**修士学位論文**

題目

# コード片の生存期間がコードクローンと欠陥修正の有無に与える影響分析

指導教員
井上 克郎 教授

報告者
齋藤 晃

平成 23 年 2 月 7 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 ソフトウェア工学講座

平成 22 年度 修士学位論文

# コード片の生存期間がコードクローンと欠陥修正の有無に与える影響分析

齋藤 晃

**内容梗概**

コードクローンとはソースコード中の全部又は一部の重複したコード片であり，今までに様々なコードクローンの検出手法が提案されている．コードクローンはプログラムの可読性を低下させ，ソフトウェアの保守を困難にするため積極的に除去すべきと言われている．このため，コードクローンを除去するためのリファクタリング手法が数多く提案されている．その一方，コードクローンはソフトウェアの欠陥に大きな影響を与えないという調査結果も存在する．既存の調査ではソフトウェア中のコードクローンに含まれる欠陥修正の割合を計測し，コードクローンが欠陥修正に与える影響を調べている．その結果欠陥修正によって変更されたコードにコードクローンが含まれている割合は小さく，コードクローンは欠陥を引き起こす要因に当たらないという結果が得られている．

しかしこの調査では，全ての欠陥コードが単にコードクローンを含むかどうかのみ判断しており，コードがそれまでにどの程度の期間存在していたかどうかについては考慮していない．我々は，コードクローンの分析においては，コード片がソフトウェア中に存在していた期間が重要であると考えている．なぜなら既に修正が頻繁に生じているコード片は今後も修正が生じやすいと考えられるからである．そこで本研究では，既存研究で用いられた手法に加えソフトウェア中に生存期間を考慮した上でコードクローンが欠陥コードを含む割合の調査を行う．その結果，生存期間が短いコードほど欠陥コードを含まれる割合が高く，生存期間が長いコードクローンはコードクローンでないコードよりも欠陥コードを含まれる割合が低くなることが分かった．

**主な用語**

リポジトリマイニング (Repository Mining)
コードクローン (Code Clone)
ソフトウェア進化分析 (Analysis of Software Evolution)

# 目 次

## 1　まえがき

コードクローンとはソースコード中の全部又は一部の重複したコード片であり，今までに様々なコードクローンの検出手法が提案されている [13][15]．コードクローンがソフトウェアに与える影響に関してさまざまな報告がある．Fowler らは重複したコード片が存在することを不吉な匂い (Bad Smell) のうちの 1 つと定義し，メソッドの抽出などにより重複を除去するべきと述べている [8]．またコードクローンはプログラムの可読性を低下させ，ソフトウェアの保守の観点で望ましくないために積極的に除去すべきだとも言われている [14]．このため，コードクローンを除去するためのリファクタリング手法が現在までに数多く提案されている [4][20]．

その一方，コードクローンはソフトウェアの欠陥に大きな影響を与えないという調査結果も存在する [9][16]．この調査はソフトウェア中の欠陥修正によって変更されたコード片を特定し，それらがどの程度コードクローンを含んでいるのかを調べている．その結果欠陥修正によって変更されたコードにコードクローンが含まれている割合は小さく，コードクローンは欠陥を引き起こす要因に当たらないという結果が得られている．

しかしこの調査では，全ての欠陥コードが単にコードクローンを含むかどうかのみ判断しているだけであり，コードがそれまでにどの程度の期間存在していたかどうかについては考慮していない．

我々は，コードクローンの分析においては，コードクローンに包含されるコード片が作成されてから削除されるまでの期間 (コード片の生存期間と定義する) が重要であると考えている．その理由として，既に修正が頻繁に生じているコード片は今後も修正が生じやすいと考えられること，また，たとえコードクローンであってもよく知られたプログラミングロジックや長期間使われている実績のあるコードであれば，欠陥修正との関連は小さくなると考えられることが挙げられる．

そこで本研究では，既存研究で用いられた手法に加えコードの生存期間を考慮した上でコードクローンが欠陥コードを含む割合の調査を行う．本研究では 2 種類の調査を行い， 1 つ目の調査では欠陥コードを含むコードクローンを生存期間が長い集合と生存期間が短い集合の 2 つに分け，それぞれにおいて欠陥コードが含むコードクローンの割合を調べた． 2 つ目の調査では，コードを欠陥修正が生じたものとそうでないものに分割し，それぞれをさらにコードクローンであるものとコードクローンでないものに分割し，計 4 つの集合に分割を行う．さらにこれらの集合において生存期間に差異があるかどうかを調べた．

1 つ目の調査の結果では，生存期間が短い欠陥コードはより多くのコードクローンを含み，逆に生存期間が長い欠陥コードはコードクローンを含む割合が小さい傾向が得られた．また 2 つ目の調査では，コードクローンとなるコードとそうでないコードの両方で，生存期間が

短いコードほどより多くの欠陥コードを含んでいる傾向が得られた．また生存期間の長いコードクローンは，コードクローンでないコード片に比べ欠陥コードの含む割合が小さくなる傾向が得られた．

本論文の構成は以下の通りである．まず，2 章で研究の背景としてコードクローン，欠陥コードの検出を述べる．次に 3 章で既存手法の調査について説明する．4 章では本研究で実施した 2 つの調査内容について説明する．5 章では実施した調査の結果報告を行い，6 章で考察を行う．その後 7 章で本調査の妥当性について説明し，8 章で関連研究について述べ，最後にまとめを述べる．

## 2　背景

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作業が重要視されている．ソフトウェア保守作業の効率に影響を与える要因の 1 つとして，コードクローンの有無が議論されている．本章では，研究背景としてコードクローンが欠陥修正に与える影響を調査した既存研究と，それらに関連する用語の説明を行う．

### 2.1　コードクローン

コードクローンとはソースコード中の全部または一部の重複したコード片のことを指す．コードクローンが生成される要因は様々な理由が挙げられているが，主な理由は以下のようなものである [5][18][21].

**既存コードのコピーペーストによる再利用**　オブジェクト指向設計などのソフトウェア設計手法を利用すれば，部品の再利用が可能となる．しかし，ゼロからソースコードを書くよりも既存コードをコピーして部分的な変更を加える方が信頼性が高いということもあり，実際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存在する．

**プログラミング言語に固有する問題**　プログラミング言語の特性上の問題で 1 箇所にまとめて記述することができない場合，コードクローンとして複数箇所に現れる．例えばプログラミング言語 Java の場合，例外処理は 1 箇所にまとめて記述することができず，複数箇所に同様の処理を記述する必要がある．

**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．

コードクローンはソースコードに対して一貫した変更を加えることを難しくするため，ソフトウェアの保守効率を低下させると言われている [8]．そのため，次節で紹介するコードクローン検出手法を用いてコードクローンを検出し，コードクローンを統合する手法が現在までに提案されている [22]．その一方 3 章で述べるように，コードクローンが欠陥修正に及ぼす影響を調べ，コードクローンはソースコードの欠陥修正とは影響しないという結果を報告した事例も存在する．

## 2.2 コードクローン検出

現在までにソースコード中からコードクローンを検出するための様々な手法が提案されている [13][15][21]．既存の検出手法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．

- 行単位の検出
- 字句 (トークン) 単位の検出
- 抽象構文木 (Abstract Syntax Tree) を用いた検出
- プログラム依存グラフを用いた検出
- メトリクスやフィンガープリントなど，その他の技術を用いた検出

これらの検出手法はそれぞれ検出可能なコードクローンの種類や，コードクローンに検出に要する時間が異なる．

コードクローンの検出結果は検出手法や検出ツールによって異なるが，典型的にはコード片の集合 (クローンセット) または対 (クローンペア) として表される．このとき，コード片 $c$ は $c = \langle f_k, l_s, l_e \rangle$ と表すことによって一意に位置を識別できる．$f_k$ はコード片が含まれるファイルを，$l_s$ と $l_e$ はコード片の開始行と終了行を表す．

本研究では抽象構文木を用いたコードクローン検出ツールを使用する．抽象構文木を用いた検出では，前処理としてソースコードに対して構文解析を行い，抽象構文木を構成する．そして構築した抽象構文木上の同型の部分木がコードクローンとして検出される．例えば，ソースコード上の for 文の場合，図 1 で示すような抽象構文木が構築される (但し図中の抽象構文木は一部簡略している)．抽象構文木を用いたコードクローン検出は，字句単位の検出と同様に，検出結果がコーディングスタイルや識別子の差異に依存しない．

本来コードクローンは集合または対のことを指すが，本論文では，あるコード片が任意のクローンセットの要素に含まれている場合，便宜上そのコード片はコードクローンである，と表現する．

## 2.3 版管理システム

ソフトウェアを効率的に管理・保守するために多くの場合，版管理システムが用いられている．版管理システムを用いることで，ソースコードを含むすべてのファイルの作成日時，変更日時，変更箇所などの履歴を保管することができる．これらのファイルの変更情報を管理するデータベースをソフトウェアリポジトリ (又は単にリポジトリ) と呼ぶ．版管理システムは集中型，分散型などいくつかの形態が存在するが，多くの場合ソフトウェアの更新 (コ

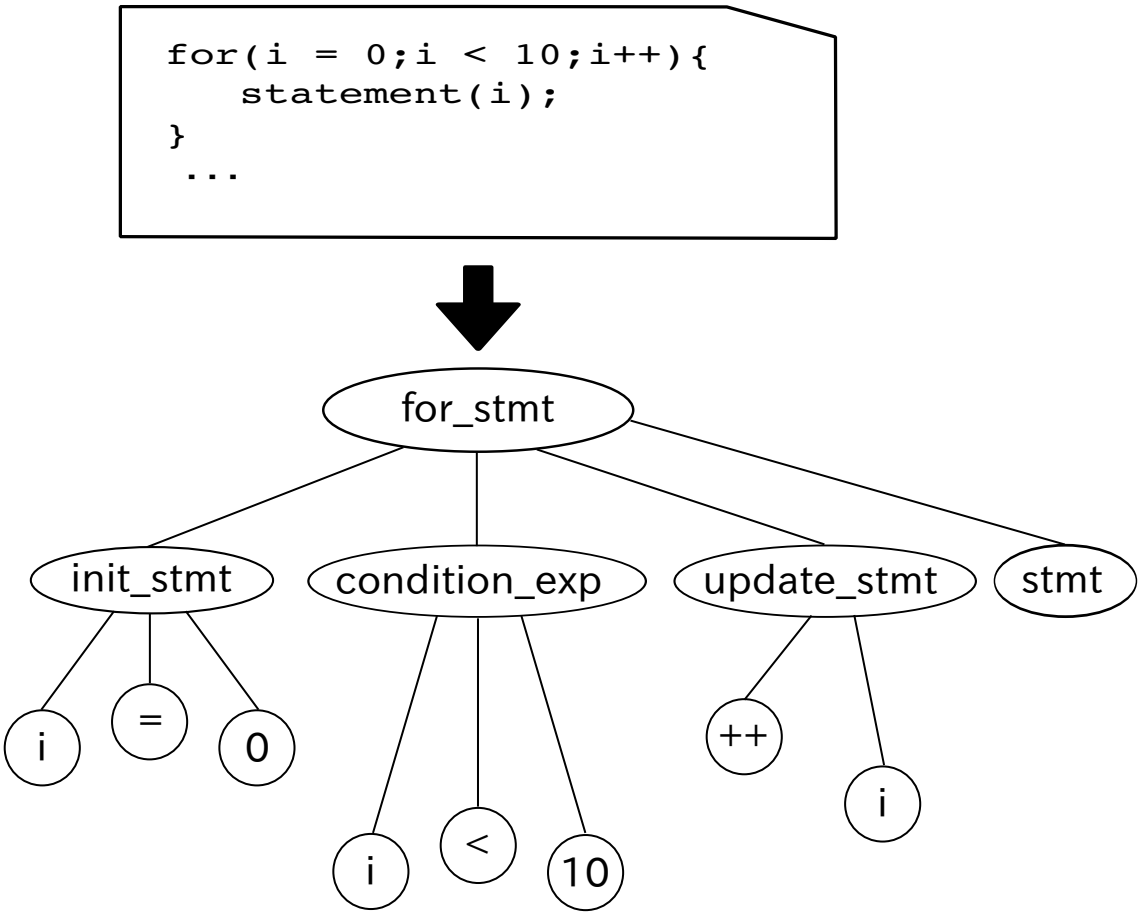

for(i = 0;i < 10;i++){
statement(i);
}
...
for_stmt
init_stmt
condition_exp
update_stmt
stmt
i
=
0
i
<
10
++
i

図 1: 抽象構文木の構築

```
fe0bf162 (Editor Name 2002-05-13 23:30:23 22) #include "config.h"
fe0bf162 (Editor Name 2002-05-13 23:30:23 23)
d7b24953 (Editor Name 2000-06-01 14:59:22 24) #include "gimp.h"
dd9a1db6 (Editor Name 2006-11-03 21:06:22 25) #undef GIMP_DISABLE_··
dd9a1db6 (Editor Name 2006-11-03 21:06:22 26) #undef __GIMP_LAYER_··
dd9a1db6 (Editor Name 2006-11-03 21:06:22 27) #include "gimplayer.h"
```

図 2: blame 機能の実行例

ミット) を行う度にその状態が一意に識別できるリビジョン番号が付与される．１つのリビジョン $r$ 内に含まれる情報を形式的に定義すれば $r = \langle A, T, l, f_1, f_2, ..., f_n \rangle$ と表される． ここで $A$ は修正を行った作業者の名前を，$T$ は時刻を，$l$ はコミットを行う際のログメッセージを，$f_i$ は修正が完了した後のファイル群を表す．よく使用されている版管理システムとして CVS，Subversion，Git などが挙げられる．これらの版管理システムはあるリビジョンでのファイルを，それよりも過去のファイルとの差分を比較することができる．これにより，ファイルの行ごとの作成日時・変更日時を算出できる．例えば版管理システム Git において blame 機能を使用したとき，図 2 のような出力結果が得られ，各行の編集者の名前，リビジョン番号，変更日時などがソースコードと共に表示される (図中の “Editor Name” には作業者の名前が表示される)．

本調査で対象とした版管理システムは Git であるので，以降では版管理システムは Git を想定して説明する．

### 2.4 欠陥管理システム

版管理システムとは別にソフトウェアの欠陥情報を一元して管理するために欠陥管理システムが用いられている．欠陥管理システムはソフトウェアに欠陥が生じたときにその日時，発見者，欠陥の内容を含む欠陥情報を登録することできる．欠陥情報を登録するとそれを一意に識別できる欠陥 ID が付与され，図 3 のように 1 つの欠陥情報に対して，修正状況を表す Status と修正結果を表す Resolution が記述される．

オープンソースソフトウェアで有名な欠陥管理システムとして，BugZilla がよく知られている．

| 欠陥ID | Status | Resolution | 詳細 |
|---|---|---|---|
| 100001 | New | --- | ・・・ |
| 100002 | Closed | FIXED | ・・・ |
| 100003 | Closed | WONTFIX | ・・・ |
| ・・・ | ・・・ | ・・・ | ・・・ |

図 3: 欠陥管理システムの各レコードの構成

## 3 既存手法

従来までコードクローンの存在はソフトウェアの保守性を低下させるためリファクタリングによって積極的にコードクローンを除去すべきと言われていた．しかし近年はコードクローンはソフトウェアの保守性を低下させず，積極的なコードクローンの除去は有効でないという報告がされている [16]．その中の 1 つに，Rahman らは欠陥管理システムに含まれる欠陥情報を基に，コードクローンが欠陥に直接影響するかどうかを調べている．

本章ではコードクローンと欠陥修正にどの程度影響を与えるかを調査した Rahman らの手法について説明する．Rahman らの手法では下記の手順で解析を行っている．

1. スナップショットの取得とコードクローンの検出
2. 欠陥修正に対応するコミットの取得
3. 欠陥修正に含まれるコードクローンの割合の算出

以降ではこれらの各ステップにおける手順を説明する．

### 3.1 スナップショットの取得とコードクローンの検出

ソフトウェアリポジトリのリビジョン群 $R$ のなか，スナップショットの集合 $S$ を抽出する．ここでは 1ヶ月おきのリビジョンをスナップショットとして抽出している．そして取得したそれぞれのスナップショットに対してコードクローン検出を行う．ここでコードクローン検出ツール DECKARD を使用している [13]．DECKARD はパラメータとして ⟨最小トークン長，類似度，ストライド⟩ の 3 つを与えることができるが，既存手法では $\langle 50, 1.0, 2\rangle$，$\langle 50, 0.99, 2\rangle$ の 2 通りの実験を行っている．

コードクローン検出の出力結果は $O = \langle g_1, g_2, ..., g_n\rangle$ と表され，ここで $g_i$ は 1 つのクローンセットである．このコードクローン検出処理によって，全てのコード片はコードクローン

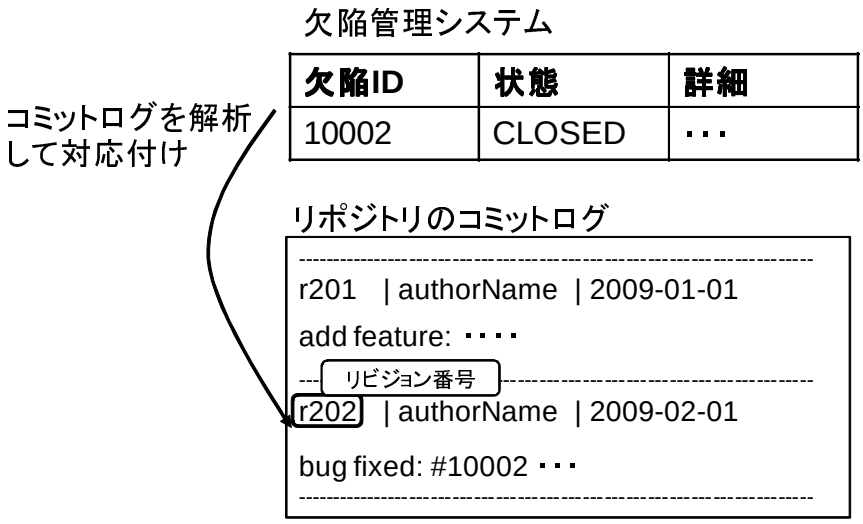



図 4: 欠陥情報とリビジョンの対応付け

であるかユニークなコード片のどちらかに分類される．

### 3.2 欠陥修正に対応するコミットの取得

欠陥を含むコード片がどこに存在するのか分かっていることが理想的であるが, 一般に欠陥となるコード片の位置を特定することは困難である．そのため BugZilla の欠陥管理システムに登録されている修正済みの欠陥情報を取得し，その欠陥の修正時に変更されたコード全てを欠陥コード (Buggy Code) とみなすことによって欠陥となるコード片の場所を特定する．欠陥管理システムに登録されている欠陥情報から修正されたコードを特定するには，リポジトリから欠陥が修正されたコミットと欠陥情報を対応付ける必要がある．この対応付けを行うために図 4 で示すように欠陥管理システムに登録されている欠陥 ID とリポジトリ内に保存されているコミットログを用いる．この対応付けを行うには，コミットログ内に欠陥修正が行われたと判断できる文言が存在するかどうか調べればよい．既存手法では *Śliwerski* らの手法 [1] を基にヒューリスティックを用いて拡張している．*Śliwerski* らの手法は基本的にはコミットログの内容が以下の正規表現に該当すれば，欠陥修正であると判定している．

- `bug[# \t]*[0-9]+`
- `pr[# \t]*[0-9]+`
- `show\_bug\.cgi\?id=[0-9]+ or \[[0-9]+\]`

欠陥修正が行われたリビジョン $r$ を特定出来れば，特定したリビジョン $r$ とその直前のリビジョン $r-1$ との差分を抽出するとこにより，Buggy Code を得ることができる．この差分を得る処理はリポジトリの diff 機能を用いることによって抽出できる．

### 3.3 欠陥修正に含まれるコードクローンの割合の算出

それぞれの欠陥情報と修正されたコミットの対応付けの完了後，該当するリビジョン $r$ と最も近いスナップショット $s$ を対応付ける．そしてスナップショット $s$ 中の Buggy Code の中に含まれるコードクローンの割合を算出する．欠陥修正が行われたリビジョン $r$ とスナップショット $s$ の間にコードの変更が生じる可能性があるので，diff 機能を用いて調整を行う．例えば，リビジョン $r$ とスナップショット $s$ の間にコードが $n$ 行追加された場合，追加された行以降のコードの対応付けを $n$ 行ずらして行う．

このような手法によって実験を行ったところ，欠陥修正によって変更されたコードにコードクローンが含まれている割合は小さく, コードクローンは欠陥を引き起こす要因に当たらないという結果が報告している．

# 4　調査手法

本章では，前章の既存手法を基にコード片の生存期間を考慮した 2 つの調査手法について説明する．

## 4.1　既存手法の拡張

既存手法では，全てのコード片がコードクローンであるかユニークなコード片であるかのどちらかに分類してそれらがどの程度 Buggy Code に含まれているかを調査したが．コードがそれまでにどの程度の期間存在していたかについては考慮していない．我々は，コードクローンの分析においてコード片の生存期間が重要であると考えている．コード片の生存期間が重要である理由は，既に修正が頻繁に生じているコード片は今後も修正が生じやすいと考えられ，また，たとえコードクローンであってもよく知られたプログラミングロジックや長期間使われている実績のあるコードであればそれらのコードは長期間存在して欠陥修正が行われにくいと考えられるからである．そのため，本研究では生存期間を考慮した上で，コードクローンと Buggy Code の割合の調査を行う．

本章では既存手法を基にして，生存期間がコードクローンと Buggy Code にどのような影響を与えるかを調査する．調査は大きく分けて 2 つ行い，以下の内容を調べる．

- 生存期間の短いコードクローンが欠陥修正に含まれる割合
- コード片を生存期間に応じて分類した時の欠陥コードを含む割合

それぞれの調査手法について以降で述べる．

## 4.2　生存期間の短いコードクローンが欠陥修正に含まれる割合の調査

既存手法ではコードクローンが欠陥修正に影響を与えるかを調べているが，コードクローンの生存期間に関しては考慮していない．コードクローンの生存期間を考慮することで，より欠陥を多く含むコードクローンを特定することができると考える．したがって本節で説明する実験により，以下の仮説の検証を行う．

**仮説 1** Buggy Code には生存期間の短いコードクローンが生存期間の長いコードクローンよりも多く含まれている．

3 節で説明した手順と同様に，スナップショットの取得，コードクローンの検出，欠陥修正と対応するコミットの取得を行う．但し既存手法ではスナップショットの取得間隔を 1 ヶ月おきとしていたが，本手法では取得間隔をより小さくし，欠陥 ID と欠陥修正のコミットの

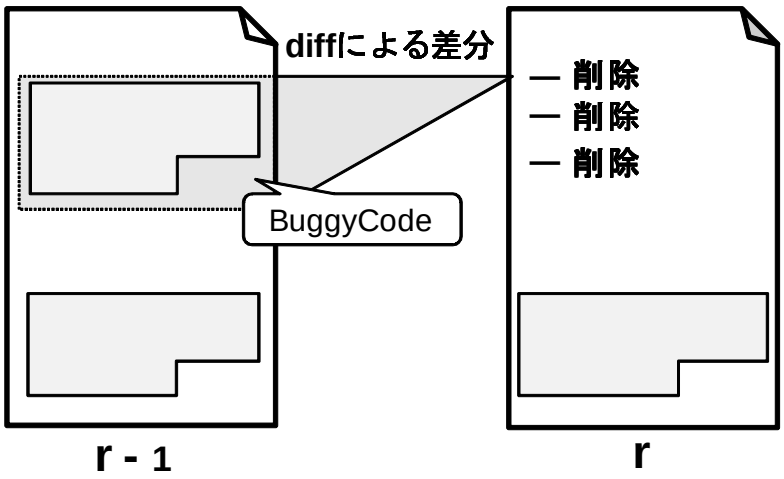



図 5: 欠陥コード (Buggy Code) の抽出

対応付けが完了したすべてのリビジョンにおいてスナップショットを取得し，コードクローン検出を行っている．

その後，以下のような手順で欠陥修正に含まれるコードクローンの割合の調査を行う．

1. Buggy Code 内に含まれるコードクローンの重複率の算出
2. コード片の変更の有無を過去のリビジョンと比較
3. コード片の生存期間の算出

以降でこれらの各ステップの手順を示す．

**Buggy Code 内に含まれるコードクローンの重複率の算出**　欠陥修正が生じたリビジョン $r$ をコミットログによって対応付ければ，リビジョン $r$ 直前のリビジョン $r-1$ から diff 機能による差分を取得することによって Buggy Code の位置を特定できる．diff 機能は 2 つのファイル間の追加された箇所と削除された箇所を抽出することが可能であるので，図 5 のように直前のコミットによって削除されたコード片が Buggy Code として抽出できる．

次に各スナップショットに対してコードクローン検出を行っているので，以下のようにして重複率 $d$ を求める．

$$d = \frac{l_{buggy}}{l_e - l_b} \tag{1}$$

ここで $l_s$，$l_e$ はコードクローンの開始行と終了行を，$l_{buggy}$ はコードクローンの開始行と終了行の中で Buggy Code が含まれる行数である．また，コードクローン検出時にコードクローンが含まれるリビジョン，ファイル名も取得する．

**コード片の変更の有無を過去のリビジョンと比較**　手順 1 で抽出されたコードクローンに包含されるコード片に最も新しく変更が加えられた時刻を算出する．図 6 で示すようにコードクローンに包含されるコード片が得られたリビジョン $r_s$ より約 1ヶ月毎に過去のリビジョン群 $r = \langle r_0, r_1, ..., r_n \rangle\, (r_i < r_s)$ を取得する．取得したリビジョン群から下記の条件を満たすリビジョン $r_m$ を取得する．

$$r_m = \max_i \left( r_i * \delta_{diff}(r_i, r_s, f, l_b, l_e) \right) \tag{2}$$

ここで $\delta_{diff}$ は，リビジョン $r_s$ 中のファイル $f$ 内の開始行 $l_b$，終了行 $l_e$ にあるコード片がリビジョン $r_i$ まで変化がなければ 1，そうでなければ 0 を返す関数と定義する．つまり $r_m$ はコードクローンに包含されるコード片のリビジョンより古く，かつその中で最も新しい変更が生じたリビジョンを表す．コードの変化は diff 機能により変更のあった開始行 $l_{mb}$，終了行 $l_{me}$ が得られるので，図 6 に示すようにコードクローンに包含されるコード片の開始行 $l_b$，終了行 $l_e$ との間に以下の関係が満たされれば，変更があったと判定する．

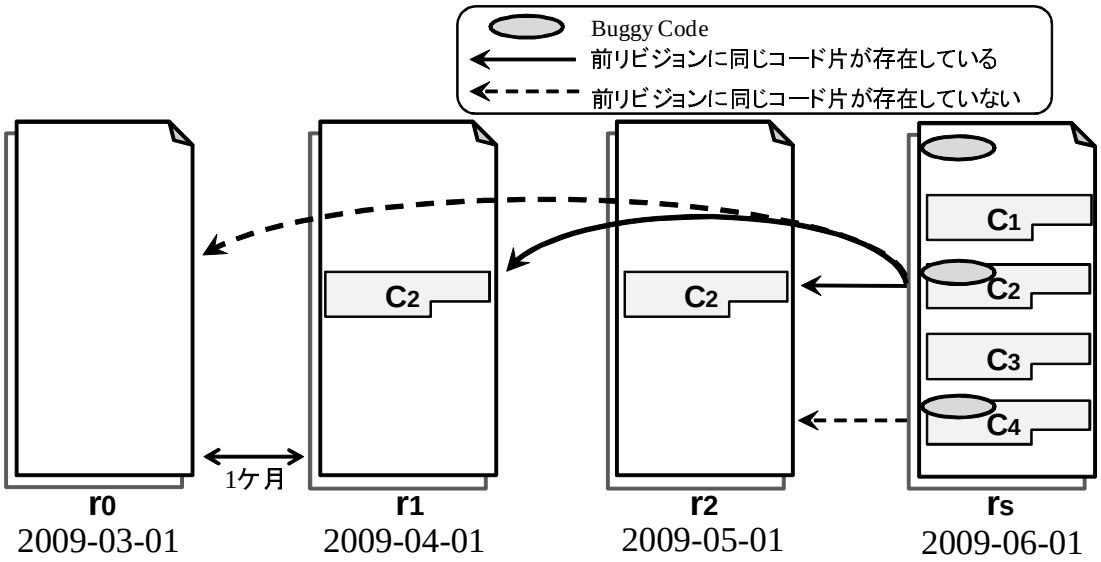



図 6: コード片の生存期間の取得

1. $l_b > l_{mb}$ かつ $l_e \geq l_{me}$：コード片の開始行をまたいだ変更

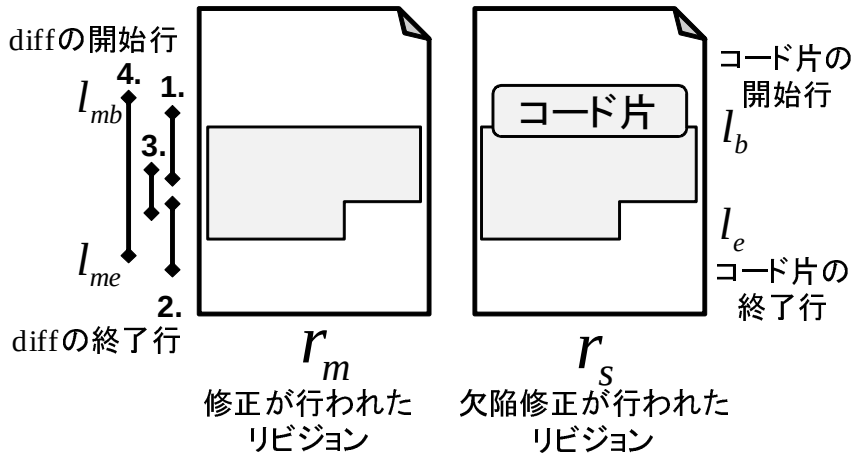



図 7: diff 機能による変更の有無の判定

2. $l_b \leq l_{mb}$ かつ $l_e < l_{me}$：コード片の終了行をまたいだ変更

3. $l_b \leq l_{mb}$ かつ $l_e \geq l_{me}$：コード片の内部の変更

4. $l_b > l_{mb}$ かつ $l_e < l_{me}$：コード片全体を包含する変更

上記の 4 つのそれぞれの条件は，図 7 において各番号の $l_{mb}$，$l_{me}$ の位置に差分が存在した時に対応している．

**コード片の生存期間の算出**　手順 2 で求めたリビジョン $r_m$ と $r_s$ からそのコミットが行われた日時を取得する．その両者の日時の差を日数の単位になおし，コードクローンの生存期間として算出する．

### 4.3　コード片を生存期間に応じて分類した時の欠陥コードを含む割合の調査

前節の調査方法では，生存期間がコードクローンの含む欠陥修正の割合を調査したが，コードクローンでないコードに対しては調査できていない．

そこで，コードクローンとコードクローンでないコードを生存期間に応じて分類したときに，Buggy Code を含む割合に差異が存在するかどうかを調べる．本調査は以下の 3 つの仮説を検証する．

**仮説 2** 生存期間が短いコードクローンは生存期間が長いコードクローンより多くの Buggy Code を含む．

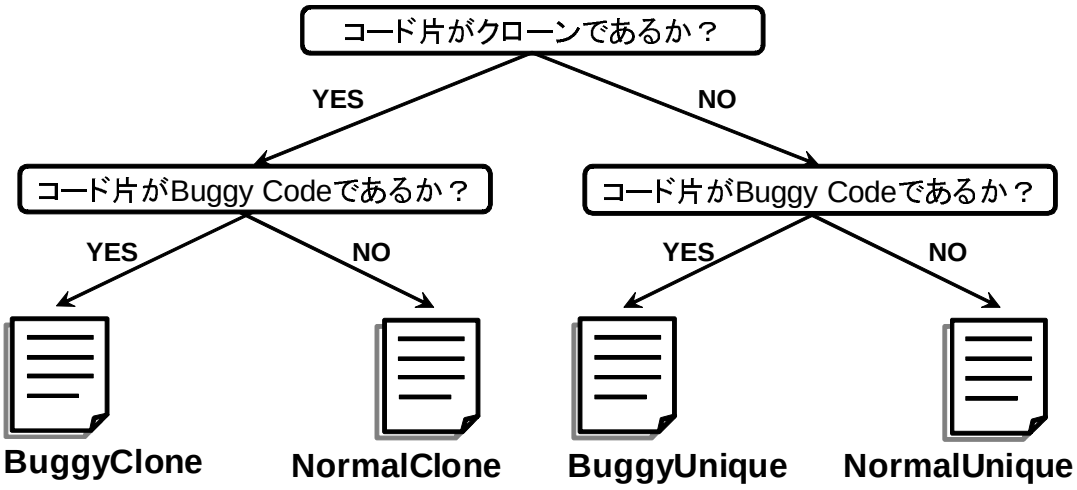



図 8: コード片の 4 つの分類

**仮説 3** 生存期間が短い非コードクローンは生存期間が長い非コードクローンより Buggy Code を多く含む.

**仮説 4** 仮説 2，仮説 3 と比較すると生存期間の短いコードクローンの方が非コードクローンより多くの欠陥を含む.

これらの仮説を検証するために，図 8 のようにコード片をコードクローンであるかコードクローンでないかに分類し，さらにそれらを Buggy Code であるかどうかの 4 つに分類する．これらの分割されたグループをそれぞれ集合とし以下のような名前を付ける.

- $BuggyClone(BC)$ ⋯ Buggy Code かつコードクローンとなるコード片の集合
- $NormalClone(NC)$ ⋯ Buggy Code でないかつコードクローンとなるコード片の集合
- $BuggyUuique(BU)$ ⋯ Buggy Code かつコードクローンでないコード片の集合
- $NormalUnique(NU)$ ⋯ Buggy Code でないかつコードクローンでないコード片の集合

以上の定義した集合の間で生存期間が与える影響の調査を行う．具体的には以下のような手順でコード片の生存期間が与える影響の調査を行う.

1. スナップショットの取得，コードクローンの取得，欠陥修正と対応するコミットの取得
2. リポジトリ内のファイル一覧の取得

3. ファイル一覧から空行とコメント行の除去

4. blame 機能による生存期間の取得

5. 各コード片の集合での生存期間の集計

以降でこれらの各ステップの手順を示す.

**スナップショットの取得，コードクローンの取得，欠陥修正と対応するコミットの取得**　前節の調査手法と同様にスナップショットを取得し，それぞれからコードクローンの検出を行う．またそれと同時に欠陥管理システムに登録されている欠陥情報と，その欠陥修正が行われたコミットとの対応付けを行う.

**リポジトリ内のファイル一覧の取得**　取得したスナップショットから解析対象となるソースコードのファイルを抽出する．抽出方法はファイルの拡張子を用いた正規表現を用いて行う．具体的にはプログラミング言語 C を対象とした解析を行う時は “∗.c” を，プログラミング言語 Java を対象とした解析を行う時は “∗.java” を正規表現として用いたときに適合したファイル群を抽出する.

**ファイル一覧から空行とコメント行の除去**　取得したソースファイル一覧の中に含まれる空行，コメントに該当する行の除去を行う．ここで，対象となるソースコードから直接コメントや空行を除去してしまうと，コード片の位置情報が変化してしまい生存期間の取得が困難になる．そこで空行とコメントに該当する行はそれを表すフラグを別途設け，それを別に管理することで空行・コメントの位置を判断する．例えば，図 9(a) のソースコードに対して，空行・コメント除去を行うと，図 9(b) のような箇所に空行，コメントのフラグが付与される.

上記の 4 つの集合の中でコードクローンとなるコード片を含むものは，コードクローン検出においてコード片の大きさが決定できたが，コードクローンでないユニークなコード片はその大きさを決定することが困難であるため，下記の図 10 のようにコード片の 1 行単位にコードが作成された日時を求め，ファイルの更新日時のとの差分を取得する．1 行単位でコード片の作成日時を取得するために，ここでは Git の blame 機能を使用する.

**各集合での生存期間の集計**　blame 機能によって取得したコード 1 行ごとの生存期間を，BuggyClone, NormalClone, BuggyUnique, NormalUnique の 4 つのコード片の集合ごとに集計する．手順 1 によりコードクローンと Buggy Code がソースコード上のどの位置にある

```
/**gimp_vectors_new_from_file:
 * Since: GIMP 2.4
 */
gboolean
gimp_vectors_new_from_file (…
{
GimpParam *return_vals;
gint nreturn_vals;
gboolean success = TRUE;

return = gimp_run_procedure(…);
```

(a) 空行コメント除去前のソースコード

```
*  /**gimp_vectors_new_from_file:
*   * Since: GIMP 2.4
*   */
    gboolean
    gimp_vectors_new_from_file (…
    {
    GimpParam *return_vals;
    gint nreturn_vals;
    gboolean success = TRUE;
*
    return = gimp_run_procedure(…);
```

(b) 空行コメント除去後のソースコード

図 9: ソースコードの空行，コメントの除去

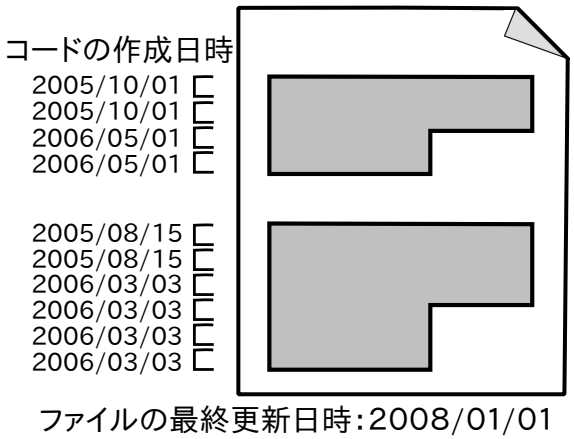



図 10: １行単位での作成日時の取得

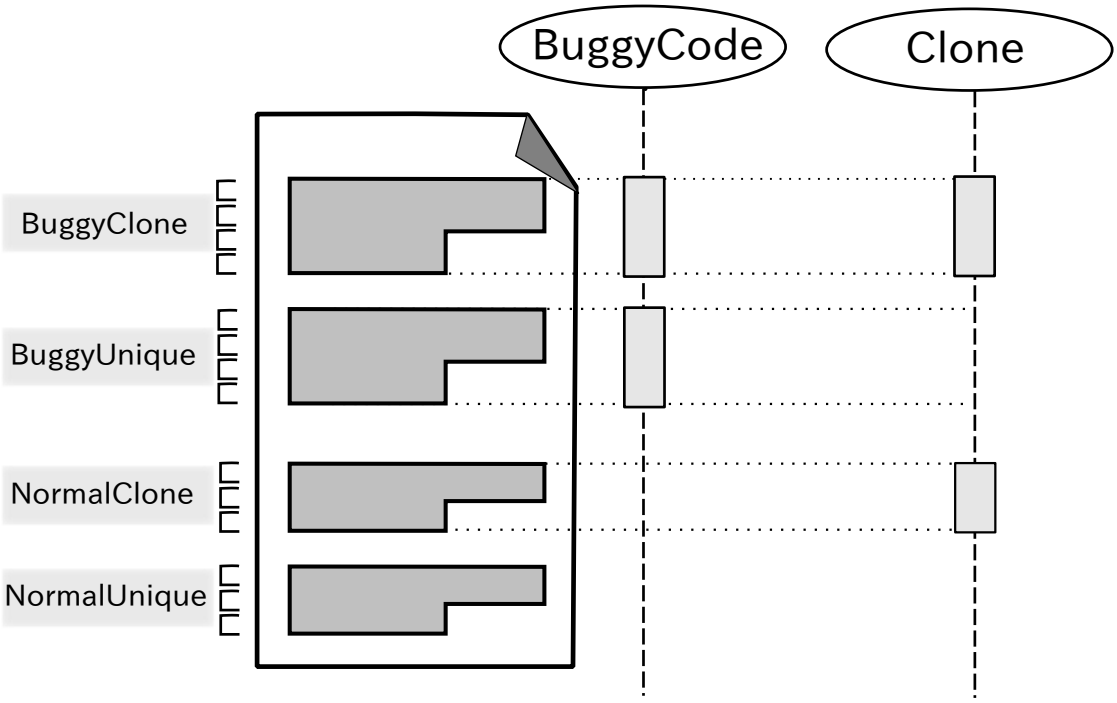



図 11: コード片の各集合での分類

かが分かっているので，下記の図 11 のように集計する．図 11 の右側に表示される長方形の図が Buggy Code とコードクローンであるコード片の位置情報を表している．

その後，生存期間が長いコードと短いコードに分けたとき，BuggyUnique の集合と NormalUnique の集合に含まれるコード片がどの程度存在しているかを算出する．同様の処理を，BuggyClone の集合と NormalClone の集合に対しても行う．このように生存期間がクローンに与える影響を調査する．

# 5　適用実験

本章では，調査手法で説明した 2 つの調査を行い，得られた結果について説明する．

## 5.1　生存期間の短いコードクローンが欠陥修正に含まれる割合の調査の結果

### 5.1.1　実験概要

4.2 節で述べた内容を以下の 2 つのオープンソースソフトウェアを対象として実験を行った．

**Gimp**　オープンソースソフトウェアの画像編集ソフトウェア [10]．2010 年 7 月時点でファイル数 2831，約 946kLOC

**Evolution**　Gnome デスクトップ環境に付随している標準メールクライアント [7]．2010 年 2 月時点でファイル数 1127，約 456kLOC

いずれの対象も欠陥情報は BugZilla で管理され，ソースコードは Git リポジトリによって管理されている．BugZilla で管理されている欠陥情報のうち，下記の 2 つの条件を満すもののみを抽出した．

- 欠陥情報の “Status” が “RESOLVED”，“VERIFIED”，“CLOSED” のいずれかに設定
- 欠陥情報の “Resolution” が “FIXED” に設定

コードクローン検出には既存手法同様に DECKARD を使用した．また DECKARD に与えるパラメータ ⟨最小トークン長，類似度，ストライド⟩ は $\langle 50, 1.0, 2\rangle$ を用いた．

### 5.1.2　実験結果

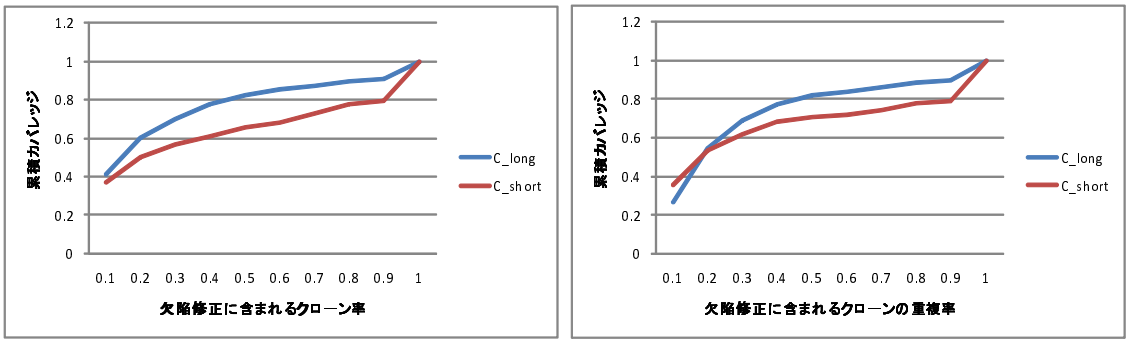
各プロジェクトにおける抽出した欠陥情報の数，対応付けを行ったリビジョンの数を表 1 に示す．表 1 に記述しているコードクローンの数は，一部でも Buggy Code を含むものを表している．また対応付けたリビジョン数とは，コミットログ中に含まれる欠陥 ID や修正メッセージからその欠陥 ID が欠陥管理システムから見つかった数を表している．その後，欠陥情報と修正を行ったリビジョンを対応付け，その中で欠陥コードに含まれるコードクローンを抽出した．抽出したコードクローンの数とその生存期間を表 2 に示す．ここでは生存期間最大とは，コードクローンに該当するコード片が修正されるまでの最も長い期間を日数で表している．

次に生存期間を取得したコード片を以下の 2 つのグループに分類する．

- $C_{long}$ … 生存期間 (中央値) よりも生存期間が長いコード片のグループ

- $C_{short}$ ･･･ 生存期間 (中央値) よりも生存期間が短いコード片のグループ

それぞれのグループにおいて，各コードクローンに包含されるコード片が，Buggy Code 内に何割含まれていたのかを算出する． Gimp プロジェクトと Evolution プロジェクトにおいて各コードクローンが含む Buggy Code を図 12 に示す．図 12 の横軸はコードクローンが重複している Buggy Code の割合であり，これはコードクローンを含む Buggy Code の全体行のうち，何割がコードクローンであったかを表している．縦軸はそれらのコードクローンが全体の何割を占めているかの累積カバレッジであり，欠陥修正に含まれるクローンの重複率 $d$ 以下の欠陥修正が全体の何割存在するかを表している．図 12 では，Gimp, Evolution プロジェクトともに，生存期間が短いグループ $C_{short}$ のほうが欠陥修正により多くのコードクローンが存在する傾向があることが分かる．例えば，Gimp プロジェクトにおいてクローン率 0.4 以下の欠陥修正は，生存期間が長いコード片のグループでは，78.0%を占めていたが，生存期間が短いコード片のグループでは 61.2%しか占めていない．同様に Evolution プロジェクトでもクローン率 0.5 以下の欠陥修正は生存期間が長いコード片のグループでは 82.0%を占めていたが生存期間が短いコード片のグループでは 71.0%であった．




(a)Gimp プロジェクト　　(b)Evolution プロジェクト

図 12: 欠陥コード内でクローンが占める割合

表 1: 調査を行ったプロジェクトの概要

| プロジェクト名 | Gimp | Evolution |
|---|---|---|
| 欠陥情報の数 | 698 | 11973 |
| 対応付けたリビジョン数 | 387 | 1676 |
| 最も古いリビジョンの日付 | 2004-01-11 | 2005-04-13 |
| 最も新しいリビジョンの日付 | 2009-01-23 | 2010-08-20 |

### 5.2 コード片を生存期間に応じて分類した時の欠陥修正の割合の調査の結果

#### 5.2.1 実験概要

4.3 節で述べた内容を前節の調査での 2 つのオープンソースソフトウェアに加えて，下記の実験対象を加えて実験を行なった.

**Ant** Java プログラムを対象とした自動ビルドツール [2]．2011 年 1 月時点でファイル数 1193，約 254kLOC.

**Nautilus** Gnome デスクトップの標準 GUI マネージャ[17]．2011 年 1 月時点でファイル数 158，約 139kLOC.

**Eclipse JDT Core** 統合開発環境 Eclipse に標準搭載されているプログラム開発ツール群内のコアパッケージ [12]．2010 年 12 月時点でファイル数 1183, 約 439kLOC.

#### 5.2.2 実験結果

各プロジェクトにおけるコードを 4 つの集合に分類した時の，それぞれの集合に含まれるコード行数と，各プロジェクトが欠陥コードを含む割合を 3 に示す.

ここで表中の数値は欠陥管理システムに登録されている欠陥情報が 版管理システムのコミットログと対応付けされたリビジョン全てのソースコードの行数の和である．また表 3 中の総行数と欠陥コード率は以下のようにして算出している.

$$\text{総行数 [行]} = BuggyClone + NormalClone + BuggyUnique + NormalUnique \quad (3)$$

$$\text{欠陥コード率 [\%]} = \frac{BuggyClone + BuggyUnique}{\text{総行数}} * 100 \quad (4)$$

例えば Gimp プロジェクトの欠陥コード率は 0.012[%] であるので，コード 10 万行に対して Buggy Code が約 12 行含まれていたと言える.

表 2: 欠陥情報に含まれるコードクローンの概要

| プロジェクト名 | Gimp | Evolution |
|---|---|---|
| コードクローンの数 | 1093 | 1114 |
| 生存期間最大 [日] | 3460 | 3812 |
| 生存期間最小 [日] | 32 | 41 |
| 生存期間中央値 [日] | 574 | 1791 |

各プロジェクトのコード片の生存期間の分布を取得した時の生存期間の平均値と中央値を表 4 に示す．この数値の単位は日数を表している．表 4 中の NormalClone と BuggyClone でコード片の生存期間を比較すると，全てのプロジェクトにおいて BuggyClone の方が平均値，中央値ともに低くなっている．NormalUnique と BuggyClone を比較しても同様に BuggyClone の方が中央値，平均値ともに低いことが分かる．

次に，各プロジェクトのコード片の生存期間の分布から第 1 四分位，中央値，第 3 四分位を算出する．さらに各プロジェクトの生存期間の長さに応じて以下に示す 4 つの集合に分割する．

- $D1$ … 生存期間が第 1 四分位以下のコード片
- $D2$ … 生存期間が第 1 四分位を超え，中央値以下のコード片
- $D3$ … 生存期間が中央値を超え，第 3 四分位以下のコード片
- $D4$ … 生存期間が第 3 四分位を超えるコード片

表 3: 各プロジェクトの概要

| | NormalClone | BuggyClone | NormalUnique | BuggyUnique | 総行数 | 欠陥コード率 [%] |
|---|---|---|---|---|---|---|
| Ant | 15,997,102 | 1,655 | 37,629,266 | 4,463 | 53,632,486 | 0.011 |
| Gimp | 226,622,231 | 22,915 | 1,188,173,223 | 143,240 | 1,414,961,609 | 0.012 |
| Evolution | 51,213,834 | 4,583 | 639,581,649 | 45,228 | 690,845,294 | 0.007 |
| Nautilus | 3,970,709 | 862 | 38,384,602 | 6,132 | 42,362,305 | 0.017 |
| Eclipse JDT | 93,210,198 | 12,974 | 135,431,123 | 16,537 | 228,670,832 | 0.013 |

表 4: 各プロジェクトでの生存期間の平均値と中央値

| | | NormalClone | BuggyClone | NormalUnique | BuggyUnique |
|---|---|---|---|---|---|
| Ant | 平均値 [日] | 976 | 441 | 1080 | 515 |
| | 中央値 [日] | 747 | 282 | 864 | 327 |
| Gimp | 平均値 [日] | 984 | 558 | 1024 | 772 |
| | 中央値 [日] | 813 | 382 | 832 | 606 |
| Evolution | 平均値 [日] | 1593 | 1226 | 1372 | 1146 |
| | 中央値 [日] | 1629 | 971 | 1302 | 974 |
| Nautilus | 平均値 [日] | 707 | 244 | 1066 | 750 |
| | 中央値 [日] | 352 | 103 | 764 | 539 |
| Eclipse JDT | 平均値 [日] | 901 | 468 | 903 | 567 |
| | 中央値 [日] | 733 | 309 | 738 | 302 |

ここで，BuggyClone(BC) のコード片は $BC_{D1}, BC_{D2}, BC_{D3}, BC_{D4}$ に分割できる．NormalClone, NormalUnique, BuggyUnique に対しても同様に分割できる．

次に，コードクローンとコードクローンでないコードに対して，生存期間別に分けたときに Buggy Code の含む割合を算出する．コードクローンとなるコードが生存期間 $D_i(i = 1..4)$ における Buggy Code を含む割合 $RC(Di)$ は以下のようして算出する．

$$RC(Di) = \frac{BC_{Di}}{NC_{Di}} \tag{5}$$

上記の式は，NormalClone に対する BuggyClone の割合を表している．同様に，コードクローンでないコードが生存期間 $D_i(i = 1..4)$ における Buggy Code を含む割合 $RU(Di)$ は以下のようにして算出する．

$$RU(Di) = \frac{BU_{Di}}{NU_{Di}} \tag{6}$$

上記の式は，NormalUnique に対する BuggyUnique の割合を表している．

さらに生存期間ごとに Buggy Code の含まれる割合を相対的に比較する為に正規化を行う．ここでは各プロジェクトに対して 1 つの値での正規化と，各系列ごと正規化の 2 種類行う．

**各プロジェクトに対して正規化** $RU(D4)$ を基準として生存期間ごとに Buggy Code を含む割合を求める．コードクローンとなるコードとコードクローンでないコードが生存期間 $D_i$ における Buggy Code を含む正規化された割合 $\widetilde{RC}(Di)$，$\widetilde{RU}(Di)$ は以下のように定義する．

$$\widetilde{RC}(Di) = \frac{RC(Di)}{RU(D4)}, \qquad \widetilde{RU}(Di) = \frac{RU(Di)}{RU(D4)} \tag{7}$$

定義より，$\widetilde{RU}(D4)$ は常に 1 となる．

**各系列に対して正規化** $RC(D4)$，$RU(D4)$ をそれぞれ基準として生存期間ごとに Buggy Code を含む割合を求める．コードクローンとなるコードとコードクローンでないコードが生存期間 $D_i$ における Buggy Code を含む正規化された割合 $\overline{RC}(Di)$，$\overline{RU}(Di)$ は以下のように定義する．

$$\overline{RC}(Di) = \frac{RC(Di)}{RC(D4)}, \qquad \overline{RU}(Di) = \frac{RU(Di)}{RU(D4)} \tag{8}$$

定義より，$\overline{RC}(D4)$，$\overline{RU}(D4)$ は常に 1 となる．

各プロジェクトに対して正規化した生存期間ごとの Buggy Code を含む割合を表 5 に，各系列に対して正規化した Buggy Code を含む割合を表 6 に示す．表の列は生存期間の集合を表し，行は各プロジェクトのコードクローンとなるコード片とコードクローンでないコード片が含む Buggy Code の割合を示している．また表中の下線の部分が正規化の基準として使

用されていることを表している．表中の数値は，基準値に対してどの程度の割合で Buggy Code を含んでいるかを示している．例えば，Ant プロジェクトにおいて $\widetilde{RC}(D4)$ は 0.53 であるので，生存期間が第 3 四分位を超えるコードクローンとなるコード片はコードクローンでないコード片よりも約 0.53 倍の欠陥を含んでいることを示している．同様に Gimp プロジェクトにおいて $\overline{RC}(D1)$ は 5.68 であるので，生存期間が第 1 四分位以下でコードクローンとなるコード片は，生存期間が第 3 四分位を超えるコードクローンとなるコード片よりも約 5.68 倍 Buggy Code を含んでいることを示している．

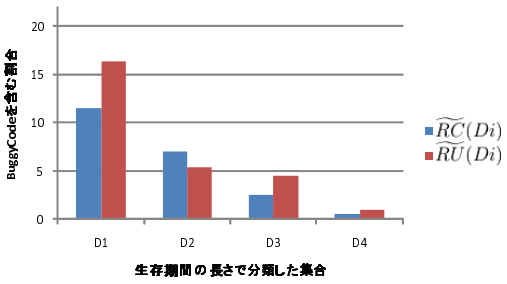
また，各プロジェクトにおける Buggy Code の含む割合をグラフに示したものを図 13 と図 14 に示す．

表 5: Buggy Code を含む割合 (各プロジェクトに対して正規化)

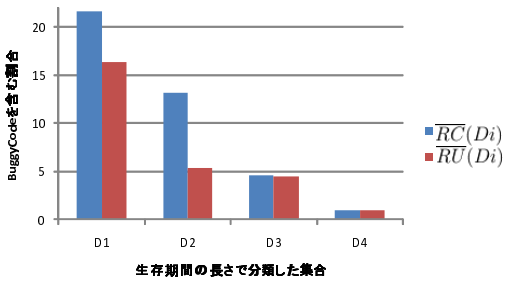
| | | D1 | D2 | D3 | D4 |
|---|---|---|---|---|---|
| Ant | $\widetilde{RC}(Di)$ | 11.52 | 6.98 | 2.46 | 0.53 |
| | $\widetilde{RU}(Di)$ | 16.38 | 5.36 | 4.54 | 1.00 |
| Gimp | $\widetilde{RC}(Di)$ | 2.27 | 1.65 | 1.10 | 0.40 |
| | $\widetilde{RU}(Di)$ | 2.41 | 1.43 | 1.67 | 1.00 |
| Evolution | $\widetilde{RC}(Di)$ | 1.45 | 3.72 | 0.80 | 0.99 |
| | $\widetilde{RU}(Di)$ | 1.68 | 1.40 | 1.01 | 1.00 |
| Nautilus | $\widetilde{RC}(Di)$ | 2.87 | 1.17 | 0.63 | 0.31 |
| | $\widetilde{RU}(Di)$ | 2.21 | 1.44 | 0.61 | 1.00 |
| Eclipse JDT | $\widetilde{RC}(Di)$ | 4.50 | 2.92 | 1.29 | 0.55 |
| | $\widetilde{RU}(Di)$ | 4.05 | 1.52 | 1.58 | 1.00 |

表 6: Buggy Code を含む割合 (各系列で正規化)

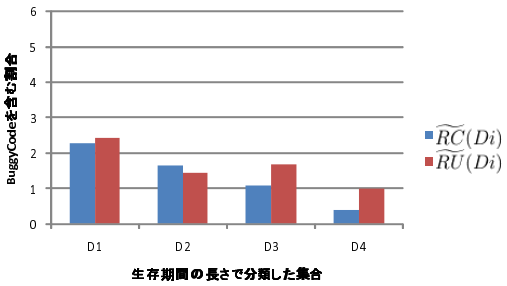
| | | D1 | D2 | D3 | D4 |
|---|---|---|---|---|---|
| Ant | $\overline{RC}(Di)$ | 21.63 | 13.12 | 4.62 | 1.00 |
| | $\overline{RU}(Di)$ | 16.38 | 5.36 | 4.54 | 1.00 |
| Gimp | $\overline{RC}(Di)$ | 5.68 | 4.12 | 2.73 | 1.00 |
| | $\overline{RU}(Di)$ | 2.41 | 1.43 | 1.67 | 1.00 |
| Evolution | $\overline{RC}(Di)$ | 1.47 | 3.75 | 0.80 | 1.00 |
| | $\overline{RU}(Di)$ | 1.68 | 1.40 | 1.01 | 1.00 |
| Nautilus | $\overline{RC}(Di)$ | 9.16 | 3.75 | 2.02 | 1.00 |
| | $\overline{RU}(Di)$ | 2.21 | 1.44 | 0.61 | 1.00 |
| Eclipse JDT | $\overline{RC}(Di)$ | 8.11 | 5.27 | 2.32 | 1.00 |
| | $\overline{RU}(Di)$ | 4.05 | 1.52 | 1.58 | 1.00 |



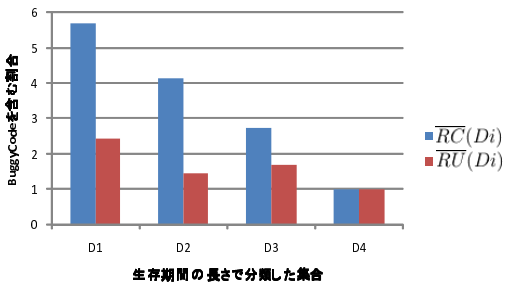

(a) Ant(各プロジェクトで正規化)



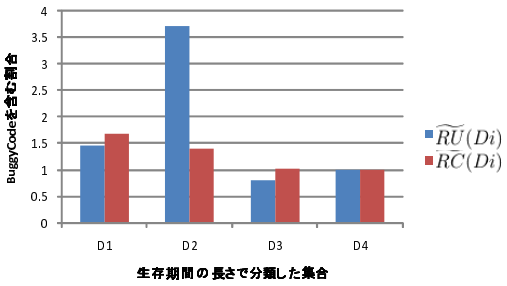

(b) Ant(各系列で正規化)



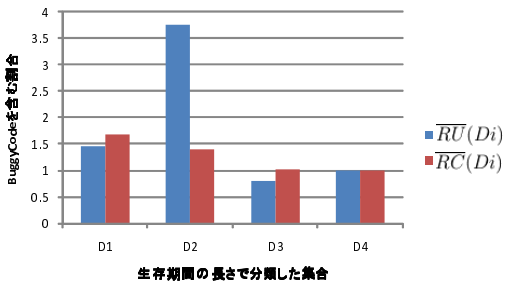

(c) Gimp(各プロジェクトで正規化)



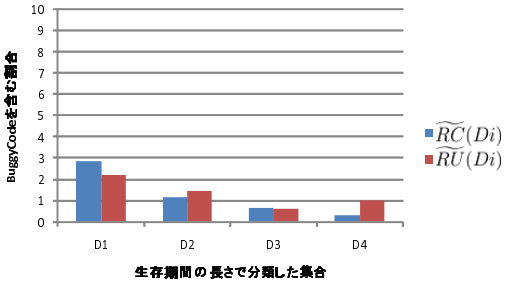

(d) Gimp(各系列で正規化)



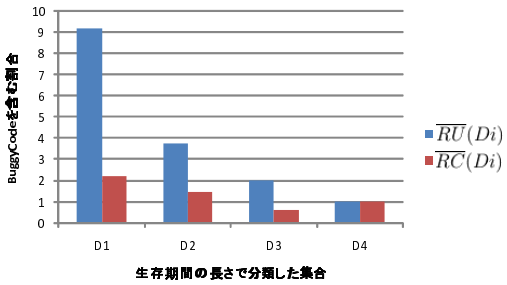

(e) Evolution(各プロジェクトで正規化)




(f) Evolution(各系列で正規化)

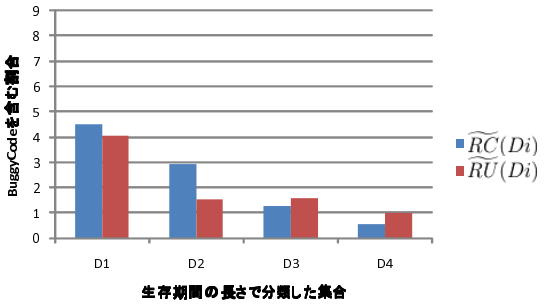



(g) Nautilus(各プロジェクトで正規化)

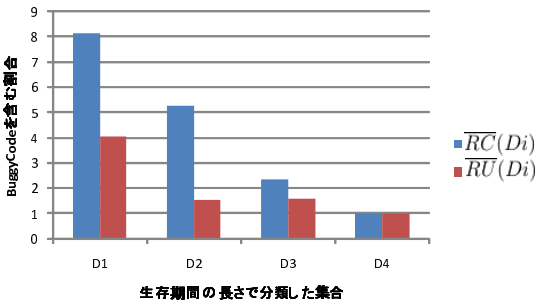



(h) Nautilus(各系列で正規化)

図 13: 各集合での Buggy Code を含む割合 (1)




(i)Eclipse JDT(各プロジェクトで正規化)




(j)Eclipse JDT(各系列で正規化)

図 14: 各集合での Buggy Code を含む割合 (2)

# 6 考察

本章では調査手法で示した仮説が成り立つかを検証し，また調査結果から得られた知見に関して述べる．

## 6.1 仮説の検証

調査手法で示した仮説を再掲する．

**仮説 1** Buggy Code には生存期間の短いコードクローンが生存期間が長いコードクローンよりも多く含まれている．

**仮説 2** 生存期間の短いコードクローンは，生存期間が長いコードクローンよりも Buggy Code を多く含む．

**仮説 3** 生存期間が短い非コードクローンは生存期間が長い非コードクローンより Buggy Code を多く含む．

**仮説 4** 仮説 2，仮説 3 と比較すると生存期間の短いコードクローンの方が非コードクローンより多くの欠陥を含む．

これらの仮説が成り立つがどうかについて以降で説明する．

**仮説 1 の検証** この仮説は 4.2 節で示した調査によって検証する． 図 4.2 節の調査の結果で示したように，コードクローンに包含されるコード片を 1 つの単位として Buggy Code の含まれる割合を調べた．コード片の生存期間を中央値を境として長いグループと短いグループに分けたとき，生存期間が短いコード片のほうがより多くのコードクロー

ンを含んでいた．従って本調査で調べた Evolution プロジェクトと Gimp プロジェクトにおいては Buggy Code 内に生存期間の短いコードクローンが含まれる割合が高いと言える．

**仮説 2 の検証** コードクローンとなるコード片に対して生存期間による Buggy Code の含む割合の変化は表 6，図 13，図 14 の $\overline{RC}(Di)(i = 1..4)$ を比較することによって評価できる．$\overline{RC}(Di)$ は，調査した 5 つのプロジェクトのうち Evolution を除く 4 つのプロジェクトに対して単調減少している．このことから，多数のプロジェクトにおいて，生存期間の短いコードクローンは，生存期間が長いコードクローンより Buggy Code を含む割合が高いと言える．Evolution プロジェクトに関しては，$\overline{RC}(D2)$ の値が一番高くなり，$\overline{RC}(D3)$ の値が $\overline{RC}(D4)$ よりも低くなり，仮説を満たさない結果を示している．しかし，生存期間の集合を中央値で分け，$D_{short}$ と $D_{long}$ の 2 種類に分けたときに同様の評価を行うと，表 7 の示すように $D_{short}$ の方が Buggy Code を含む割合が高くなり，仮説を満たす傾向が得られる．Evolution プロジェクトにおいて $\overline{RC}(D2)$ の値が高くなった理由として，Evolution プロジェクトは他のプロジェクトよりも Buggy Code 率が低いことが考えられる．表 3 で示すように，Evolution プロジェクトの Buggy Code 率は他のプロジェクトの半分程度しかない．したがってある Buggy Code を含むコミットが特定の期間に集中しやすくなる傾向があると考えられる．

**仮説 3 の検証** コードクローンでないコード片に対して生存期間による Buggy Code を含む変化も仮説 2 と同様に 表 6，図 13，図 14 の $\overline{RU}(Di)(i = 1..4)$ を比較することによって評価できる． $\overline{RU}(Di)$ は，調査した 5 つのプロジェクトのうち Ant, Evolution において単調減少している．それ以外のプロジェクトではクトでは $\overline{RU}(D3)$ の値が $\overline{RU}(D2)$ より高い結果となっている．しかしこれらの差異は，$\overline{RU}(D1)$ と $\overline{RU}(D3)$ の差異に比べ小さく，表 7 と同様に生存期間の集合を $D_{short}$ と $D_{long}$ に分けた場合，表 8 に示すように，全てのプロジェクトで生存期間が短い集合の方が Buggy Code を含む割合が高くなっている．このことから，コードクローンでないコードに対しても，生存期間が短いコード片が多くの Buggy Code を含む傾向があると言える．

表 7: Evolution プロジェクトにおける Buggy Code の含む割合

| | | $D_{short}$ | $D_{long}$ |
|---|---|---|---|
| Evolution | $\overline{RC}(Di)$ | 2.95 | 1.00 |
| | $\overline{RU}(Di)$ | 1.53 | 1.00 |

**仮説 4 の検証** 仮説 2，仮説 3 から生存期間が短いコードクローン，非コードクローンともに Buggy Code の含む割合が高い傾向が得られた．次にこれらの結果からコードクローンによる影響を検証する．コードクローンの影響を検証するには，$\widetilde{RC}(Di)$ と $\widetilde{RU}(Di)$ の値を比較すればよい．生存期間 $D1$ に関して比較すると，$\widetilde{RC}(D1)$ と $\widetilde{RU}(D1)$ では，Nautilus プロジェクトと Eclipse プロジェクトでは $\widetilde{RC}(D1)$ の方が大きくなり，コードクローンの方がより多くの Buggy Code を含んでいるが，その他のプロジェクトでは逆の結果になっている．このことから，生存期間の短いコードクローンがコードクローンでないコードより Buggy Code を含む強い傾向は見られなかった．しかし，生存期間 $D4$ に関して比較すると，全てのプロジェクトにおいて $\widetilde{RC}(D4)$ が $\widetilde{RU}(D4)$ より小さい値となっている．このことは生存期間の長いコードクローンは，コードクローンでないコードよりもより Buggy Code を含む割合が小さくなるという結果を表している．このような結果が得られた理由としては，生存期間が長いコードクローンはよく知られたプログラミングロジックが再利用されている可能性が高く，そのようなコードは欠陥を含む可能性が小さいからだと予想される．このことから，コード片の生存期間に着目することで，より欠陥を含みやすいコードクローンとそうでないものに分類できると考えられる．

### 6.2 結果の有用性

本研究の調査結果をまとめると以下のような傾向が得られたと言える．

- 生存期間の短いコードは生存期間が長いコードに比べ Buggy Code を多く含む．これはコードクローンとコードクローンでないコードの両方でこの傾向が得られた．
- 生存期間が長いコードクローンは，生存期間が長いコードクローンでないコードに比べ，Buggy Code を含む割合が少ない．

表 8: Gimp, Nautilus, Eclipse JDT プロジェクトにおける Buggy Code の含む割合

| | | $D_{short}$ | $D_{long}$ |
|---|---|---|---|
| Gimp | $\overline{RC}(Di)$ | 2.69 | 1.00 |
| | $\overline{RU}(Di)$ | 1.43 | 1.00 |
| Nautilus | $\overline{RC}(Di)$ | 2.24 | 1.00 |
| | $\overline{RU}(Di)$ | 4.05 | 1.00 |
| Eclipse JDT | $\overline{RC}(Di)$ | 2.16 | 1.00 |
| | $\overline{RU}(Di)$ | 1.53 | 1.00 |

既存手法である Rahman らの手法では，コード片がコードクローンであるかコードクローンでないかのみでしか分類していないので，このような結果は得られていない．本研究では，生存期間に応じてコードを分類することにより，生存期間の長さによって欠陥コードの含む割合に差異が生じることを確認できた．

また，この結果はコードの品質評価に応用できると考えられる．生存期間を取得することによって，より生存期間の短いコード片は欠陥を含む可能性が高いので，優先的に確認をする必要がある．またコード片の生存期間は版管理システムでソースコードが管理されていれば取得可能であるので，簡単に利用できる．

コードクローンに関する応用として生存期間はコードクローンのリファクタリング候補の順位付けの指標として利用できると考えられる．ソフトウェア開発現場で扱うプロジェクトの多くはコードクローン検出を行った際に大量のコードクローンが抽出され，全てのコードクローンに対してリファクタリングを検討することは現実的でない．そのため，コードクローンのリファクタリング候補の抽出を行う必要がある．現在までにコードクローンのリファクタリング候補の抽出を行う手法は提案されている [11] が，コード片の生存期間を考慮していない．そこでコードクーンのリファクタング候補の抽出の際にコード片の生存期間を指標の 1 つとすれば，より欠陥に関連のないコードクローンを除去できると考えられる．

## 7　妥当性の検証

本研究で行ったコード片の生存期間が与える影響調査に関する妥当性を，内的妥当性と外的妥当性に分けて検証する．

**内的妥当性**　本研究では欠陥と考えられるコードを Buggy Code と定義し，欠陥管理システムと版管理システムのコミットログを対応付けることで Buggy Code を特定している．この対応付けに使用しているコミットログは作業者が自由に記述できるため，この Buggy Code の対応付けの精度は作業者の書くコミットログに依存する．また全ての欠陥修正がコミットログに記述されないので，すべての Buggy Code を特定することは困難である．しかしコミットログの記述方式はある程度記述様式が定められており，大規模なプロジェクトであればコミットログの様式による差異は小さいと考えられる．Buggy Code の全てが本当に欠陥に関連しているかどうかも定かではない．例えば 100 行の修正が Buggy Code としてが検出されても，本当に欠陥を引き起こしているコードはその中の 1 行という可能性もある．しかし，作成者に直接聞く方法は効率的ではないため，客観的な手法で大量のコードの中から欠陥を含むコードを特定できる妥当な方法と考えられる．

本研究で使用した版管理システムは Git であるが，Git は全ての更新履歴を格納していない．Git はデータベースを分散させることができるため，1 つのデータベースに更新を集約してもコミットが含まれなかったり，コミット時刻が実際にコミットを行った時刻と差異が生じる可能性がある．このため本研究で取得した生存期間は厳密に正確な期間を表していない可能性がある．他の版管理システムを対象とした手法も提案されている [3] が，近年のオープンソースソフトウェアは Git でソースコードを管理しており，他の既存研究でも Git を使用しているので本研究でも Git を対象とした．

欠陥管理システムからのデータ抽出に関しても誤差が生じる．欠陥管理システムは自由にユーザが欠陥を報告できる場合がある．その場合本来欠陥でない情報が欠陥情報として取得される可能性がある．本手法では Status が “Resolved” となる欠陥情報の ID を使用しているが，作成者の投稿頻度や緊急度が高いもののみを選べばこのような誤差が減らせると考えられる．

**外的妥当性**　本研究の調査対象は全てオープンソースソフトウェアであるので，商用ソフトウェアなどオープンソースソフトウェアでないものに関しては結果が異なる可能性がある．また対象としたプログラミング言語は C と Java だけであり，他の言語を使用した時の生存期間が与える影響は調査できていない．本研究ではコードが 10 万行以上のプロジェクトを対象としており，それ以下の規模のプロジェクトではコード片の生存期間はばらつきが生じるため，本研究では示した調査の傾向が現れない場合がある．しかし本研究で示したように，

ある程度プロジェクトの規模が大きく，欠陥報告やコミット数がある程度以上存在すれば差異が生じると考えられる．

## 8 関連研究

版管理システムや過去のスナップショットを用いてコード片の修正履歴を取得し，それらのコード修正が欠陥に関連するかを調べる手法は数多く提案されている． Elmar らはコードクローンに対して一貫性の無い修正に着目し，コードクローンの一貫性の無い修正が欠陥に起因しているかを調査している [6]．この調査では大規模な商用ソフトウェアとオープンソースソフトウェアを対象としており，以下の仮説を検証している．

**RQ1** コードクローンが一貫して変更されているか ．

**RQ2** 一貫した変更でない場合，それらの変更は意図して変更されているか．

**RQ3** 意図的な変更でない場合，それらの変更は欠陥に起因しているか．

調査の結果，全体の約半分のコードクローンが一貫性の無い修正が加えられていた．また，それらの修正の意図を修正者に尋ねてみたところ，一貫性の無いコードクローンのうち約 4 分の 1 が意図されていない修正であった．さらにその修正のうち約 3〜23%が欠陥を含んでいたと報告されている．このように Elmar らは一貫性の変更を加えたコードクローンは欠陥に起因することを報告したが，開発者に修正の意図を尋ねる手間が大きかったため，特定のシステムでいくつかのスナップショットに対してしか調査していない．

Kim らは版管理システムを用いて複数バージョン間でコードクローンの推移を調査している [16]．この調査ではコードクローンの推移を 6 種類の組み合わせで表現できるモデル (Evolution Pattern) を提案し，コードクローンの変化における系統図 (Genealogy) を作成した．系統図からはコードクローンの生存期間が取得でき，それらの生存期間とコードクローンがリファクタリング可能かどうかを調査した．その結果，多くのコードクローンは生存期間が短く，途中でコードクローンが消滅しており，また生存期間が長いコードクローンの多くはリファクタリングが困難なものが多いという結果を報告している．リファクタリングが困難である理由としては，プログラミング言語の制約上 1 つにコードがまとめられなかったり，ロジックが抽象化できないコードが多かったためと説明している．これらの結果からリファクタングは必ずしも有効でないという結果を述べている．この調査ではコードクローンの生存期間とコードクローンがリファクタリング可能かどうかの関係を調べており，本研究ではコードクローンの生存期間と欠陥修正の関係を調査している．

また，Kim らは版管理システムを用いて欠陥修正が起こりやすい場所の特定を行っている．これらはコードをエンティティ単位 (関数やメソッド) に分け，欠陥が生じうる以下の 4 つの原因を仮定している．

**変更の発生** エンティティに変更が加えられた場合，欠陥が生じやすい．

**新規作成** エンティティが新規に追加された場合，欠陥が生じやすい．

**欠陥による修正** エンティティが欠陥修正された場合，再度欠陥が生じやすい．

**論理的に類似したコードの修正** 過去の履歴を調べ，同時に変更された回数が多いエンティティ群は，そのうちの 1 つが変更された場合，他のエンティティに欠陥が生じやすい．

これらの項目を用いて欠陥を抽出した場合，効率的に欠陥が抽出することができたと報告している．この手法においてもコード片が過去に変更されたという情報を用いているが，本研究ではコード片の生存期間を用いてより欠陥が含まれる割合が高いコード片の抽出を行っている．

川口らはコードクローンの履歴分析を行うことで編集が施される前のコードクローン分析を行っている [19]．コードクローンはソフトウェアが進化するにつれて編集が加わり変化するので，最新のソースコードだけを解析してもコードクローンが検出できない場合がある．例えば，リビジョン $r$ でクローンセット $C$ 内のコード片に異なる修正が加えらるとリビジョン $r+1$ ではクローンセット $C_1$，$C_2$ に分かれ $C_1$，$C_2$ はそれぞれ別のクローンセットとして検出される場合がある．さらに $C_1$ に欠陥が発見された場合，クローンセット $C_2$ も $C_1$ と同じクローンセットから派生しているので欠陥の有無を確認する必要があるが，最新のソースコードからはコードクローンとして検出することができない．そこで，この手法ではこれらの派生したクローンセットを分岐クローンセットと定義し，コードクローンの履歴を用いることで分岐クローンセットの解析を行っている．この手法はコードクローンの過去の位置情報を特定するためにリポジトリの diff 機能を用いている．diff 機能から得られる情報ではコード片の位置情報を一意に特定できない場合が存在するので，さらにコード片のテキスト類似度を定義しコード片の位置を特定している．川口らの手法はコードクローンの履歴情報を用いて欠陥が生じた時の修正候補の提示を行っているが，本研究は diff 機能と blame 機能を用いてコードクローンの履歴情報から生存期間を取得し，それらが欠陥コードにどのように影響するかを分析している．

## 9　むすび

コードクローンの有無がソフトウェアの保守性に影響するかに関してさまざまな議論がある．近年の研究では欠陥管理システムの欠陥情報と版管理システムのコミットログを対応付けることによって欠陥修正が生じたコードを特定し，特定したコードクローンの欠陥に与える影響を報告していた．既存研究ではコード片がコードクローンかコードクローンでないかでのみ判断していたので，我々はコード片の生存期間で分類することによって，より多くの欠陥修正を含むコードクローンの特定を試みた． 1 つ目の調査では 2 つのオープンソースソフトウェアに対して生存期間の長いコードクローンと生存期間の短いコードクローンに分けたところ，生存期間が短いコードクローンの方が多くの欠陥を含まれている結果となった． 2 つ目の調査では，コード片を 1 行単位で扱い 4 つの集合に分類することで，生存期間が短いコードクローンには欠陥修正が多く含まれ，また生存期間が短いコード片は欠陥修正である割合が高いかどうかを分析した．その結果，生存期間が短いコードクローンとコードクローンでないコードは，より多くの欠陥修正コードを含んでいることが分かった．また生存期間が長いコードクローンは，生存期間が長いコードクローンでないコードに比べて欠陥コードの含まれる割合が小さいことが分かった．これらの結果を応用して，コード片の生存期間をコードクローンの品質評価やリファクタングによる除去候補の算出に利用することが出来ると考えられる．

## 謝辞

本研究の全過程を通して，常に適切な御指導および御助言を賜り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 井上 克郎 教授に心より深く感謝いたします.

本研究を通して，随時適切な御指導および御助言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 松下 誠 准教授に深く感謝いたします.

本研究において，常時適切な御指導および御助言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 石尾 隆 助教に深く感謝いたします.

本研究に対して，終始適切な御指導および御助言を頂き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 吉田 則裕 助教に深く感謝いたします.

最後に，その他様々な御指導，御助言等を頂い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井上研究室の皆様に深く感謝いたします.

## 参考文献


[1] J.Śliwerski, T. Zimmermann, and A. Zeller. When do change induce fixes? In *Proceedings of the International workshop on Mining software repositories (MSR '05)*, pp. 1–5, 2005.

[2] Ant. `http://ant.apache.org/`.

[3] A. Bachmann and A. Bernstein. Data retrieval, processing and linking for software process data analysis. Technical report, Dynamic and Distributed Information System Group, Departmenet of Informatics, University of Zurich, 2009.

[4] M. Balazinska, E. Merlo, M. Dagenais, B. Lagüe, and K. Kontogiannis. Advanced clone-analysis to support object-oriented system refactoring. In *Proceedings of the 7th Working Conference on Reverse Engineering(WCRE'00)*, pp. 98–107, Los Alamitos, USA, 2000.

[5] I. Baxter, A. Yahin, L. Moura, M. Anna, and L.Bier. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance(ICSM'98)*, pp. 368–377, 1998.

[6] B. Hummel E. Juergens, F. Deissenboeck and S. Wagner. Do code clones matter? In *Proceedings of the International Conference on Software Engineering(ICSE'09)*, pp. 485–495, 2009.

[7] Evolution. `http://projects.gnome.org/evolution/`.

[8] M. Fowler. *Refactoring: improving the design of existing code.* Addison Wesley, 1999.

[9] F.Rahman, C.Bird, and P. Devanbu. Clones: What is that smell? In *Proceedings of the Seventh IEEE Working Conference on Mining Software Repositories(MSR' 10)*, pp. 72–81, Cape Town, South Africa, 2010.

[10] Gimp. `http://www.gimp.org/`.

[11] Y. Higo, S. Kusumoto, and K. Inoue. A metric-based approach to identifying refactoring opportunities for merging code clones in a java software system. *Journal of Software Maintenance and Evolution*, Vol. 20, No. 6, pp. 435–461, 2008.

[12] Eclipse JDT. `http://www.eclipse.org/jdt/`.

[13] L. Jiang, Z. Su G. Misherghi, and S. Glondu. DECKARD :scalable and accurate tree-based detection of code clones. In Proceedings of the 29th International Conference on Software Engineering (ICSE '07), pp. 96–105, Minneapolis, USA, 2007.

[14] L. Jiang, Z. Su, and E. Chiu. Context-based detection of clone-related bugs. In *Proceedings of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering(ESEC-FSE '07)*, pp. 55–64, Dubrovnik, Croatia, 2007.

[15] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, Vol. 28, No. 7, pp. 654–670, 2002.

[16] M. Kim, L. Bergman, T. Lau, and D. Notkin. An empirical study of code clone genealogies. In *Proceedings of of the Joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering(ESEC/FSE'05)*, pp. 187–196, 2005.

[17] Nautilus. `http://live.gnome.org/Nautilus`.

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

[19] 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.

[20] 吉田則裕, 肥後芳樹, 神谷年洋, 楠本真二, 井上克郎. コードクローン間の依存関係に基づくリファクタリング支援. 情報処理学会論文誌, Vol. 48, No. 3, pp. 1431–1442, 2007.

[21] 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. 91-D, No. 6, pp. 1465–1481, 2008.

[22] 肥後芳樹, 神谷年洋, 楠本真二, 井上克郎. コードクローン を対象としたリファクタリング支援環境. 電子情報通信学会論文誌, Vol. 88-D-I, No. 2, pp. 186–195, 2005.