NAIST-IS-MT0851080

# 修士論文

# ソースコードの改版履歴情報を利用したコードクローンの有害性に関する分析

西田 皓司

2010年2月4日

奈良先端科学技術大学院大学
情報科学研究科 情報システム学専攻

本論文は奈良先端科学技術大学院大学情報科学研究科に
修士 (工学) 授与の要件として提出した修士論文である。

西田 皓司

審査委員：

飯田 元 教授　（主指導教員）
松本 健一 教授　（副指導教員）
関 浩之 教授　（副指導教員）
川口真司 助教

# ソースコードの改版履歴情報を利用したコードクローンの有害性に関する分析*

西田 皓司


## 内容梗概

本論文では，ソースコード中に存在する重複コード列であるコードクローンのソフトウェアにおける有害性について明らかにすることを目的として，コードクローンが開発の過程でどう変化してきたのかを示すコードクローンの履歴情報に着目し，コードクローンに関する分析を行った．

分析では，ソフトウェアのあるバージョンにおいて，互いにコードクローンの関係にあるコード片全てに一貫していない変更を行った場合に生じると考えられる，「クローンの分岐」と「バグの発生」との関連性についての仮説を設けた上で，仮説の実証を試みた．

分析の結果，分岐を含まない場合と比べて，分岐を含む場合の方がバグが発生する傾向にあり，その数も多いことが分かった．この結果から，クローンの分岐は，その後のバグの発生に影響しており，そのようなクローンに対しては特に注意を払うことが有効であることを確認した．



## キーワード

コードクローン，改版履歴情報，コードクローンの分岐，コードクローンの有害性，版管理システム


---

*奈良先端科学技術大学院大学 情報科学研究科 情報システム学専攻 修士論文，NAIST-IS-MT0851080, 2010 年 2 月 4 日.

# An Analysis of Harmful Code Clone using A Change History*

Koji Nishida

**Abstract**

In this paper, for revealing harmfulness of code clone which is duplicated code in the source code, we analyzed code clone focusing on change history which represent how code clone is changed in the development process. In the analysis, we made a hypotheses of the relation between ”divergence of the code clone”, generated by inconsistent change to clone group, and ”occurrence of the bug”. Then, we carried out an experimentation using Eclipse development data.

The result shows that code clone containing the divergence tends to have more occurrence of the bug. From this result, we made a conclusion that divergence of the code clone influenced the occurrence of the bug afrterwards, and we need to pay attention to such clone.

**Keywords:**

Code clone, Change history, Divergence of code clone, Harmfulness of code clone, Version management system

*Master's Thesis, Department of Information Systems, Graduate School of Information Science, Nara Institute of Science and Technology, NAIST-IS-MT0851080, February 4, 2010.

# 目次

**1. はじめに** **1**

**2. 関連研究** **3**

2.1. コードクローンに関する支援ツール . . . 3

2.2. コードクローンの分析 . . . 3

**3. コードクローンについて** **6**

3.1. コードクローンの定義 . . . 6

3.2. コードクローン検出方法 . . . 6

3.3. クローン履歴について . . . 8

3.4. 履歴情報の取得 . . . 10

3.5. クローンセットの分岐について . . . 11

3.6. クローンに関するバグ修正記録 . . . 11

**4. 仮説** **13**

4.1. 仮説 . . . 13

4.1.1. 履歴情報を用いるねらい . . . 13

4.1.2. コードクローンが分岐することとバグの発生との関係についての仮説 . . . 14

**5. ケーススタディ** **16**

5.1. 対象データ . . . 16

5.2. 分析手順 . . . 16

5.2.1. コードクローンの検出と履歴情報の取得 . . . 16

5.2.2. クローンセットヒストリーの分類 . . . 16

5.2.3. 有害性の分析 . . . 20

5.2.4. 結果 . . . 22

**6. 考察と分析結果の妥当性** **24**

6.1. 考察 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
6.2. 分析結果の妥当性について . . . . . . . . . . . . . . . . . . . . . . . . 24

**7. おわりに** **26**

**謝辞** **27**

# 図目次

1 例として用いるコード片 . . . . . . . . . . . . . . . . . . . . . . . . 8
2 トークンに分割したコード片 . . . . . . . . . . . . . . . . . . . . . . 8
3 トークン変換を行ったコード片 . . . . . . . . . . . . . . . . . . . . . 8
4 コードクローン検出の例 . . . . . . . . . . . . . . . . . . . . . . . . 8
5 クローン履歴 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
6 クローンの変化と分岐の例 . . . . . . . . . . . . . . . . . . . . . . . 12
7 仮説の説明 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
8 クローンセットヒストリーの分類手順 . . . . . . . . . . . . . . . . . . 17
9 分岐と判断する場合の例 . . . . . . . . . . . . . . . . . . . . . . . . 19
10 分岐の回数が多い経路でバグが生じるとは限らない例 . . . . . . . 25

# 表目次

1 クローンの検出結果と履歴情報の取得結果 . . . . . . . . . . . . . . 22
2 クローンセットヒストリーの分類結果 . . . . . . . . . . . . . . . . . 23
3 分岐の有無によるバグ混入率の比較結果 . . . . . . . . . . . . . . . . 23
4 バグの有無による分岐存在率の比較結果 . . . . . . . . . . . . . . . . 23
5 分岐の有無による平均バグクローンの個数の比較結果 . . . . . . . . 23

# 1. はじめに

近年，ソフトウェアの大規模化と運用の長期化に伴い，ソフトウェア開発工程における保守工程の占める割合は年々増加している．保守工程における大きな問題の1つとして，ソースコード中に含まれる重複したコード列であるコードクローンの存在が挙げられる．例えば，コードクローンを含むソースコードに不具合が見つかった場合，関係する他の全てのコードクローンについて，同様の修正が必要か調査した上で，必要な箇所には修正を施し，さらに修正箇所が正しく動作するかを再テストしなくてはならないためである．

このような背景から，ソースコード中のコードクローンを見つけ出し，積極的に除去すべきであるということが多く言われている．しかし，すでに存在するコードクローンを除去するための修正作業には多大なコストがかかる上に，新たなバグが混入し，逆にソフトウェアの信頼性の低下を招く恐れがある．また，熟練したプログラマーが意図的にコードクローンを生成するというような，コードクローンを肯定的に利用する場合もある [17]．つまり，全てのコードクローンが問題とは一概に言い切れないため，検出されたコードクローンの中から，除去すべき・注意すべきコードクローンだけを見極めることができなければ，コードクローンに対する保守作業の効率が低下してしまう．

本論文では，除去すべき・注意すべきコードクローンとして，「互いにコードクローンの関係にあるコード片全てに一貫した変更が行われずに一部のみ変更された」コードクローンに注目した．このような変更が行われると，元々1種類のコードクローン群だったものが，2種類の別々のコードクローン群として検出されるようになる．このとき，もしどちらか一方のコードクローン群にバグなどの修正すべき箇所が見つかった場合，もう片方のコードクローン群にも同じソースコードが多く含まれており，同じ修正を必要とする可能性が高くても，2つのコードクローン群がすでに別種のコードクローン群として認識される．そのため，このような部分を特定することは，コードクローン検出ツールを利用しても発見が困難となる．

そこで，本研究では，コードクローンが開発の過程でどう変化してきたのかを示すコードクローンの履歴情報に着目することで，このようなコードクローンを判別

し，有害性について調査することを目的とする．履歴情報には，バージョン間でのコードクローンの関係や，いつどのコードクローンに対して変更が行われたのか，いつどのコードクローンで問題が生じたのかといった様々な情報が含まれる．よって，履歴情報を用いることで，一貫した変更が行われなかったコードクローンを判別し，そのようなコードクローンが問題を引き起こしているのかを調査することができる．

このような一貫していない変更が問題を引き起こすことを確かめるために，履歴情報を用いて以下の仮説を実証する分析を行った．

「あるバージョンにおいて，互いにコードクローンの関係にあるコード片全てに一貫した変更を行わずに一部のみを変更した場合，その後，それらのコード片について，バグが発生しやすくなる．」

以降，2 節では関連研究を説明する．3 節ではコードクローンの定義や検出方法，コードクローンの履歴情報について詳しく説明する．4 節では仮説について説明する．5 節ではケーススタディの手順と結果についてを説明する．6 節では考察を述べる．最後に 7 節でまとめについて述べる．

# 2. 関連研究

## 2.1. コードクローンに関する支援ツール

Kamiya らは，接尾辞木検索アルゴリズムを用いた検出手法を提案し，検出ツール CCFinder X を開発している [7]．CCFinder X は，広く用いられている複数のプログラミング言語に対応している．

山科らは，コードクローンの修正を支援するツール SHINOBI を開発している [28]．SHINOBI は，自動的にコードクローンを検出し，修正中のコードと関係が強い順にランキングして表示を行うことで，コードクローンに対する保守作業を容易にする．

肥後らは散布図によってコードクローンの可視化するためのツール Gemini を開発している [27]．Gemini は，メトリクスによるフィルタリングにより，調査の必要がないコードクローンを他のコードクローンと異なって表示することにより，それらを除外して，効率的にコードクローンの分析作業を行うことができる．

Adar らは，複数のバージョンからコードクローンを検出して，その遷移の様子を表示するのツール GoftGUESS を開発している [1]．このツールを使うことによって，バージョンを通じて安定しているコードクローンや，異なる修正が加わった結果，コードクローンではなくなったものなど，コードクローンの状態の変化を容易に知ることができる．

## 2.2. コードクローンの分析

Kim らは，一般的に言われている「コードクローンは有害である」ということの妥当性を実証することを目的として，クローンの履歴情報に着目した分析を行っている [18]．分析では，長期間に渡ってクローンがどのように変化していくのかを調査し，変化のパターンによる分類を行った．その結果，長期間一致した変更を加えられてきたクローングループや，言語の制約によって取り除くことができないコードクローンが，多く存在し，これらはリファクタリングによって取り除くのが困難なコードクローンであるとしている．一方で，短期間しか存在しないコードクロー

ンに対しては，リファクタリングの必要がないとしている．この研究は，履歴情報に着目しているという点では我々の研究と似ているが，リファクタリングの可否でコードクローンの分類を行っているという点で，異なっている．

Krinke は，コードクローンに対する変更は，一致した変更であるという仮説のもと，５つのプロジェクトに対して実験を行い，コードクローンに対する一致した変更の割合について調査を行っている [20]．その結果，一致した変更は 50 パーセント程度で，一般的に言われているよりも少なかったと結論付けている．また，一致した変更が行われなかったものに関しては，そのほとんどが独立したまま，バージョンを重ねていくとしている．この研究も我々の研究と同様，過去の履歴情報を用いた分析を行っているが，コードクローンが与えるソフトウェアへの影響性については言及していない．

Juergens らは，コードクローンの存在が実際のソフトウェアに悪影響を及ぼしているのかを調べるために，5 つのプロジェクトに対してケーススタディを行った [14]．ケーススタディでは，クローンセット中のクローンに対して一貫した変更を行っていないケースがどの程度あるのか，更にそれが意図したものなのかということについて分析を行っている．結果，一貫しない変更が多く存在し，それが意図しないものであった場合は，問題が発生しやすいということを示した．この研究は，コードクローンに対する変更の一貫性に着目し，コードクローンがソフトウェアに及ぼす悪影響について述べている点で，我々の研究と類似しているが，影響性に関する判断に際して，明確な基準を設けておらず，我々の研究よりも定性的な分析であるといえる．

Kapser らは，コードクローンがソフトウェアに与える有害性を検証することを目的とし，ケーススタディによって，ソースコードの意味，コードクローンを生成した理由，どのように生成されたのかという観点でコードクローンの分類を行った [17]．分類の結果，コードクローンを４グループに分類し，それぞれについて，メンテナンスのしやすさという観点で評価を行い，プロジェクトやトークン長の違いによって，33% から 71% の範囲で，メンテナンス上良い影響を与えているとした．この研究は，コードクローンの有害性に関する分析という点で我々の研究と類似しているが，メンテナンスのしやすさという基準で有害性の判断を行っている点で，我々の研究とは異なる．

左藤らは，コードクローンの長さや，広がり，複雑度という３つのメトリクスに基づき，コードクローンとソフトウェアの信頼性との関係について調査を行っている [26]．調査の結果，コードクローンの種類によって，信頼性を向上させる性質を持つコードクローンと信頼性を低下させるコードクローンに分類している．この研究は，定量的な分析により，コードクローンを評価しており，その点で我々の研究と共通している．しかし，評価の基準や，履歴情報に着目していないという点で我々の研究とは異なっている．

# 3. コードクローンについて

本節では，本論文での分析対象であるコードクローンの定義と，その検出方法について述べる．そして，コードクローンの有害性を評価するために本論文が着目しているコードクローンの履歴，およびその取得方法について述べる．最後に，コードクローンへの一環しない変更をモデル化した概念であるコードクローンの分岐について述べる．

## 3.1. コードクローンの定義

コードクローンとはソースコード中の重複したコード列のことである．これはソフトウェアの開発中や保守作業中に，次の要因によって生じる [6]．

- コード列のコピー＆ペースト
- コードジェネレータによるコード生成
- 特定のコーディングスタイルの利用
- パフォーマンスを向上させるための意図的なコード記述の繰り返し
- 偶然の一致

コピー＆ペーストによって作られた場合には，コピー後のコード列に部分的な変更を加えることも多く，一般的にそのような変更を加えられた類似コード列の組も，コードクローンと見なす．コードクローンの多くはデバッグや機能拡張の際に同時に更新しなければならず，ソフトウェアの保守工数を増大させている．このため，保守工程においてはコードクローンを検出することが望ましい．ただし，どのようなコード列をコードクローンと見なすかの定義は，検出方法やツールごとに異なる [2][3][4][5][6][9][15][16][12][13][19][21][22]．

## 3.2. コードクローン検出方法

コードクローン検出方法はいくつか提案されているが，本節では，神谷らが提案したトークンベースのコードクローン検出方法について述べる [15][16]．この方

法はプログラミング言語の構文規則に基づき，ソースコードをトークン列に変換し，比較する．なお，トークンとは，プログラム言語上における最小語を表す．これによって，コード列中の空白やコメント，インデントが異なっている場合でも，コードクローンを抽出することができる．また，神谷らの方法では，関数名，変数名など識別子の差異を故意に無視することで，完全一致するコードだけでなく，類似したコードの抽出も可能にしている．この方法を実装したツール CCFinderX[7] は，Java や C++ 等のプログラミング言語で記述されたソフトウェアからコードクローンを検出できる．図 1 に示すコード片を例として，トークンベースのコードクローン検出手順の概要を次に示す．

(1) コードクローン検出対象のソフトウェアに含まれる全てのソースコードを，プログラミング言語の構文規則に従ってトークンに分割する．ソースコード中の空白やコメントは無視される．図 2 にソースコードをトークンに分割した例を示す．
(2) 型，変数，定数に属するトークンは，型を t トークン，変数を p トークン，定数を i トークンに置き換える．図 3 にその例を示す．この置き換えにより，型名，変数名，定数だけが異なるコード列の組をコードクローンとして検出することができる．
(3) 変換後のトークン列に含まれる全ての部分列の集合から，同一の部分列の組を探し出してコードクローンとして検出する．図 4 の例では，黒線の四角で囲ったコード片をコードクローンとして検出している．検出にあたっては Suffix Tree マッチングアルゴリズム [11] を用いることで，ソフトウェアのサイズ n に対して O(n) の計算量で検出できる．最後に，検出されたトークン列の位置情報を元のソースコード上の行番号に変換し，コードクローンとして出力する．

```
----
y = 0 ;
x = y – z ;
if ( x > 0 ) n = 1 ;
z = 0 ;
----
```
ファイルAに含まれるコード片1

```
----
}
x = b – c ;
if ( x > 0 ) n = 0 ;
while ( b > 0 ) {
----
```
ファイルBに含まれるコード片2

図 1 例として用いるコード片

```
----
y = 0 ;
x = y – z ;
if ( x > 0 ) n = 1 ;
z = 0 ;
----
```
ファイルAに含まれるコード片1

```
----
}
x = b – c ;
if ( x > 0 ) n = 0 ;
while ( b > 0 ) {
----
```
ファイルBに含まれるコード片2

図 2 トークンに分割したコード片

```
----
p = i ;
p = p – p ;
if ( p > i ) p = i ;
p = i ;
----
```
ファイルAに含まれるコード片1

```
----
}
p = p – p ;
if ( p > i ) p = i ;
while ( p > i ) {
----
```
ファイルBに含まれるコード片2

図 3 トークン変換を行ったコード片

```
----
p = i ;
p = p – p ;
if ( p > i ) p = i ;
p = i ;
----
```
ファイルAに含まれるコード片1

コードクローン

```
----
}
p = p – p ;
if ( p > i ) p = i ;
while ( p > i ) {
----
```
ファイルBに含まれるコード片2

図 4 コードクローン検出の例

## 3.3. クローン履歴について

過去に存在したコードクローンが現在のソースコードのどこに対応するのかという情報をクローン履歴という．クローン履歴は，3.2 で示した方法により，あるバージョンでのコードクローンを検出しただけでは得ることができない．クローン履歴とクローン履歴の必要性について，図 5 を用いて説明する．

図 5 は，バージョン $V_i$ とバージョン $V_{i+1}$ のそれぞれにおいて，検出されたクローンを表している．本研究では，$a_1$，$a_2$，$a_3$，$a_4$ それぞれを 1 つのクローンと呼ぶ．また，$(a_1, a_2)$ のようにクローン検出ツールによって類似コードと判定されたクローンを，互いに「クローン関係」があると呼ぶ．そして，互いにクローン関係にあるクローンの集合を「クローンセット」と呼ぶ．また，図中の点線は，クローン履歴を表しており，クローン履歴によって結ばれる 2 つのクローンを，お互いに「履歴関係」にあると呼ぶ．

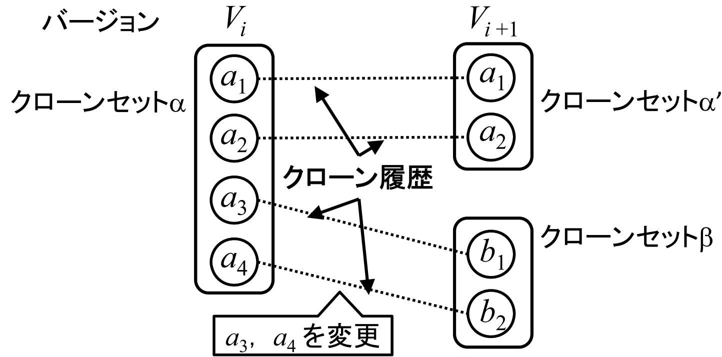


図 5　クローン履歴

本研究では，クローン履歴および履歴関係の定義として，川口ら [25] が提案しているクローン履歴関係の定義を用いた．川口らは，クローン履歴関係として，以下の $HC_t$, $HT_t$, $HA_t$ の和集合として履歴関係を定義している．

1. $HC_t$: $(a, a')$ が $F_{t-1}$ に関するクローンペア，$(b, b')$ が $F_t$ に関するクローンペアで，$b$ が $a$ の，$b'$ が $a'$ の子となる $a', b'$ が存在する．
2. $HA_t$: $a$ は $F_{t-1}$ に関するクローン，$b$ は $F_t$ に関するクローンで，$b$ と 1 行以上重複する位置にある $b' \in Change_{\Delta t}$ について $(a, b')$ が $\{F_{t-1} \cup Change_{\Delta t}\}$ についてのクローンペアである．ただし，$Change_{\Delta t}$ は $F_t$ のコード片の集合のうち $F_{t-1}$ からの変更時に追加，編集された部分，$\{F_{t-1} \cup Change_{\Delta t}\}$ は $F_{t-1}$ 全体に $Change_{\Delta t}$ のコード片を追加した部分とする．
3. $HT_t$: $a$ は $F_{t-1}$ に関するクローン，もしくは $F_{t-1}$ のコード片で $\exists x, (x, a) \in HT_{t-1}$ ．$b$ は $a$ の子で $a$ と $b$ のテキスト類似度が高い

次にクローン履歴の必要性について述べる．図 5 でバージョン $V_i$ の時点でクローンセット $\alpha$ のクローン $a_3$，$a_4$ に対して変更を加えた結果，バージョン $V_{i+1}$ では $a_3$，$a_4$ はクローンセット $\beta$ となる．このとき，クローンセット $\alpha'$ とクローンセット $\beta$ は関連性がある可能性が高いが，バージョン $V_{i+1}$ において 2 つは別のクローンセットとして検出されてしまう．そのため，これ以後，一方で変更を加えた場合，もう一方のクローンセット内のクローンも変更する必要があるにもかかわら

ず，見逃してしまいやすくなってしまう．クローン履歴を用いることで，このような 2 つのクローンセットを関連性のあるものとして判別することが可能となる．

次節では，クローンの履歴関係を取得する方法について説明する．

## 3.4. 履歴情報の取得

クローンの履歴関係を取得するためには，まず過去のソースコードが必要である．過去のソースコードを得るためには，版管理システムを用いる．版管理システムとは CVS[8] や Subversion[23] のようにプロダクトの保管・管理に用いられるシステムである．版管理システムでは，管理下のプロダクトを任意の時点の状態に復元して取得する機能が提供されている．

次に，版管理システムを用いて取得した過去のソースコードそれぞれに対して，3.2 節で記述した方法により，コードクローンの検出を行い，その結果に対して以下に記述する手法 [25] によって，履歴情報を取得する．

クローン履歴解析を適用する期間の始まりを時刻 0，終わりを時刻 $T$ で表し，$C_t$ を $F_t$ に関するクローンの集合，$CT_t$ を $HT_t$ を満たすコード片のうち $F_t$ に属するコード片の集合とする．

(0) $H_0 = \emptyset$, $CT_0 = \emptyset$.

(1) $t = 0$ の $F_0$ を版管理システムから取得し，クローン解析手法を用いて $C_0$ を求める．

(2) for $t = 1, 2, ..., T$

(2-a) 版管理システムから $F_t$ を取得し，クローン解析手法を用いて $C_t$ を求める．

(2-b) $F_{t-1}$ に関するすべてのクローンとそれぞれの子について，$HC_t$ を求める．

(2-c) $\{F_{t-1} \cup Change_{\Delta t}\}$ に対して，CCFinder を適用して，$HA_t$ を求める．

(2-d) $C_{t-1}$ の残りと $CT_{t-1}$ について，$HT_t$ を求める．

このように，本手法では解析を行う期間 $[0...T]$ を $\Delta t$ ごとに区切って考え，クローン履歴対応関係 $H_1, H_2, ..., H_T$ を順次，抽出する．

### 3.5. クローンセットの分岐について

3.3 節で記述したように，クローンセット中の一部のコードクローンにだけ，変更を加えることによって，次バージョンでは，クローンセットが 2 つ，もしくは 3 つ以上の独立したクローンセットとして存在することがある．本論分では，このように，あるバージョンでは 1 つだったクローンセットが，2 つ以上になる現象を，「分岐」と呼ぶ．

図 6 は，クローンセットが，変化・分岐する様子を示している．詳細を以下に記す．

- バージョン $V_{i-1}$ において，クローン $a_1$，$a_2$ からなるクローンセット $\alpha$ が存在している．
- $a_1$，$a_2$ をコピー&ペーストすることにより，バージョン $V_i$ では，$a_1$，$a_2$，$a_3$，$a_4$ からなるクローンセット $\alpha$ となる．
- $a_3$，$a_4$ に対して変更を加えることにより，バージョン $V_{i+1}$ では，クローンセット $\alpha'$ とクローンセット $\beta$ に分岐する．

また，本稿では，クローン関係および履歴関係によってお互いに到達可能なクローンの集合を，1 つの「クローンセットヒストリー」と定義する．図 2 の例では，クローン $a_1$，$a_2$，$a_3$，$a_4$，$b_1$，$b_2$ は 1 つのクローンセットヒストリーに属する．

### 3.6. クローンに関するバグ修正記録

本研究では，有害なクローンか否かを，クローン履歴の中にバグが修正された記録を含んでいるかどうか，で判断する．

3.3 節で述べたクローン履歴は，コミット（版管理システムの管理下にあるファイルを更新すること）されたときに記録されるコミット情報との対応関係を保持している．さらに，コミット情報の中にバグへの対処記録があるかどうかを判定するために，Fischer らの手法 [10] を用いる．Fischer らの手法は，版管理システムに加えて Bugzilla や GNATS などのバグ管理システムを利用しているソフトウェアの開発記録を対象に，コミット情報に残されたバグへの対処記録を抽出する手法で

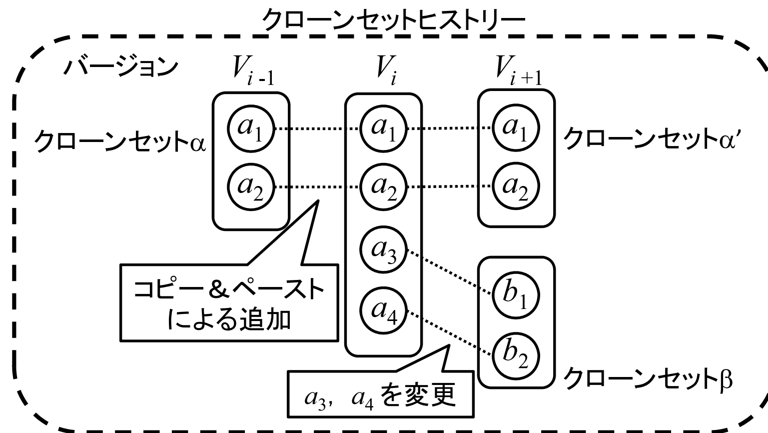


図 6　クローンの変化と分岐の例

ある．

本研究では，Fischer らの手法によって，コミット情報からバグの対処履歴を取り出すことが可能なソフトウェア開発記録を対象として分析を行う．

# 4.　仮説

## 4.1.　仮説

本節では，履歴情報を用いるに至った理由と，コードクローンの分岐とバグの発生との関係性についての仮説について説明する．

### 4.1.1.　履歴情報を用いるねらい

1 節で述べたように，コードクローンは一般的に除去すべきだとされているが，全てのクローンが問題だとは一概に言いきれない．以下に良いクローンと悪いクローンの例を示す．

良いクローンの例：

- コードジェネレータによって自動生成されたコード
- 長年運用されていて，バグは存在しないことが十分確認されているコード
- テンプレートを用いた定型的な処理を行うコード

悪いクローンの例：

- まだテストが行われていない，不具合の有無が確認されてないコード
- 修正変更が頻繁に行われているコード

このようにコードクローンにも様々な種類が存在するが，検出された膨大なコードクローンからこれらを見分けることは困難である．コードクローンに対する保守作業を効率的に行うためには，除去すべき・注意すべきコードクローンだけを見極める必要があるということがいえる．

本研究では，除去すべき・注意すべきコードクローンとして，一貫していない変更が行われたことにより分岐したコードクローンに注目した．3.3 で述べたように，コードクローンの分岐が生じた場合，それらのコードクローンを，分岐後において，互いに関係のあるコードクローンとして，検出することは難しい．そのため，分岐後のコードクローンのいずれかで問題が生じた場合に，分岐以前はコードクローンの関係にあったその他のコードクローンに対しての問題の可能性を見過ごしてしま

うということが起こり得るからである．そこで，このようなコードクローンを見分け，それらの有害性について調査するために，履歴情報に着目した．履歴情報には，あるバージョン内でのコードクローンどうしの繋がりだけでなく，バージョン間でのコードクローンの繋がり（クローン履歴）や，いつどのコードクローンに対して変更が行われたのか，いつどのコードクローンで問題が生じたのかといった様々な情報が含まれるため，履歴情報を用いることによって，以下の事が可能となると考えられるからである．

- 分岐がいつ生じたのかを判別できる
- 分岐以後も，関連性のあるコードクローンとして判別できる
- 分岐以後に，それらのコードクローンでバグが生じたかどうかを確認できる

#### 4.1.2. コードクローンが分岐することとバグの発生との関係についての仮説

これまでに述べたように，クローンの分岐は後々問題を引き起こす可能性がある．このことからから「あるバージョンにおいて，互いにクローンの関係にあるコード片全てに一貫した変更を行わずに一部のコード片のみを変更した場合，すなわち，分岐が生じた場合，その後，それらのコード片について，バグが発生しやすくなる」という考えのもと，以下の 3 つの仮説を立てた．

仮説 1 分岐を含むクローンセットヒストリーの方が，分岐を含まないクローンセットヒストリーよりもバグ修正を含む割合が高い (図 7(1) 参照)

仮説 2 バグ修正を含むクローンセットヒストリーの方が，バグ修正を含まないクローンセットヒストリーよりも分岐を含む割合が高い (図 7(2) 参照)

仮説 3 分岐を含むクローンセットヒストリーの方が，分岐を含まないクローンセットヒストリーよりも，より多くのバグ修正を含む (図 7(3) 参照)

そして，以上 3 つの仮説を実証するために，履歴情報を用いて分析を行った．次節では，分析についての詳細を説明する．

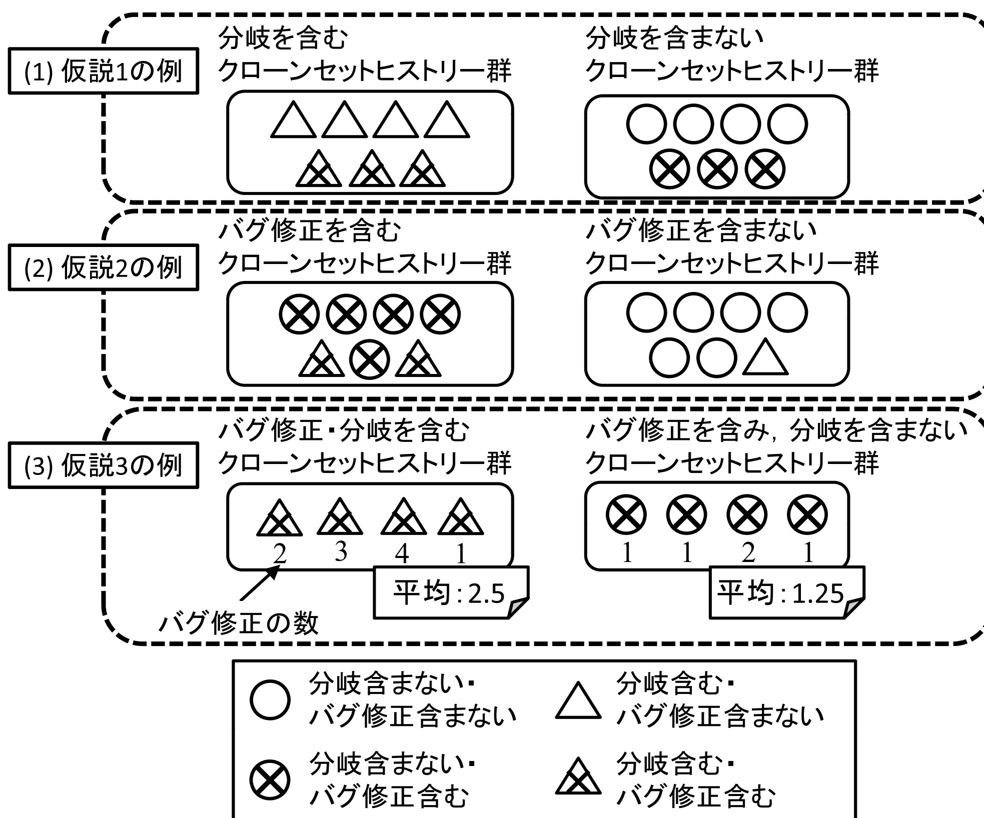


図 7 仮説の説明

# 5.　ケーススタディ

本節では，前節で示した仮説を実証するために，ケーススタディを行った．ケーススタディの手順，結果を以下に記す．

## 5.1.　対象データ

オープンソースの統合ソフトウェア開発環境 Eclipse のバージョン 2.0, 2.1, 3.0 のソースコード，およびその間の編集履歴情報・バグ情報を対象とし，分析を行った．編集履歴情報とバグ情報の対応関係に関しては，Zimmermann ら [24] によるデータを用いた．また，本実験では，一部のモジュール (org.eclipse.jdt.ui) を実験対象とした．

## 5.2.　分析手順

### 5.2.1.　コードクローンの検出と履歴情報の取得

3.2 節で述べたトークンベースによるコードクローン検出手法を採用している CCFinderX[7] を用いて，それぞれのリビジョンでのソースコードからコードクローンを検出した．検出の際の設定におけるトークン数は CCFinderX の初期値でである 50 を採用した．

得られた結果から，3.4 節で述べた方法により，履歴情報を抽出した．

### 5.2.2.　クローンセットヒストリーの分類

クローンセットヒストリー中に，コミットを含むか，バグ修正のあったコードクローンを含むか，分岐を含むか，バグが分岐の後で発生しているか，の観点で，クローンセットヒストリーを分類し，以下の A, B, C, D, E, F の 6 グループに分類した．

A バグ有り・分岐有り・分岐の後にバグ有り
B バグ有り・分岐有り・分岐の後にバグ無し
C バグ有り・分岐無し

D バグ無し・分岐有り

E バグ無し・分岐無し

F コミット無し

以下，A グループに分類されたクローンセットヒストリーのことを，クローンセットヒストリー a $\in$ A，A グループのクローンセットヒストリーの総数を $|A|$ と表記する．B, C, D, E, F についても同様の表記を行う．

分類は，以下に示す手順 (1), (2), (3), (4) の順で行った．また，図 8 に，分類手順を示す．

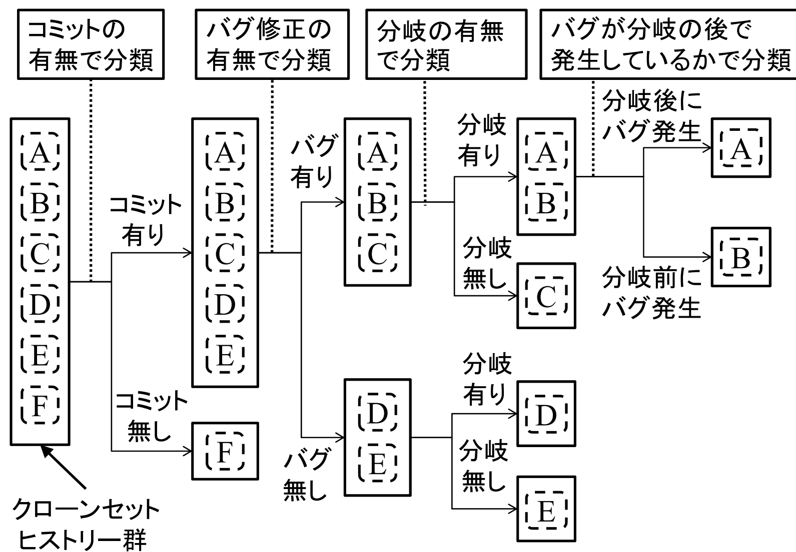


図 8 クローンセットヒストリーの分類手順

(1) 全てのクローンセットヒストリー群の中から，コミットのあったコードクローンを含むクローンセットヒストリーのみの抽出を行った．

(2) (1) で抽出されたクローンセットヒストリー群に対して，バグ修正を含むか含まないかで分類を行った．

(3) (2) で分類されたクローンセットヒストリー全てに対して，分岐を含むか含まないかでさらに分類を行った．

この分類の際，以下の 2 点の条件のいずれかを満たした場合に「分岐が有る」とした判断した．

条件 1: ある時点でクローンセットの数が 2 つ以上ある場合（図 6 の $V_{i+1}$ のような状態）．

条件 2: クローンとしては認識されていないが，クローンの可能性が高い場合 (詳細は後述)．

条件 2 は，分岐の特別な場合として単一のコードクローンにのみ変更が加えられたケースを想定した条件である．例を図 9 に示す．このように，クローンの集合のうち 1 つのクローンにだけ変更が加えられると，その部分とクローン関係になる箇所がなくなり，結果としてその部分はコードクローンとならなくなる．そのようなクローンの判定として，以下の 3 つの条件を全て満たしている場合に，条件 2 を満たしていると判断した．

(i) 隣り合うバージョン $V_i$, $V_{i+1}$ におけるクローンの数を $N_i$, $N_{i+1}$ とすると，$N_i > N_{i+1}$ である

(ii) バージョン $V_i$ において，コミットがあり，かつ，バージョン $V_{i+1}$ において履歴関係にあるクローンが存在しないようなクローンが 1 つ以上存在する

(iii) (ii) を満たすクローンについて，
(クローンの行数) $>$ (コミットによる削除行数) $* 0.30$
を満たすクローンが 1 つ以上存在する

(4) 分岐がバグに影響を与えているかを確認するために，(3) で得られたクローンセットヒストリーの中の，バグ有り・分岐有りのクローンセットヒストリーについて，更に，バグの発生時期が分岐の発生時期の前か後かで分類を行った．本研究では，分類するにあたってまず，バグの発生以前に，一つでも分岐が存在していれば，そのバグを「分岐の後に発生したバグ」，逆に，バグ発生以前に一つも分岐がない場合は，そのバグを「分岐の前に発生したバグ」としてカウントした．そして，「分岐の後に発生したバグ」の数が 1 個以上の場合を「分岐の後にバグ有り」，逆に「分岐の後に発生したバグ」の数が 0 個の場合を「分岐の後にバグ無し」として，クローンセットヒストリーを

分類した.

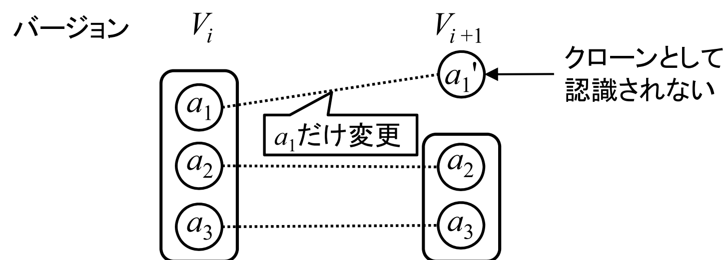


図 9　分岐と判断する場合の例

#### 5.2.3. 有害性の分析

5.2.2 の手順による分類の結果を用いて有害性の分析を行う．なお，本分析では，分岐がバグの発生に影響するかということについて確認したいため，B グループ「バグ有り・分岐有り・分岐の後にバグ無し」と F グループ「コミット無し」のクローンセットヒストリーは除外する．よって，分析で用いるグループは，以下の A, C, D, E の 4 グループである．

A バグ有り・分岐有り・分岐の後にバグ有り
C バグ有り・分岐無し
D バグ無し・分岐有り
E バグ無し・分岐無し

以上を用いて，前節で立てた仮説を実証するための分析を行った．以下にそれぞれの詳細を記す．

- **分岐の有無によるバグ混入率の比較 (仮説 1 に対する分析)**
  クローンセットヒストリー A, C, D, E を分岐の有無で分類し，バグを含むクローンセットヒストリーの割合を比較した．分岐有りと分岐無しのそれぞれについて，バグ修正を含むクローンセットヒストリーの割合 $p1$, $p_2$ を計算した．計算式は分岐有りの場合と無しの場合でそれぞれ，式 1，式 2 である．

$$p_1 = \frac{|A|}{|A| + |D|} \tag{1}$$

$$p_2 = \frac{|D|}{|C| + |E|} \tag{2}$$

さらに，この割合についての有意差検定を行った．以下の検定式（式 3）により，検定量 $Z_1$ を計算した後，この $Z_1$ の値を近似的に正規分布するものとみなし，正規分布表と比べ有意差があるかを判断した．

$$Z_1 = \frac{|p_1 - p_2|}{\sqrt{P_x(1 - P_x)(\frac{1}{|A|+|D|} + \frac{1}{|C|+|E|})}} \tag{3}$$

$$\left(P_x = \frac{p_1(|A|+|D|)+p_2(|C|+|E|)}{|A|+|D|+|C|+|E|}\right)$$

- **バグの有無による分岐存在率の比較 (仮説 2 に対する分析)**
  クローンセットヒストリー A, C, D, E をバグの有無で分類し，分岐を含むクローンセットヒストリーの割合を比較した．バグ修正を含む場合と含まない場合のそれぞれについて，分岐が有る割合 $p_3$, $p_4$ を計算した．計算式は分岐有りの場合と無しの場合でそれぞれ，式 4，式 5 である．

$$p_3 = \frac{|A|}{|A|+|C|} \tag{4}$$

$$p_4 = \frac{|D|}{|D|+|E|} \tag{5}$$

  さらに，この割合について，仮説 1 に対する分析と同様の方法により有意差検定を行った．検定量 $Z_2$ を求める検定式を式 6 に示す．

$$Z_2 = \frac{|p_3 - p_4|}{\sqrt{P_y(1-P_y)(\frac{1}{|A|+|C|}+\frac{1}{|D|+|E|})}} \tag{6}$$

$$\left(P_y = \frac{p_3(|A|+|C|)+p_4(|D|+|E|)}{|A|+|C|+|D|+|E|}\right)$$

- **分岐の有無による平均バグ修正数の比較 (仮説 3 に対する分析)**
  バグを含むクローンセットヒストリー，すなわち，クローンセットヒストリー A, C について，バグのあったクローンの個数の平均を比較した．クローンセットヒストリー A, C のそれぞれについて，バグのあったコードクローンの個数を数え，その平均個数を計算した．さらにこの平均個数については，マン・ホイットニーの U 検定により，有意差検定を行った．

#### 5.2.4. 結果

実験対象としたクローンセットヒストリーの数と，それらに含まれるコードクローンの数・コミットの有ったクローンの数・バグのあったクローンの数を表 1 に示す．これら 1473 個のクローンセットヒストリーに対して，5.2.2 で示した手順により分類を行った結果と，バグの有るクローンセットヒストリーについては，それぞれに含まれるバグ修正のあったクローンの数を，表 2 に示す．

以上の結果を用いて，5.2.3 で述べた手順に従い，前節で立てた 3 つの仮説を実証するための分析をそれぞれ行った．

1. **分岐の有無によるバグ混入率の比較（仮説 1 に対する分析）**
   結果を表 3 に示す．また，バグ有りの割合に関して，有意差検定を行った結果，有意水準５%という結果が出た．
2. **バグの有無による分岐存在率の比較（仮説 2 に対する分析）**
   結果を表 4 に示す．また，分岐有りの割合に関して，有意差検定を行った結果，有意水準５%という結果が出た．
3. **分岐の有無による平均バグ修正数の比較（仮説 3 に対する分析）**
   結果を表 5 に示す．また，平均個数に関して，有意検定を行った結果，有意水準１%という結果が出た．

表 1 クローンの検出結果と履歴情報の取得結果

| | |
|---|---|
| クローンセットヒストリーの数 | 1473 |
| コミットのあったクローンの数 | 11017 |
| バグのあったクローンの数 | 358 |

表 2　クローンセットヒストリーの分類結果

| | クローンセット<br>ヒストリーの数 | バグの有った<br>クローンの数 |
|---|---|---|
| A:バグ有り・分岐有り・<br>分岐後にバグ有り | 25 | 179 |
| B:バグ有り・分岐有り・<br>分岐後にバグ無し | 4 | 27 |
| C:バグ有り・分岐無し | 156 | 358 |
| D:バグ無し・分岐有り | 109 | - |
| E:バグ無し・分岐無し | 1179 | - |
| 合計 | 1473 | 564 |

表 3　分岐の有無によるバグ混入率の比較結果

| | クローンセット<br>ヒストリーの数 | バグ有り | バグ有り<br>の割合 | 検定量<br>$Z_1$ |
|---|---|---|---|---|
| 分岐有り | 134 | 25 | $p_1 = 0.19$ | 2.35 |
| 分岐無し | 1335 | 156 | $p_2 = 0.12$ | |

表 4　バグの有無による分岐存在率の比較結果

| | クローンセット<br>ヒストリーの数 | 分岐有り | 分岐有り<br>の割合 | 検定量<br>$Z_2$ |
|---|---|---|---|---|
| バグ有り | 181 | 25 | $p_3 = 0.14$ | 2.34 |
| バグ無し | 1288 | 109 | $p_4 = 0.08$ | |

表 5　分岐の有無による平均バグクローンの個数の比較結果

| | クローンセット<br>ヒストリーの数 | バグクローン<br>の個数 | 平均バグ<br>クローンの個数 | $p$ 値 |
|---|---|---|---|---|
| 分岐有り | 25 | 179 | 7.1 | 0.00 |
| 分岐無し | 156 | 358 | 2.3 | |

# 6.　考察と分析結果の妥当性

## 6.1.　考察

以上で示した 3 つの分析の結果から，それぞれ以下のことが分かった．

1. 分岐の有無によるバグ混入率の比較の結果，分岐有りのクローンセットヒストリーの方が，バグを含む割合が高かった
2. バグの有無による分岐存在率の比較の結果，バグ有りのクローンセットヒストリーの方が，分岐を含んでいる割合が高かった
3. 分岐の有無による平均バグ修正数の比較の結果，分岐を含むクローンセットヒストリーの方が，バグのあるコードクローンを平均的に多く含んでいた

1，2，3 は，それぞれ 4 節で立てた仮説 1，仮説 2，仮説 3 を実証できる結果だといえる．よって，「分岐が生じた場合，その後それらのコード片についてバグが発生しやすくなる」ということを，実証できたといえる．したがって，このようなコードクローンは，ソフトウェアにとって有害であり，注意が必要なコードクローンである可能性が高いといえる．この知見は，コードクローンに対する保守作業の際に，分岐を含むクローンセットヒストリーに含まれるコードクローンに注目することで注意すべきコードクローンを発見しやすくなるという点で有用であろう．さらに有害なコードクローンの一例をオープンソースリポジトリのデータから実際に示せたこと，また，コードクローンの履歴情報に着目することの有用性を確かめられたという点も本研究の成果であると言えよう．

## 6.2.　分析結果の妥当性について

本分析結果は，大規模なオープンソース開発での知見であり，コードクローンの数・コミットの数も多くあるため，妥当な結果と考えられる．また，分岐以前に生じたバグに関しては分析対象から排除し，3 つの側面で比較を行うことによって，分岐によるバグへの影響性に関して，より妥当な評価ができたと考えられる．

ただし，以下の点において，この結果の妥当性について議論の余地があると考え

られる．

- **分岐発生からバグ発生までの期間については考慮していない点**
  分岐がバグの発生に関与しているとするならば，分岐が生じたバージョンの直後，もしくはできるだけ近いバージョンにおいてバグが発生しているのが望ましい．しかし，今回の分析では，このような分岐発生とバグ発生の期間に制約を設けていない．
- **分岐の数がより多い経路で，バグが起こりやすいのかまでは確認していない点**
  今回の分析では，どの分岐の経路でバグが発生したのかということや，バグ発生に至るまでの分岐の回数を考慮していない．例えば，図 10 に示すように，分岐の回数が少ない経路でバグが発生している可能性が考えられる．
- **バグの発生と分岐との前後関係を決定する上での定義**
  バグ有り・分岐有りのクローンセットヒストリーに関して，バグ発生以前に一つでも分岐があれば，「分岐の後にバグがある」として分類したが，この分類の定義は問題があるかもしれない．例えば，バグ発生以前に分岐が 1 つあるが，バグ発生以後には分岐が 5 つあるといった場合があったとすると，これは，「分岐がバグの発生に関与している」とするのは難しいかもしれない．

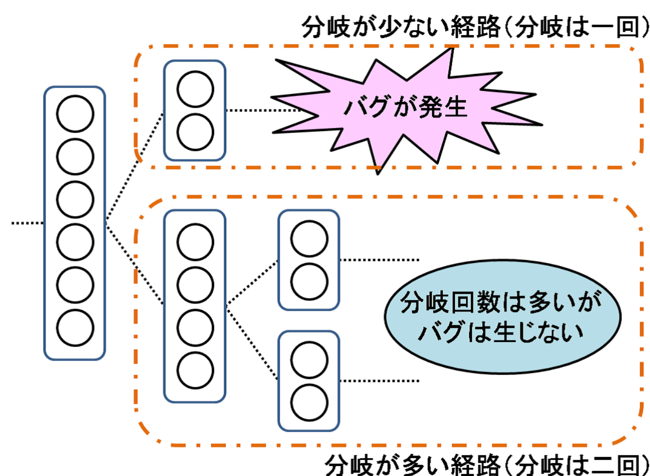


図 10　分岐の回数が多い経路でバグが生じるとは限らない例

# 7. おわりに

本研究では，有害なコードクローンの判別を支援するために，コードクローンの分岐とバグの発生との関係性についての分析を行った．分析の結果からは，

- 分岐の有無によるバグ混入率の比較の結果，分岐有りのクローンセットヒストリーの方が，バグを含む割合が高かった
- バグの有無による分岐存在率の比較の結果，バグ有りのクローンセットヒストリーの方が，分岐を含んでいる割合が高かった
- 分岐の有無による平均バグ修正数の比較の結果，分岐を含むクローンセットヒストリーの方が，バグクローンを平均的に多く含んでいた

ということが分かり，これらの結果から，分岐とバグ発生との間には関連性があると結論付けた．したがって，ソフトウェア開発・保守の際には，

- コードクローンの分岐，すなわち一部のコードクローンだけを編集することはできるだけ避けること．
- コードクローンの分岐が避けられない場合，その部分は有害なコードクローンと言えるので，今後保守の際にはそのような箇所を優先的に見るよう留意すること．

が重要であるといえる．

今後の課題としては，前節で述べた 3 点について確認を行い，分岐とバグの関係性をより明確にする必要がある．また，今回の分析では，一つのプロジェクトの一部のモジュールに限ったものであったので，分析の対象を拡大していく必要がある．さらに，分岐とバグの関係性という側面だけでなく，別の側面からも，コードクローンがソフトウェアに与える有害性についての評価を行うことも必要だと考えられる．

# 謝辞

本研究を進めるにあたり，多くの方々に御指導，御協力を頂きました．ここに感謝の意を表せて頂きたいと思います．本当にありがとうございました．

奈良先端科学技術大学院大学 情報科学研究科 ソフトウェア設計学講座 飯田 元 教授には，本研究の主導教員を担当して頂き，本研究の全過程において熱心な御指導を賜りました．研究方針だけではなく，研究に対する姿勢，論文執筆，発表方法についても多くの御助言を頂きました．心より厚く御礼を申し上げます．

奈良先端科学技術大学院大学 情報科学研究科 情報基礎学講座 関 浩之 教授には，副指導教員を担当して頂き，学内の発表において多数の御質問と御指導を頂きました．心より感謝申し上げます．

奈良先端科学技術大学院大学 情報科学研究科 ソフトウェア工学講座 松本 健一 教授には，副指導教員を担当して頂き，本研究の課題や方針などで的確な御助言，御指摘を頂きました．心より厚く御礼申し上げます．

奈良先端科学技術大学院大学 情報科学研究科 ソフトウェア設計学講座 川口 真司助教には，本研究を進めるに当たり，広範囲かつ多大な御助力を頂きました．心より感謝申し上げます．

日立製作所 名倉 正剛 氏には，本研究を進めるに当たり，貴重な御指導，御助言を頂戴いたしました貴重な御助力を頂きました．心より感謝申し上げます．

奈良先端科学技術大学院大学 情報科学研究科 ソフトウェア設計学講座 伏田 享平氏には，学会発表や論文投稿時に貴重な御助言・御協力を頂戴いたしました．心より感謝申し上げます．

奈良先端科学技術大学院大学 情報科学研究科 ソフトウェア設計学講座，ならびにソフトウェア工学講座の皆様には，日頃より多大な御協力と御助言を頂き，公私ともに支えていただきました．ありがとうございました．

最後に，日頃より私を励まし，暖かく見守ってくれた家族に心より深く感謝します．

# 参考文献

[1] E. Adar and M. Kim, “ Visualization and exploration of code clones in context, ” Proc. 29th Internationnal Conference on Software Engineering, pp.762-766, 2007.

[2] B. S. Baker, “ A program for identifying duplicated code, ”Proc. 24th Symposium on the Interface: Computing Science and Statistics, pp.49-57, 1992.

[3] B. S. Baker, “ On finding duplication and nearduplication in large software system, ” Proc. 2nd IEEE Working Conference on Reverse Engineering, pp.86-95, 1995.

[4] M. Balazinska, E. Merlo, M. Dagenais, B. Lague and K. A. Kontogiannis, “ Measuring clone based reengineering opportunities, ” Proc. International Symposium on Software Metrics, pp.292-303, 1999.

[5] M. Balazinska, E. Merlo, M. Dagenais, B. Lague and K. A. Kontogiannis, “ Paratial redesign of Java software systems based on clone analysis, ” Proc. 6th IEEE Working Conference on Reverse Engineering, pp.326-336, 1999.

[6] I. D. Baxter , A. Yahin, L. Moura, M. Sant ’ Anna and L. Bier, “ Clone detection using abstract syntax trees, ” Proc. International Conference on Software Maintenance, pp.368-377, 1998.

[7] CCFinderX, http://www.ccfinder.net/index-j.html.

[8] CVS, http://www.cvshome.org.

[9] S. Ducasse, M. Rieger and S. Demeyer, “ A language independent approach for detecting duplicated code, ” Proc. International Conference on Software Maintenance, pp.109-118, 1999.

[10] M. Fischer, M. Pinzger, and H. Gall, “Populating a release history database from version control and bug tracking systems,” Proc. International Conference on Software Maintenance, pp.23-33, 2003.

[11] D. Gusfield, “ Algorithms on strings, trees and sequences, ” Cambridge University Press, pp.89-180, 1997.

[12] J. H. Johnson, “ Substring matching for clone detection and change tracking, ” Proc, IEEE International Conference on Software Maintenance, pp.120-126, 1994.

[13] J. H. Johnson,“ Indentifying redundancy in source code using fingerprints, ” Proc. conference of the Centre for Advanced Studies on Collaborative research, pp.171-183, 1993.

[14] E. Juergens, F. Deissenboeck, B. Hummel and S. Wagner,“ Do Code Clones Matter?, ” Proc. International Conference on Software Engineering, pp.485-495, 2009.

[15] T. Kamiya, F. Ohara, K. Kondou, S. Kusumoto and K. Inoue, “ Maintenance support tools for Java programs:CCFinder and JAAT, ” Proc. 23th International Conference on Software Engineering, pp.837-838, 2001.

[16] 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, 2001.

[17] C. Kapser and M. Godfrey, “ ’Cloning Considered Harmful’ Considered Harmful:Patterns of Cloning in Software, ” Empirical Software Engineering, pp.645-692, 2008.

[18] M. Kim, V. Sazawal, D. Notkin and G. C. Murphy, “ Anempirical study of code clone genealogies, ” Proc. Joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp.187-196, 2005.

[19] K. A. Kontogiannis, R. De Mori, E. Merlo, M. Galler and M. Bernstein, “ Pattern matching techniques for clone detection and concept detection, ” Journal of Automated Software Engineering, Vol.3, pp.770-108, 1996.

[20] J. Krinke, “ A Study of Consistent and Inconsistent Changes to Code Clones, ” Proc. Working Conference on Rverse Engineering, pp.170-178, 2007.

[21] B. Lague, E.M. Merlo, J. Mayrand and J. Hudepohl,“ Assessing the benefits of incorporating function clone detection in a development process, ” Proc.

IEEE International Conference on Software Maintenance, pp.314-321, 1997.

[22] J. Mayland, C. Leblanc and E. M. Merlo, “ Experiment on the automatic detection of function clones in a software system using metrics, ”Proc. IEEE International Conference Software Maintenance, pp.244-253, 1996.

[23] Subversion, http://subversion.tigris.org/.

[24] T. Zimmermann, R. Premraj and A. Zeller, “ Predicting Defects for Eclipse, ”Proc. 3rd International Workshop on Predictor Models in Software Enginnering, pp.9, 2007.

[25] 川口真司, 松下誠, 井上克郎, “ 版管理システムを用いたクローン履歴分析手法の提案, ” 電子情報通信学会論文誌 D, Vol.J89-D, No.10, pp.2279-2287, 2006.

[26] 左藤裕紀，亀井靖高，上野秀剛，門田暁人，川口真司，名倉正剛，松本健一, 飯田元, “ コードクローンの長さとソフトウェア信頼性の関係の分析, ” 電子情報通信学会信学技報，Vol. 108，No. 242，pp. 43-48, 2008.

[27] 肥後芳樹, 吉田則祐, 楠本真二，井上克郎， “ 産学連携に基づいたコードクローン可視化手法の改良と実装, ” 情報学論, Vol.48, no.2, pp.811-822, 2007.

[28] 山科隆伸，上野秀剛，伏田享平，亀井靖高，名倉正剛，川口真司，飯田 元, “ コードクローンに着目したソフトウェア保守支援ツールの設計と実装, ” 電子情報通信学会技術研究報告，Vol. 108，No. 64，pp. 65-70, 2008.