置情報をもとに，すべての計測対象リビジョンについて，重複コードに加えられた修正箇所数と非重複コードに加えられた修正箇所数をそれぞれ計測する．

手順 6：修正頻度の算出

手順 5 で計測した修正箇所数とソースコードの総行数，および重複コードの総行数を用いて，4.3 節で定義した MF_d，MF_n を算出する．

5. 実　　験

5.1　実験の概要

本論文では以下に示す 2 種類の実験を行う．

第 1 実験：様々な規模のソフトウェアに対して，*CCFinder* を用いて MF_d および MF_n の計測を行う．この実験の目的は，ソフトウェアの規模（開発期間）と重複コードの修正されやすさとの関係を調査することである．

第 2 実験：小規模のソフトウェアに対して，複数の重複コード検出ツールを用いて MF_d および MF_n の計測を行う．この実験の目的は，重複コード検出ツールの違いによる重複コードの修正されやすさへの影響を調査することである．

また，それぞれの実験において，以下の 2 つの項目を調査する．

項目 A：開発の全期間における MF_d と MF_n を計測する．この調査項目は重複コードと非重複コードのどちらが高い頻度で修正されているかを知ることを目的としている．

項目 B：開発期間を 10 の区間に等分割し，それぞれの区間における MF_d と MF_n を計測する．この調査項目は開発の初期と後期で重複コードの修正されやすさに差があるかどうかを知ることを目的としている．

なお，本論文では実装の都合上，ソースファイルの差分取得のために外部ツール *diff-win* [14] を用い，またバージョン管理システム *Subversion* の開発履歴情報を計測に使用している．

5.2　実 験 対 象

それぞれの実験における実験対象を**表 2** に示す．なお，これらを選択した基準は以下のとおりである．

- *Sourceforge* において公開されているオープンソースソフトウェアである．
- バージョン管理システム *Subversion* を用いて開発を行っている．
- 代表的なプログラミング言語である Java，C，C++のいずれかを開発に用いている．

5.3　実 験 結 果

5.3.1　第 1 実 験

第 1 実験における項目 A の調査結果を**図 2** に示す．調査の結果，すべてのソフトウェアにおいて MF_d は MF_n より小さい，つまり重複コードは非重複コードに比べて修正されにくいという結果となった．

次に，第 1 実験における項目 B の調査結果を**図 3** に示す．グラフの横軸が分割した区間を表しており，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．調査の結果，FileZilla，FreeCol，WinMerge の 3 つのソフトウェアにおいては，それぞれ 1 つの区間を除くすべての区間で MF_d が MF_n より小さく，SQuirrel SQL Client においてはすべての区間で MF_d が MF_n より小さいという結果となった．この結果から，開発期間が長くなるほど重複コードは修正されにくい傾向にあるが，修正頻度の推移はそれぞれのソフトウェアによって異なるといえる．また WinMerge について，最後の区間において MF_d が MF_n より大きくなっている．この区間ではテストケースへの修正が多くなっており，テストケースの多くの部分が重複コードとして検出されたため，このような結果が得られた可能性が高い．

第 1 実験より，ソフトウェアの規模が大きくかつ開発期間が長い場合，*CCFinder* によっ

表 2　実験対象
Table 2　Target software systems.

(a) 第 1 実験（Experiment 1）

ソフトウェア名	プログラミング言語	リビジョン数	総行数（最新リビジョン）	開発期間		
EclEmma	Java	788	15,328	2006.8.26	–	2009.10.22
FileZilla	C++	3,450	87,282	2004.3.8	–	2009.10.18
FreeCol	Java	5,963	89,661	2002.1.14	–	2009.11.16
SQuirrel SQL Client	Java	5,351	207,376	2001.6.2	–	2009.9.10
WinMerge	C++	7,082	130,283	2000.10.27	–	2010.1.4

(b) 第 2 実験（Experiment 2）

ソフトウェア名	プログラミング言語	リビジョン数	総行数（最新リビジョン）	開発期間		
ThreeCAM	Java	14	3,584	2007.8.24	–	2007.11.18
DatabaseToUML	Java	59	19,695	2009.5.29	–	2009.8.9
AdServerBeans	Java	98	7,406	2009.3.23	–	2009.8.28
NatMonitor	Java	128	1,139	2008.5.26	–	2008.11.9
OpenYMSG	Java	141	130,072	2007.3.24	–	2009.9.25
QMailAdmin	C	312	173,688	2003.6.6	–	2009.8.19
Tritonn	C/C++	100	45,368	2008.7.1	–	2009.7.22
Newsstar	C	165	192,716	2003.10.21	–	2009.8.3
Hamachi-GUI	C	190	65,790	2006.5.29	–	2009.10.27
GameScanner	C/C++	420	1,214,570	2008.7.2	–	2009.11.4

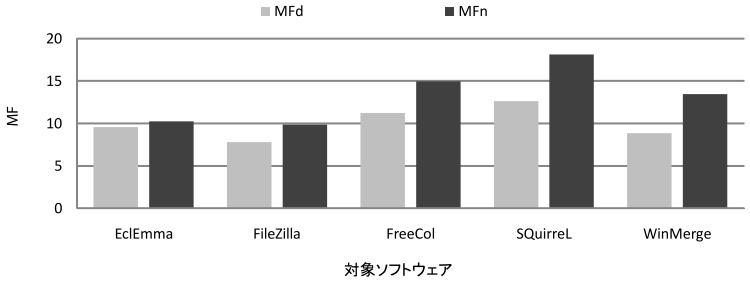
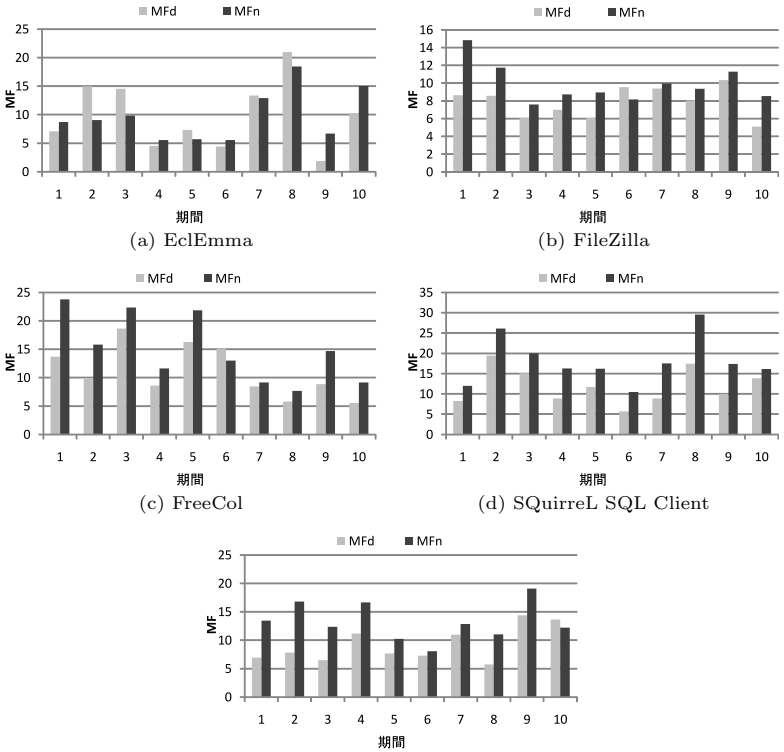


図 2　第 1 実験：全期間
Fig. 2　Result of experiment1: entire revisions.

て検出される重複コードは非重複コードと比較して修正されにくいという結果を得た．

5.3.2　第 2 実 験

第 2 実験における項目 A の調査結果を**図 4** に示す．グラフの横軸が対象ソフトウェア名と使用した重複コード検出ツールの略称を示しており，*CCFinder* → C，*CCFinderX* → X，*Simian* → Si，*Scorpio* → Sc にそれぞれ対応している．なお，*Scorpio* が Java にのみ対応しているため，*Scorpio* を用いた実験結果は Java に対するもののみとなっている．調査の結果，全 35 のデータのうち 22 のデータにおいて，MF_d が MF_n よりも小さいという結果となった．また，プログラミング言語別，および検出ツール別の MF_d と MF_n の平均値を**表 3** に示す．この表より，すべての言語およびすべての検出ツールにおいて MF_d が MF_n よりも小さいという結果となった．また，全体の平均値，言語別の平均値，検出ツール別の平均値のいずれについても，MF_d と MF_n の間に有意水準 5%で有意な差は見られなかった．



(a) EclEmma　(b) FileZilla

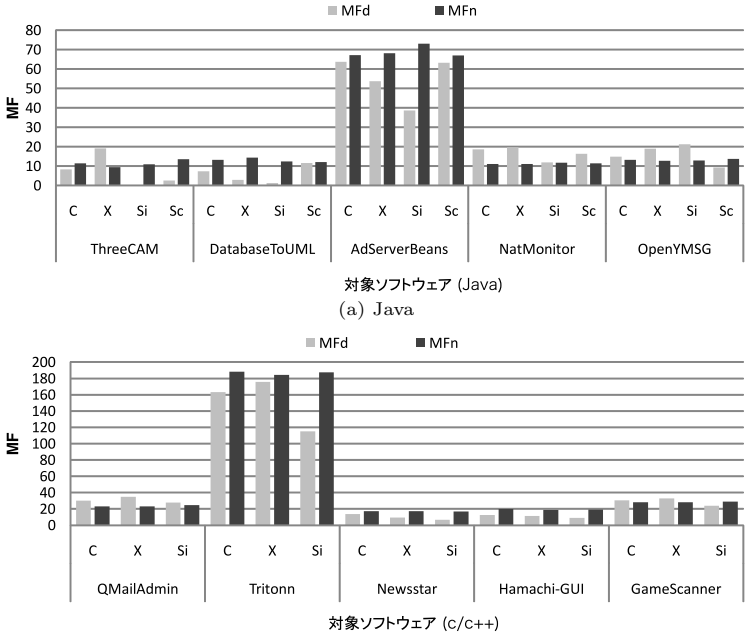
(c) FreeCol　(d) SQuirreL SQL Client

(e) WinMerge

図 3　第 1 実験：期間分割
Fig. 3　Result of experiment1: divided on period.

次に，第 2 実験における項目 B について，3 つのソフトウェアにおける調査結果を**図 5** に示す．AdServerBeans では 5 番目および 7 番目の区間以外の区間において，MF_d は MF_n より小さいという結果となった．OpenYMSG では開発の前半では MF_d が大きくなっているが後半では MF_d が小さくなっており，反対に NatMonitor では開発の前半は MF_d が小さく，後半は MF_d が大きいという結果が得られた．その他のソフトウェアについても，すべての区間において MF_d が MF_n より小さいものは存在しなかった．

第 2 実験より，中規模または小規模のソフトウェアにおいては重複コードと非重複コード



(a) Java

(b) C/C++

図 4　第 2 実験：全期間
Fig. 4　Result of experiment2: entire revisions.

の修正頻度に大きな差は見られず，また重複コード検出ツール間でも結果に顕著な差は見られないという結果を得た．

5.4 考　　察

一般的に重複コード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．しかし調査の結果，重複コードは非重複コードと比較して修正が加えられにくいという傾向が見られた．すなわち，すべての重複コードが修正の作業量を増大させているとはいえないという結果となった．

この結果から，すべての重複コードに対して同じように除去作業を行うことは効率的ではないといえる．すなわち，修正されにくい重複コードは修正されやすいものと比べ保守性に悪影響をもたらす可能性が低く，このような重複コードに対する除去作業によって得られる利点が少なく，反対に除去作業によって別の不具合を作り込んでしまうおそれがあるとい

表 3　MF_d と MF_n の平均値
Table 3　The means of MF_d and MF_n.

(a) プログラミング言語別

言語	MF_d	MF_n
Java	20.1547	23.5330
C/C++	46.4531	55.0157
ALL	32.1764	38.0545

(b) 検出ツール別

検出ツール	MF_d	MF_n
CCFinder	36.2851	39.2862
CCFinderX	37.8827	38.7334
Simian	25.5285	39.7959
Scorpio	20.5851	23.5483
ALL	32.1764	38.0545

える．しかし，修正されやすい重複コードに対する除去作業は高い効果が期待できるため，修正されやすい重複コードを特定して除去作業を行うことが必要であるといえる．

このような結果が得られた要因として，以下の点があげられる．

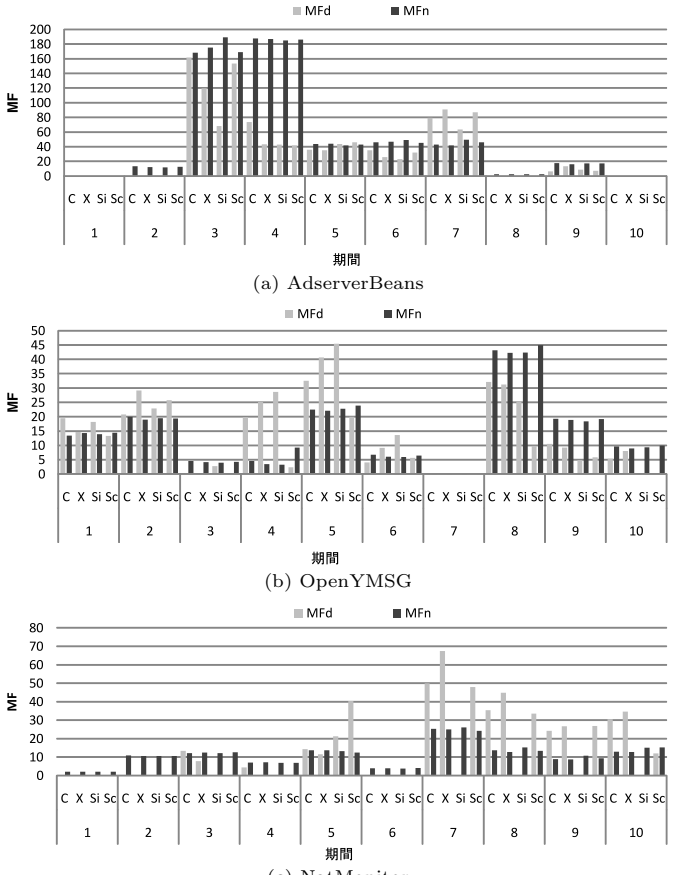
機能的に安定しているコードの再利用

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ている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少なくなる可能性が高い．

コード生成ツールの利用

コード生成ツールはコードのテンプレートから必要な処理に応じてコードを組み合わせて出力する．開発にコード生成ツールを利用している場合，同様の処理を行うコードを生成した際，生成されるコードは重複コードとして検出される可能性が高く，かつ修正されにくいコードであるといえる．

コード生成ツールを利用したコードの例を**図 6** に示す．このコードは DatabaseToUML で記述されているコードであり，“@generated” と記述されていることから自動生成されたコードであると判断できる．このメソッドは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メソッドであり，このメソッドと同様の処理を行う *addXXXPropertyDescriptor*（*XXX* はメソッドにより異なる）が 4 つのソースファイルにまたがって計 17 存在しており，生成されてから最新リビジョンに至るまでの 49 リビジョンの間，いっさい修正が加えられなかった．この例のような安定したコードが重複コードとして再利用されていたため，今回の調査結果が得られた可能性がある．



(a) AdserverBeans

(b) OpenYMSG

(c) NatMonitor

図 5　第 2 実験：期間分割
Fig. 5　Results of experiment2: divided on period.

なお，自動生成されたコードを除外することで異なる傾向の結果が得られる可能性があるが，自動生成されたコードに開発者が修正を加えることがしばしば発生する可能性があるため，今回の調査ではバージョン管理システムによって管理されているリポジトリから取得できるソースコード全体を計測の対象とした．

```
@generated
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 6　自動生成されたコードの例
Fig. 6　An example of auto generated code.

5.5　結果の妥当性

本節では，本論文の結果の妥当性を脅かすおそれがある要因について述べる．

修正に要する作業量

本論文では，1 カ所の修正に要する作業量はすべて等しいという仮定の下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 カ所の修正に多大な作業量を要する箇所も存在する．そのため，本論文での調査は修正作業量を厳密に評価できているとはいえない．

阿萬らはオブジェクト指向ソフトウェア開発におけるメトリクスの 1 つとしてクラスサイズをあげ，クラスサイズが大きいクラスほどその変更には大きな作業量を要するという考えに基づき，クラスサイズがソフトウェア変更作業量を予測する指標として有用かを調査している[15)]．その結果，精度の面で課題点が見られるもののある程度有用であると報告している．たとえばこの研究成果を応用し，クラスサイズの大きいクラスほど修正に要する作業量が大きいと考え，クラスサイズを加味した指標を用いることによって，本研究における修正作業量が加味できていないという課題点の改善が見込まれる．しかしながら，この方法を用いても修正作業量の厳密な評価は困難であるといえる．

```
if(x < 0) x = -x;
        ⇓
if (x < 0) x = -x;
```

図 7　プログラムの動作に影響を与えない修正の例
Fig. 7　An example of a modification that does not change the behavior of the program.

この課題点を完全に克服する方法として，加えられたすべての修正を人が見て修正作業量を評価する方法や，開発プロセスをモニタリングし，実際に要した修正コストを記録する方法などがあげられるが，いずれもその実現はきわめて難しいといえる．

修正箇所の判別

本論文では，連続して修正が加えられた行を 1 カ所の修正と見なしている．しかしこの判別手法では，途中修正しない行が存在することで本来 1 カ所と見なされるべき修正が複数箇所と見なされるおそれがある．反対に，本来複数箇所と見なされるべき修正が偶然連続して行われていた場合，それらの修正が 1 カ所の修正と見なされるおそれがある．このため，厳密に修正箇所を調査して実験を行った場合，今回の結果とは異なる結果が導かれる可能性がある．

この問題点を改善するための方法として，Bugzilla などのバグ管理システムで管理されているバグ修正情報とリビジョン間の差分を照らし合わせ，修正されたバグの数を修正箇所数と見なす方法があげられる．しかし，完全にこの問題による影響を取り除くためには人が見て修正箇所を判別する必要があり，その実現は困難であるといえる．

修正の内容

本論文では，修正の内容に関わりなくすべての修正を計測している．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修正による影響を取り除くため，4.4 節で述べたソースコードの正規化を行っているが，**図 7** に示す例のように，この作業を施しても取り除くことのできないものが存在する．このため，フォーマット変換のような，バグ修正や機能追加に関わりのない修正も修正箇所として含まれている．

この問題を解決するためには，修正箇所を人が見て，その修正が考慮すべきものか否かを判別する必要があるため，完全にこの問題による影響を取り除くことは現実的に困難である．

重複コード検出ツールの設定

本論文で用いたすべての重複コード検出ツールは，最小コード片の大きさや変数名の違い

を吸収するか否かなど，様々な設定を変更することが可能である．設定を変更することで検出される重複コードに違いが生じるため，設定を変更して実験を行うと今回の結果とは異なる結果が導かれる可能性がある．しかし，今回使用した重複コード検出ツールは変更可能な設定が非常に多く，たとえば *CCFinder* の場合は 15 の設定が変更可能となっている．このため，すべての設定を網羅して実験を行うことは現実的に困難であると判断し，本論文ではそれぞれの検出ツールのデフォルトの設定を用いて調査を行った．

実験対象

重複コードの割合はソフトウェアのドメインによって異なることが報告されている[16]．このため，今回の調査において対象ソフトウェアに含まれていないドメインのソフトウェアに対して計測を行うと，今回の結果とは異なる結果となる可能性がある．この課題点の改善には，より多くのソフトウェアに対する検証が必要であるといえる．

また，オープンソースとして成功せず，短命に終わったプロジェクトについては，長く継続しているプロジェクトと比較して保守性が低い可能性がある．本論文では実験対象の選択時に各プロジェクトの継続期間を考慮していないため，対象プロジェクトは様々な継続期間のものが存在する．しかし，現時点で短い期間しか継続していないプロジェクトでも，今後そのプロジェクトが長く継続する可能性がある．このため，成功しなかったプロジェクトに対して計測を行うと今回得られた結果と異なる結果が導かれる可能性がある．しかし，プロジェクトが成功したか否かを判別する画一的な方法は存在せず，また成功しなかったプロジェクトのソースコードを入手することは困難であるため，成功しなかったプロジェクトに対して実験を行うことは困難である．

6. あとがき

本論文では重複コードがソフトウェアの修正作業量に与える影響を，重複コードが非重複コードと比較して修正されやすければ重複コードが修正作業量を増大させているという考えに基づき調査した．既存研究の課題点を改善するために新たな指標を定義し，15 のオープンソースソフトウェアに対して実験を行った．その結果，重複コードは非重複コードと比較して修正が加えられにくく，重複コードが修正作業量を増大させているとは必ずしもいえないという結果を得た．

今後の課題として，修正箇所ごとに要する修正作業量の違いを考慮に入れた計測や，重複コード検出ツールの設定を変化させての計測，自動生成されたコードなどを除外した検証などがあげられる．

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技術の開発普及）の委託に基づいて行われた．また，日本学術振興会科学研究費補助金基盤研究（A）（課題番号：21240002）および（C）（課題番号：20500033），文部科学省科学研究費補助金若手研究（B）（課題番号：22700031）の助成を得た．

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) Kamiya, T., Kusumoto, S. and Inoue, K.: CCFinder: A multi-linguistic token-based code clone detection system for large scale source code, *IEEE Trans. Softw. Eng.*, Vol.28, No.7, pp.654–670 (2002).
3) CCFinderX, available from ⟨http://www.ccfinder.net/ccfinderx-j.html⟩.
4) Simian-Similarity Analyser, available from ⟨http://www.redhillconsulting.com.au/products/simian/⟩.
5) Higo, Y. and Kusumoto, S.: Enhancing Quality of Code Clone Detection with Program Dependency Graph, *Proc. 16th Working Conference on Reverse Engineering*, pp.315–316 (2009).
6) Johnson, J.: Substring matching for clone detection tools, *Proc. International Conference on Software Maintenance '94*, pp.120–126 (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) Krinke, J.: Is Cloned Code more stable than Non-Cloned Code?, *8th IEEE International Working Conference on Source Code Analysis and Manipulation*, pp.57–66 (2008).
9) Lozano, A. and Wermelinger, M.: Assessing the effect of clones on changeability, *International Conference on Software Maintenance*, pp.227–236 (2008).
10) Göde, N.: Evolution of Type-1 Clones, *9th IEEE International Working Conference on Source Code Analysis and Manipulation*, pp.77–86 (2009).
11) Göde, N. and Kosheke, R.: Incremental clone detection, *Proc. 13th European Conference on Software Maintenance and Reengineering*, pp.219–228 (2009).
12) Lozano, A., Wermelinger, M. and Nuseibeh, B.: Evaluating the harmfulness of cloning: A change based experiment, *IEEE 4th International Workshop on Mining Software Repositories*, pp.19–20 (2007).
13) Clone Tracker, available from ⟨http://mcs.open.ac.uk/alr242/CloneTracker.htm⟩.
14) diff-win, available from ⟨http://www.gfd-dennou.org/library/cc-env/diff-win/SIGEN.htm⟩.

15) 阿萬裕久，望月尚美，山田宏之，野田松太郎：クラスサイズメトリクスを用いたソフトウェア変更量予測に関する考察，ソフトウェアテストシンポジウム 2004 (2004).
16) Uchida, S., Monden, A., Ohsugi, N., Kamiya, T., Matsumoto, K. and Kudo, H.: Software Analysis by Code Clones in Open Source Software, *J. Computer Information Systems*, Vol.XLV, No.3, pp.1–11 (2005).

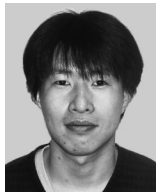
(平成 22 年 10 月 20 日受付)
(平成 23 年 6 月 3 日採録)

堀田　圭佑

平成 21 年大阪大学基礎工学部情報科学科卒業．現在，同大学大学院情報科学研究科博士前期課程在学中．コードクローン分析に関する研究に従事．

佐野由希子

平成 19 年大阪大学基礎工学部情報科学科卒業．平成 21 年同大学大学院情報科学研究科博士前期課程修了．在学時，コードクローン分析に関する研究に従事．



肥後　芳樹（正会員）

平成 14 年大阪大学基礎工学部情報科学科中退．平成 18 年同大学大学院情報科学研究科博士後期課程修了．日本学術振興会特別研究員を経て，平成 19 年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助教．博士（情報科学）．コードクローン分析，リファクタリングに関する研究に従事．IEEE 会員．

楠本　真二（正会員）

昭和 63 年大阪大学基礎工学部情報工学科卒業．平成 3 年同大学大学院博士課程中退．同年同大学基礎工学部情報工学科助手．平成 8 年同学科講師．平成 11 年同学科助教授．平成 14 年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助教授．平成 17 年同専攻教授．博士（工学）．ソフトウェアの生産性や品質の定量的評価，プロジェクト管理に関する研究に従事．IEEE，JFPUG 各会員．