

CRDを用いた コードクローンの生存期間と修正回数に関する調査

堀田 圭佑^{1,a)} 肥後 芳樹^{1,b)} 楠本 真二^{1,c)}

受付日 2013年5月13日, 採録日 2013年11月1日

概要: これまでに、コードクローンの進化の様子を分析し、その特徴を調査する研究が多数行われている。しかし、これまでの調査手法には、コードクローンに対して大きな修正が加えられた場合に追跡に失敗するおそれがあるという課題がある。また、ソフトウェア保守に悪影響を与えるコードクローンがどの程度存在するのかという点や、コードクローンが修正される時期に何らかの特徴があるかという点は明らかになっていない。修正される時期の特徴が明らかになれば、コードクローンへの対処の是非を検討する指標になると期待できる。本研究では、ソースコードに対する変更に強いコードクローン追跡手法である CRD を用いて、これまでに明らかにされていない事項に対する調査を行った。調査の結果、コードクローンの多くは短命であり、かつ修正されにくいという既存研究の知見を改めて確認した。また、コードクローンは生存期間の前半に修正されやすいことが明らかになった。さらに、生存期間が長く、かつ修正回数の多いコードクローンが全体の約 3.3% 存在することが分かった。

キーワード: コードクローン, CRD, ソフトウェア進化

A CRD-based Analysis on Survival Periods and Modifications of Code Clones

KEISUKE HOTTA^{1,a)} YOSHIKI HIGO^{1,b)} SHINJI KUSUMOTO^{1,c)}

Received: May 13, 2013, Accepted: November 1, 2013

Abstract: Many researchers analyzed evolution of code clones based on clone tracking. However, clone tracking used in previous research has an issue that it might miss links of clones among revisions if a large modification was applied to them. In addition, there are some characteristics that the previous research did not reveal. This research revisits findings of previous research with a clone tracking technique that has a good change-tolerance, named CRD. Our experimental results support findings of previous research, including ‘most of clones are short-lived’, and ‘there are a few clones that are modified multiple times’. In addition, the results showed that about 3.3% of clones are long-lived and modified multiple times. Moreover, we found that there is no obvious trends on the timing when clones are modified.

Keywords: code clone, CRD, software evolution

1. まえがき

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）が効率的なソフトウェア保守を阻害する要

因として糾弾され、注目を集める研究分野となって久しい [1], [2], [3], [4], [5], [6]. コードクローンによってもたらされる弊害を防ぐために、ソースコード中からコードクローンを自動的に検出する手法や、その除去作業を支援する手法がこれまでに多数提案されている [7], [8], [9].

コードクローンへの効率的な対処（除去や監視など、コードクローンに対して注意を払うすべての行為 [2]）を実現するためには、コードクローンが持つ特性を理解する必要がある。この考えに基づき、コードクローンを分析する研

¹ 大阪大学大学院情報科学研究科
Graduate School of Information Science and Technology,
Osaka University, Suita, Osaka 565–0871, Japan

a) k-hotta@ist.osaka-u.ac.jp

b) higo@ist.osaka-u.ac.jp

c) kusumoto@ist.osaka-u.ac.jp

究が多数行われている [10], [11], [12], [13]. 特に, ソフトウェアの開発履歴情報の分析を通じてコードクローンがどのような進化をたどっているのかを明らかにするという調査手法が広く採用されている [10], [11]. この手法に基づいた調査を行うためには, 修正前後のソースコード間でコードクローンの対応付けを行うこと, すなわちコードクローンの追跡を行うことが必要となる. 既存研究は, コード片の文字列の類似度を算出し, 高い類似度を持つコード片を対応付けることでコードクローンの追跡を実現している. しかしこの追跡方法には, コードクローンを構成するコード片に大きな修正が施された場合, 対応付けができず, 追跡に失敗するおそれがあるという課題点が存在する.

また, 近年の調査の結果, すべてのコードクローンがソフトウェア保守に悪影響を与えているとはいえず, 一部のコードクローンのみが対処を必要とするものであることが指摘されている [11], [14]. 対処を必要とするコードクローンの特性として, 以下のものがあげられる.

- 生存期間が長いこと
- 頻繁に修正されること

生存期間の短いコードクローンへ対処を行った場合, 対処したコードクローンがすぐに消滅してしまうため, かけた労力に見合う恩恵が得られない [10]. また, コードクローンによってもたらされる悪影響の最たるものとして, 修正漏れによる不具合が指摘されていることから, 修正されない安定したコードクローンへ対処を行うことも効果的であるとはいえず [14], [15], [16].

既存研究により, コードクローンの生存期間やコードクローンに加えられる修正の回数などは調査されているが, 生存期間が長くかつ多数修正されているコードクローンがどの程度存在するのかという点は明らかになっていない.

また, コードクローンがその生存期間のどの時点で修正されるのかという点も効率的なコードクローンへの対処を考えるうえで重要である. たとえば, コードクローンが生成された直後に頻繁に修正され, その後すぐに安定するのであれば, 生成されてから一定期間経過したコードクローンへ対処することは効果的ではなく, 生成直後に対処を行うことが必要であると考えられる.

そこで本研究では, ソースコードに対する変更に近いコードクローン追跡手法である CRD [17] を用いることで既存研究の課題点を解決するとともに, まだ明らかにされていないコードクローンの特性を調査する.

6つのソフトウェアに対する調査の結果, 以下のことが明らかになった.

- コードクローンの多くは短命であること, また複数回修正されるコードクローンは少ないことが報告されているが, CRD を用いた場合でもそれらが成り立つ.
- コードクローンの生存期間と修正回数の間に相関はない.

- コードクローンは生存期間の前半に修正されやすい傾向にあるが, 生存期間の後半においても修正されることがある.

- 生存期間が長く, かつ複数回修正されるコードクローンは, すべてのコードクローンのうち約 3.3% である. これらの調査結果を総合すると, 生存期間が長いコードクローンの中にはほとんど修正されないものもあれば, 頻繁に修正されるものも存在しており, コードクローンへの対処が効果を発揮する事例が実在するといえる. また, コードクローンは生存期間の前半に修正されやすいという知見は, なるべく早期にコードクローンに対処を行うことが重要であることを示唆している. ただし, コードクローンの多くは生存期間が短く, かつ修正されにくいものであるということも改めて確認できたという点から, 対処するコードクローンを慎重に選択する必要があるといえる.

以降, 2章でこれまでの調査報告を紹介するとともに, 本研究の調査内容を述べる. 3章および4章で調査手法について述べ, 5章で調査手法を実装したツール, ならびに調査対象に触れる. 6章で調査結果を報告し, 調査結果に対する議論を7章で展開する. 最後に8章で本稿をまとめる.

2. 調査の目的

2.1 コードクローンの進化に関する既存研究

これまでに, コードクローンがどのように進化しているのかを調査する研究が行われている. Kim らの研究はそれらのさきがけとなったものである [10]. この研究では, CCFinder [7] を用いて検出されたコードクローンを追跡し, その生存期間の長さを調査している. 調査の結果, コードクローンの多くは生存期間が短いことが明らかとなり, それをふまえ Kim らはコードクローンの除去によって得られる恩恵は小さいと主張している.

Göde らはコードクローンがその進化の過程でどの程度修正されているのかを調査した [11]. その結果, コードクローンのほとんどは1度も修正されないか, 1度しか修正されないかのいずれかであり, 複数回修正されるものは全体の約12%程度であったことが報告されている.

Thummalapenta らはコードクローンに加えられる修正のパターンに関する調査を行った [18]. その結果, コードクローンを構成するコード片のうち, 一部に対して修正を施し忘れる, いわゆる修正漏れはほとんど起きていないことが明らかになった. その一方で, コードクローンに対する修正漏れは数こそ少ないものの, 不具合に直結する危険性が高いということもあわせて報告されている.

これらの研究以外にも, コードクローン進化の分析を通じて, コードクローンがソフトウェアの開発, 保守に与えている影響を定量的に明らかにしようとする研究が多数行われている. これらの調査の結果, コードクローンに対する修正漏れが発生する頻度は低いこと [19], [20], [21], [22],

また、コードクローンに起因する不具合の数は多くないこと [12], [23] が指摘されている。

さらに、コードクローンに加えられる修正の頻度を比較する研究が多くの研究者によって行われている [14], [15], [16]. これらの研究の結果、コードクローンはコードクローンとっていない箇所と比較して修正されにくいことが明らかになっている。

その一方で、コードクローンに対する修正漏れが多数存在し、またそれらによって多くの不具合が引き起こされているとする結果も報告されている [24], [25].

これらの調査結果は、「すべてのコードクローンがソフトウェアの開発、保守に悪影響を与えているとはいえないが、一部そのようなコードクローンが実在する」ことを示唆している。したがって、コードクローンに対して効率的に対処を行うためには、対処を要するコードクローンを特定し、それらに注力することが必要であるということが出来る [11].

2.2 調査項目

本研究では、コードクローンの追跡を通じてコードクローンの進化の様子を分析し、その特性を明らかにすることを目指す。本調査の目的は以下の2点である。

- コードクローンの追跡を行う際に、変更に強い追跡手法を用いることで、これまでの研究で得られた知見に差異が生じるか否かを確認する。
- これまでの研究で明らかにされていないコードクローン進化の特性を明らかにする。

これらの調査目的を達成するために、以下にあげる4つの研究課題を設定した。

RQ1: 既存研究 [10] から得られた‘コードクローンの多くは生存期間が短い’という知見は CRD を用いた場合でも成立するか。

RQ2: 既存研究 [11] から得られた‘複数回修正されるコードクローンは少ない’という知見は CRD を用いた場合でも成立するか。

RQ3: コードクローンの生存期間の長さとコードクローンに加えられる修正回数の間に正の相関はあるか。

RQ4: コードクローンは生存期間の前半に修正されやすい傾向にあるか。

このうち、RQ1 と RQ2 は既存研究で調査されている項目であり、変更に強い追跡手法を用いた場合でも同様の知見が得られるのかを確認するために設定した。今回は比較対象として、Kim らの研究 [10], および Göde らの研究 [11] を選定した。これらの研究を比較対象として選択した理由は、これらの研究が他の研究報告から多数引用されており、これらの研究から得られている上述の知見がコードクローン分析の研究に与えている影響が大きいためである。

一方、RQ3 および RQ4 はこれまでの研究で得られていない特性を明らかにするために設定した項目である。生存

期間の長さと修正回数の間に正の相関があれば、生存期間の長いものは修正の回数が多くなるため、生存期間の長いものに注力することが修正回数が多いいものに注力することにもつながるといえる。また、コードクローンがその生存期間の前半に修正されやすい傾向にあるのであれば、コードクローンに対して何らかの対処を行う場合、そのコードクローンの生存期間の早い段階、すなわち生成された後なるべく早い段階で対処を始めることが、効率的なコードクローンへの対処の実現につながると考えられる。

また、本研究における調査の特色として、生存期間の長さと修正の回数を同時に調査しているという点がある。生存期間の短いコードクローンは、生成された後すぐにコードクローンではなくなるため、そのようなコードクローンへ対処を行うことは効率的ではないことが指摘されている [10]. 換言すると、生存期間が長いコードクローンは、生成された後も長期間にわたりコードクローンとして存在し続けるため、そのようなコードクローンへの対処は生存期間の短いものと比べて効果を発揮しやすいといえる。また、コードクローンによる弊害はコードクローンに対して何らかの修正が加えられた場合に引き起こされることが多いと考えられるため、修正回数が多いいコードクローンは特に注意を要するといえる。したがって、生存期間が長く、かつ修正回数の多いコードクローンに対して対処を検討することが、効率的なコードクローンへの対処を実現するうえで重要であるといえる。

3. コードクローンの検出方法

本調査では、対象ソフトウェアのすべてのリビジョンに対してコードクローン検出を行う必要がある。そのため、調査を実現するためには高速なコードクローン検出方法を用いる必要がある。

そこで本研究では、ブロック単位のコードクローン検出手法を採用した。このコードクローン検出手法における処理の手順は以下のとおりである。

ステップ1: 字句解析および構文解析

ステップ2: ブロックの特定

ステップ3: 正規化

ステップ4: ハッシュ値の算出

ステップ5: コードクローンの特定

図1に本研究で用いるコードクローン検出手法の概要を示す。以降、それぞれのステップについて述べる。

ステップ1: 字句解析および構文解析

入力として与えられたソースコードに対して、字句解析および構文解析を行う。このコードクローン検出手法はブロック単位での検出を行うため、ソースコード中のブロックを特定する必要がある。そのため、字句解析に加えて構文解析を行う必要がある。

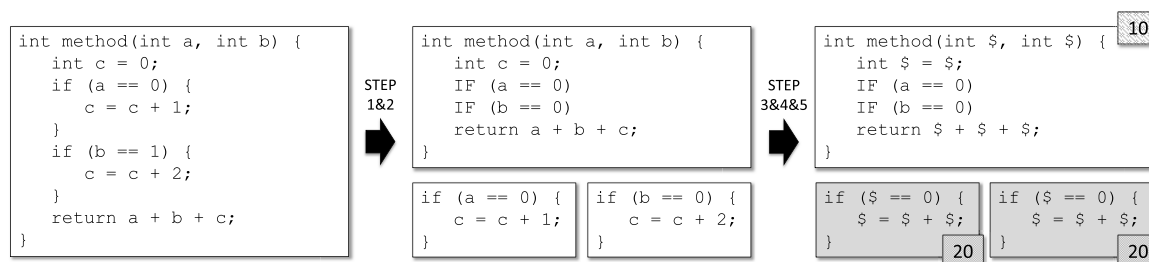


図 1 コードクローン検出手法の概要

Fig. 1 An overview of the clone detection method.

ステップ 2：ブロックの特定

このステップではブロックの特定を行う。ここで“ブロック”には、各種ブロック文に加え、クラス、メソッド、関数なども含まれている。さらに、このステップではサブブロックを特殊な文字列に置き換える処理もあわせて行う。この特殊な文字列には以下の情報が含まれている。

- ブロックの種類
- 条件式（条件式を持つ場合のみ）
- シグネチャ（メソッド、関数の場合のみ）
- クラス名（クラスの場合のみ）

この処理により、ブロックで行われている処理の流れは同じであるが、サブブロックで行われている処理が異なる場合であっても、それらのブロックをコードクローンとして検知することが可能となる。

ステップ 3：正規化

各ブロックに含まれる変数名ならびにリテラルを特殊文字に置換する。これにより、変数名やリテラルのみが異なるようなコードクローンを検出することが可能となる。

ステップ 4：ハッシュ値の算出

正規化後の各ブロックを文字列に変換し、その文字列をもとにハッシュ値を算出する。

ステップ 5：コードクローンの特定

各ブロックから得られたハッシュ値を比較し、コードクローンの検出を行う。複数のブロックが同じハッシュ値を持っている場合、それらのブロックは正規化後の文字列がすべて等しい。したがって、同じハッシュ値を持つブロックの集合が 1 つのコードクローンであると見なされる。

4. コードクローンの追跡

4.1 Clone Region Descriptor

Clone Region Descriptor（以降、**CRD**）とは、Duala-Ekoko らによって提案された、コードクローンを構成するコード片のおおよその位置情報を示す表現形式である [17]。CRD はコード片の位置を表しており、コード片が属するブロックなどに変化がなければ、コード片そのものに変更

```
<CRD> ::= <file> <class> <CM> [<method>]
<method> ::= <signature> <block>*
<block> ::= <btype> <anchor> <CM>
<btype> ::= 'for' | 'while' | 'do' | 'if' | 'else' | 'switch'
          | 'try' | 'catch' | 'finally' | 'synchronized'
```

(a) 定義

```
public class DeleteManager {
    ...
    public void delete(int n) {
        ...
        for (int i = 0; i < delete.size(); i++) {
            ...
            if (delete.get(i) instanceof ElementNode) {
                // some code
            }
            ...
        }
    }
}
```

CRD

```
packagename.DeleteManager.java,DeleteManager,5
delete(int),5
for,delete.size(),4
if,delete.get(i) instanceof ElementNode, 2
```

(b) 例

図 2 CRD の定義と例

Fig. 2 Definition and example of CRD.

が加えられても CRD は変化しない。

図 2 に CRD の定義、およびその具体例を示す。CRD の定義は図 2(a) のようになっている。CRD は Java を対象に定義されているため、ブロックの種類には Java に存在するもののみが現れている。ここで、定義中の anchor はそのブロック特有の文字列を表しており、条件式などが該当する。また、signature はメソッドのシグネチャを、CM は各ブロックのメトリクス値（サイクロマチック数+fan-out 数）をそれぞれ表している。次に、CRD の例を図 2(b) に示す。この例には、最もネストの深い位置に存在する if 文に対する CRD を記載している。あるブロックに対する CRD は、そのブロックに加えてその外側に位置するすべてのブロックの情報をを用いて定義される。この例の場合、1 行目から 3 行目までの文字列は if 文の外側のブロックの情報をを用いて算出されたものであり、4 行目の文字列が着目している if 文そのものの情報をを用いて算出されたものとなっている。なお、この例は文献 [17] で用いられているものと同じものである。

4.2 コードクローンの系譜

本節では、コードクローンの系譜の定義を与える。なお

以降の説明では、対象ソフトウェアのそれぞれのリビジョンにはその識別のために一意な整数（リビジョン番号）が割り当てられており、また連続するリビジョン間のリビジョン番号は連続しているものと仮定している。

はじめに、互いにコードクローン関係にあるブロックの集合であるクローンセットを定義する。

定義 4.1 (クローンセット) B_r をあるリビジョン r に存在するブロックの集合とする。また、 $\text{hash}(b)$ をブロック b のハッシュ値とする。このとき、式 (1) および式 (2) をともに満たす、ブロックを要素とする集合 c をクローンセットと呼ぶ。また、以降ではリビジョン r に存在するすべてのクローンセットからなる集合を C_r と表記する。

$$\forall b_\alpha, b_\beta \in c [\text{hash}(b_\alpha) = \text{hash}(b_\beta)] \quad (1)$$

$$\forall b_\alpha \in c, \forall b_\beta \in B_r [(\text{hash}(b_\alpha) = \text{hash}(b_\beta)) \rightarrow b_\beta \in c] \quad (2)$$

なお本研究におけるクローンセットは、ハッシュ値が同じである、ということと同値としたときの同値類を表している。

次に、連続する2つのリビジョン間におけるブロックおよびクローンセットの対応関係をそれぞれ以下に定義する。

定義 4.2 (ブロックの対応関係) b_r, b_{r+1} をそれぞれリビジョン $r, r+1$ に存在するブロックであるとする。 b_r のCRDと b_{r+1} のCRDが一致するとき、 b_r と b_{r+1} は対応関係にあるといい、 $b_r \leftrightarrow b_{r+1}$ と表記する。

定義 4.3 (クローンセットの対応関係) リビジョン r に存在するクローンセット c_r と、リビジョン $r+1$ に存在するクローンセット c_{r+1} を考える。 c_r と c_{r+1} が以下の式 (3) を満たすとき、 c_r と c_{r+1} は対応関係にあるといい、 $c_r \leftrightarrow c_{r+1}$ と表記する。

$$\exists b_r \in c_r, \exists b_{r+1} \in c_{r+1} [b_r \leftrightarrow b_{r+1}] \quad (3)$$

なお、クローンセットの対応関係は連続する2つのリビジョン間に存在するクローンセット間のみに定義されるものである。したがって、連続していないリビジョン間や、同一のリビジョンに存在するクローンセットの間に対応関係は存在しない。

これらを用い、コードクローンの系譜を以下に定義する。

定義 4.4 (コードクローンの系譜) 互いに対応関係にあるクローンセットの集合をコードクローンの系譜（以降、系譜）と呼ぶ。すなわち、 C_{ALL} を検査対象期間において検出されたすべてのクローンセットからなる集合としたとき、以下の式 (4)、および式 (5) をともに満たす、クローンセットの集合 g を系譜と定義する。

$$\forall c \in g, \forall c' \in C_{ALL} [(c \leftrightarrow c') \rightarrow (c' \in g)] \quad (4)$$

$$\forall c \in g, \exists c' \in g [c \leftrightarrow c'] \quad (5)$$

4.3 コードクローンの生存期間

それぞれの系譜 g について、その始端リビジョン、終端リビジョン、ならびに生存期間を以下のように定義する。

定義 4.5 (始端リビジョン、終端リビジョン、生存期間) T を分析対象となるリビジョンすべてからなる集合であるとする。このとき系譜 g について、式 (6) を満たすリビジョン $first \in T$ を g の始端リビジョンと定義する。

$$\begin{aligned} \exists c_{first} \in C_{first} [c_{first} \in g] \wedge \\ \forall l \in T [l < first \rightarrow \forall c_l \in C_l (c_l \notin g)] \end{aligned} \quad (6)$$

同様に、式 (7) を満たすリビジョン $last \in T$ を g の終端リビジョンと定義する。

$$\begin{aligned} \exists c_{last} \in C_{last} [c_{last} \in g] \wedge \\ \forall l \in T [last < l \rightarrow \forall c_l \in C_l (c_l \notin g)] \end{aligned} \quad (7)$$

さらに、 g の生存期間 R を以下の式 (8) に定義する。

$$R = \{r \in T | first \leq r \leq last\} \quad (8)$$

4.4 コードクローンへの修正

本研究では、系譜 g に加えられた修正の回数を以下のように算出する。

定義 4.6 (修正回数) ある系譜 g 、およびその生存期間に含まれるあるリビジョン r を考える。 g, r が以下の式 (9) および式 (10) のうち1つ以上を満たすとき、 g はリビジョン r において修正されたと定義する。

$$\begin{aligned} \exists c_r \in C_r [c_r \in g \wedge \exists b_r \in c_r, \exists b_{r+1} \in B_{r+1} \\ [b_r \leftrightarrow b_{r+1} \wedge \text{hash}(b_r) \neq \text{hash}(b_{r+1})]] \end{aligned} \quad (9)$$

$$\begin{aligned} \exists c_r \in C_r [c_r \in g \wedge \exists b_r \in c_r, \forall b_{r+1} \in B_{r+1} [(b_r \not\leftrightarrow b_{r+1}) \\ \vee (b_r \leftrightarrow b_{r+1} \wedge \forall c_{r+1} \in C_{r+1} [b_{r+1} \notin c_{r+1}])] \end{aligned} \quad (10)$$

式 (9) は対応関係にあるブロックのハッシュ値が異なる、すなわちブロックを構成する文字列に修正が加えられた場合を表している。また式 (10) は対応関係にあるブロックがリビジョン $r+1$ に存在しない場合、あるいは対応関係にあるブロックがリビジョン $r+1$ ではないずれのクローンセットにも含まれていない場合を表している。

さらに、対象ソフトウェアの最終リビジョンに存在しない系譜は開発期間の途中で消滅しており、その消滅のきっかけとなる何らかの修正が加えられていると考えられる。このような理由から、本調査では系譜が最終リビジョンに存在しない場合、その系譜は終端リビジョンにおいて修正されたと見なす。すなわち、 T を対象ソフトウェアのすべてのリビジョンからなる集合とし、 $last_g$ を系譜 g の終端リビジョンとしたとき、以下の式 (11) が成立すれば g は $last_g$ において修正されたと見なす。以降、式 (11) により

特定された修正を“消滅に関わる修正”と呼ぶ。

$$\exists r \in T[last_g < r] \quad (11)$$

また、系譜 g が修正されたリビジョンの数を g に加えられた修正回数と定義する。

4.5 追跡の例

図 3 にコードクローンの追跡の例を示す。この例には、A, B, C, D, E, および F の 6 つのクローンセットから構成される系譜が 1 つ存在する。リビジョン r に存在するクローンセットは、次のリビジョンへの修正によりハッシュ値が変化している。したがって、式 (9) より、この系譜はリビジョン r において修正されたと判断される。次に、リビジョン $r+2$ への修正により要素数が増加しているが、本研究では要素数の増加は修正と見なさないため、リビジョン $r+1$ では修正は行われなかったものと見なされる。リビジョン $r+2$ からリビジョン $r+3$ への修正により、系譜に枝分かれが生じている。このとき、式 (9) が成立するため、この系譜はリビジョン $r+2$ において修正されたと見なされる。次にリビジョン $r+3$ からリビジョン $r+4$ への修正によって、D のクローンセットを構成するコード片に対応するブロックがリビジョン $r+4$ に存在しなくなっている。したがって、式 (10) より、この系譜はリビジョン $r+3$ で修正されたと判断される。以上より、この系譜に加えられた修正回数は 3 となる。

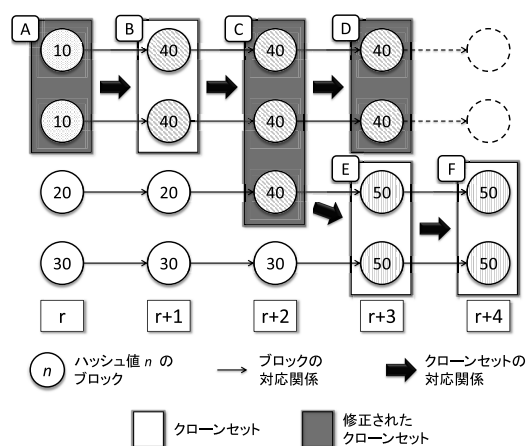


図 3 コードクローン追跡の例

Fig. 3 An example of clone tracking.

5. 調査の準備

5.1 調査手法の実装

本研究では、4 章で述べた調査手法を実装したツール **CloneTracker** を用いて調査を行う。CloneTracker は Java で記述されており、3 章で述べたコードクローン検出機能を内含している。CloneTracker は現在のところ、Java で記述されており、かつバージョン管理システム Subversion を用いて管理されているソフトウェアのみを解析可能である。字句解析、および構文解析には Java Development Tools (JDT) を使用している。また、この実装ではリレーショナルデータベースを内部で使用しており、データベース管理システムには SQLite を用いている。

検出速度をより高めるために、CloneTracker は増分的に解析を行う機能を有している。ここで、増分的な解析とは、リビジョン間で修正が加えられたファイルのみを新たに解析し、修正のなかったファイルについては以前の解析結果を流用するという解析方法を指す。これにより、ソースコードの解析に必要な計算コストを低減させている。

5.2 調査対象

本研究では、6 つのオープンソースソフトウェアを対象に調査を行った。それぞれのソフトウェアの概要を表 1 に示す。表中の“リビジョン数”は分析対象となったリビジョンの数、すなわちソースファイルに変更が加えられたリビジョンの数を示している。また“行数”は終了リビジョンにおける、調査対象以下のソースコードの行数を表している。

これらはすべて Java で記述されたソフトウェアであり、Subversion を用いて管理されている。これらのソフトウェアを選択した理由は以下のとおりである。

- 長い開発期間を有しており、また広く利用されているソフトウェアであるため (Ant, ArgoUML)。
- 著者らが開発したソフトウェアであり、詳細な分析が可能であるため (CloneTracker)。
- Kim らの研究 [10] で用いられているソフトウェアであるため (Carol, DNSJava)。
- Göde らの研究 [11] で用いられているソフトウェアであるため (JabRef)。

表 1 対象ソフトウェア

Table 1 Target software systems.

ソフトウェア	調査対象	開始リビジョン (日付)	終了リビジョン (日付)	リビジョン数	行数	系譜の数
Ant	/ant/core/trunk/src/main/	268,623 (2001/2/9)	909,962 (2010/2/14)	5,154	211,958	715
ArgoUML	/trunk/src/	1 (1998/1/27)	19,893 (2012/7/10)	3,918	362,783	2,760
CloneTracker	/	1 (2012/6/19)	419 (2013/5/3)	152	22,872	66
Carol	/trunk/carol/src/	9 (2002/8/6)	1,335 (2007/10/22)	250	17,251	258
DNSJava	/trunk/org/xbill/DNS/	2 (1998/9/6)	1,670 (2012/10/26)	1,285	22,512	406
JabRef	/trunk/jabref/src/java/net/	6 (2003/10/17)	3,718 (2011/11/11)	1,489	113,277	703

6. 調査結果

6.1 RQ1: ‘コードクローンの多くは生存期間が短い’ という知見は CRD を用いた場合でも成立するか

図 4 に各ソフトウェアにおける系譜の生存日数を示す。このグラフの x 軸は日数を、y 軸は生存日数がその日数以下である系譜数の百分率をそれぞれ表している。たとえば Ant の場合、生存期間が 400 日以下の系譜が全系譜のうち約 40% を占めていることが分かる。

いずれのソフトウェアについても、グラフの左側では急激に上昇しているが、右側では上昇が緩やかになっている。この傾向は特に DNSJava において顕著であり、生存期間が開発日数の 5% である 250 日以下という系譜が約 80% を占めている。

この結果は、Kim らの報告 [10] と同じであるといえる。したがって、RQ1 には **Yes** と回答することができる。

6.2 RQ2: ‘複数回修正されるコードクローンは少ない’ という知見は CRD を用いた場合でも成立するか

図 5 に各ソフトウェアにおいて系譜に加えられた修正回数を示す。グラフの x 軸は修正回数を棒グラフは修正回数が x 軸の値である系譜の数を表している（左軸）。また折れ線グラフは累積度数の百分率を表している（右軸）。

図 5 より、いずれのソフトウェアについても、ほとんどの系譜がまったく修正されないか 1 回だけ修正されるかのいずれかであることが分かる。

この傾向は Göde らが文献 [11] で報告したものと同じである。したがって、RQ2 に対する回答は **Yes** となる。

6.3 RQ3: コードクローンの生存期間の長さでコードクローンに加えられる修正回数に正の相関はあるか

図 6 に各ソフトウェアの各系譜ごとの修正回数を示す。グラフの x 軸は各系譜を生存期間（リビジョン数）の降順

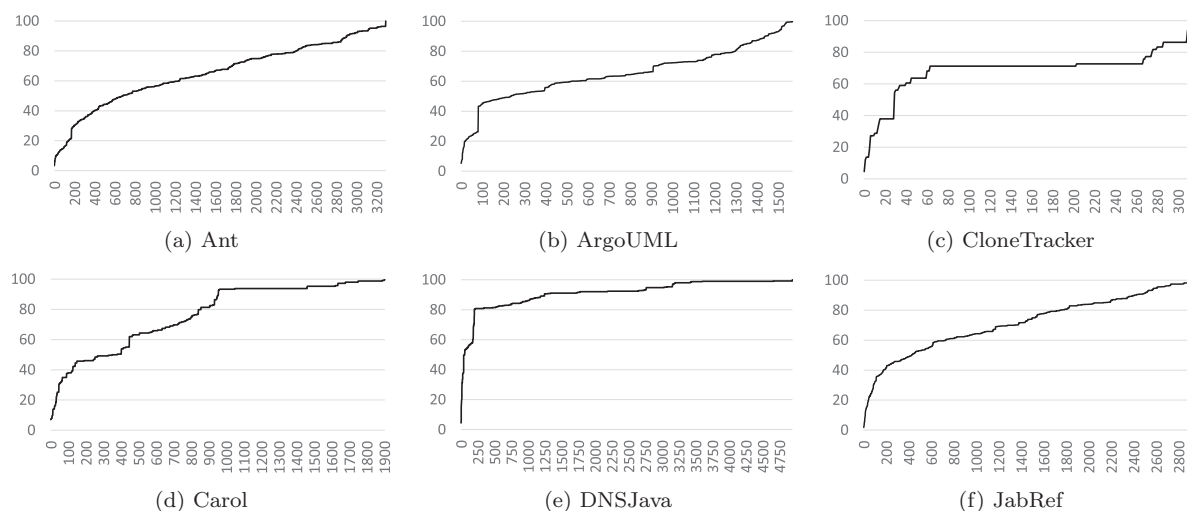


図 4 生存日数
Fig. 4 Alive days.

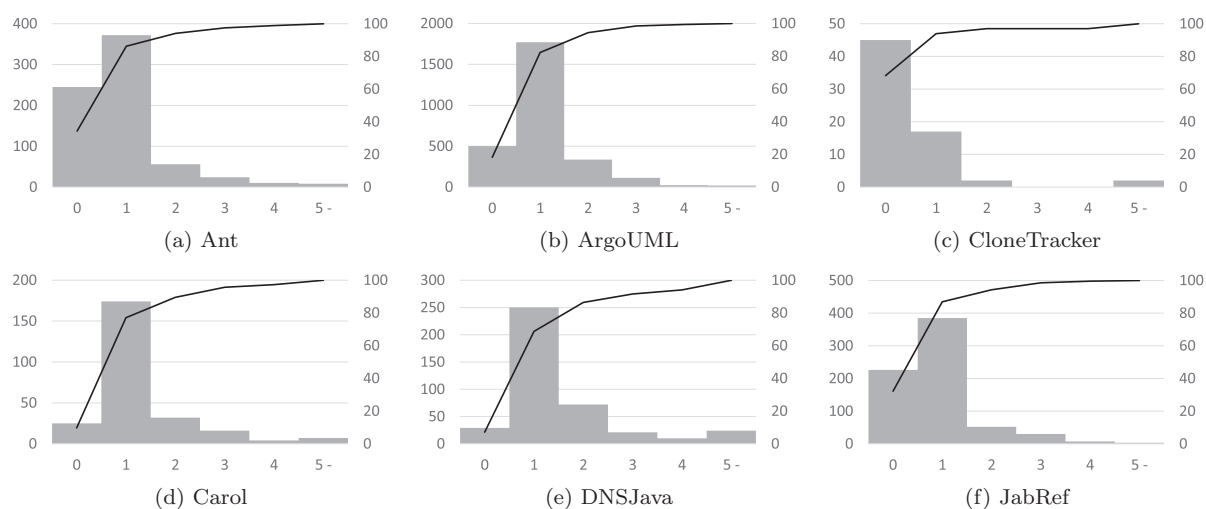


図 5 修正回数

Fig. 5 The number of modifications.

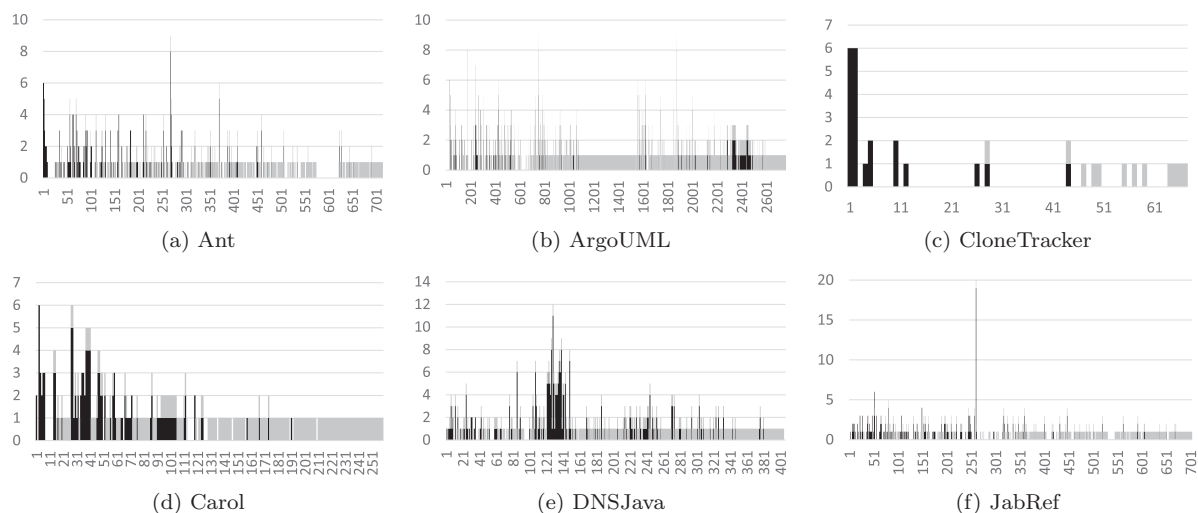


図 6 各系譜の修正回数

Fig. 6 The number of modifications on individual genealogies.

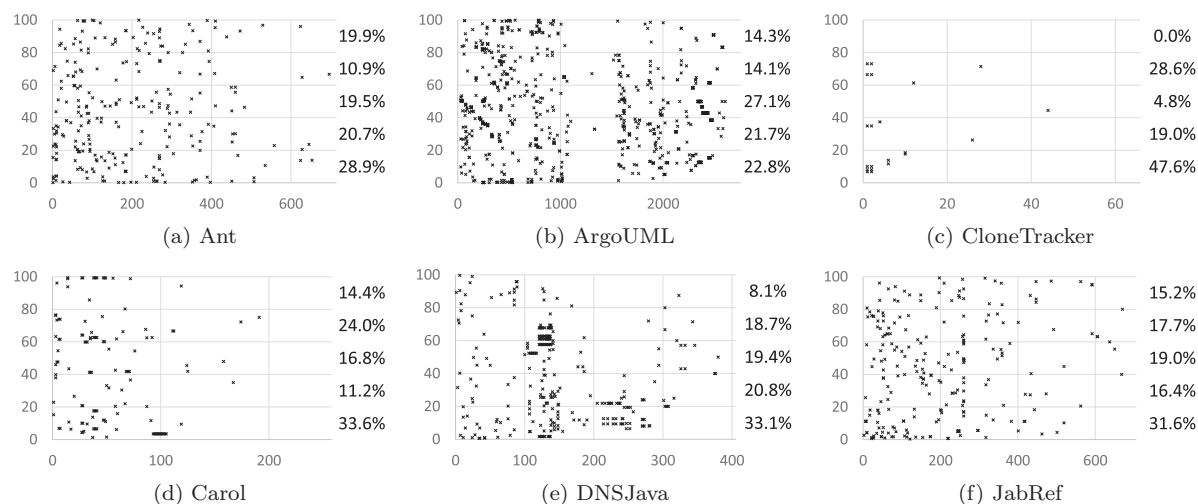


図 7 修正の時期

Fig. 7 Timing of modifications.

に並べたものであり、左側ほど生存期間の長い系譜となる。また、棒グラフはその系譜に加えられた修正の回数を表しており、グレーは消滅に関わる修正を、黒はそれ以外の修正の回数をそれぞれ表している。なおこのグラフは積み上げ棒グラフとなっており、黒の棒グラフの上にグレーの棒グラフが積み上げられている。

グラフより、消滅に関わる修正のみが加えられた系譜が多数を占めていることが分かる。また、生存期間と修正回数の間に相関があるか否かをスピアマンの順位相関係数を用いて判定した(表 2)。その結果、Ant, CloneTracker, DNSJava については有意水準 5%で相関がみられなかったが、ArgoUML, Carol, JabRef については相関がみられた。ただし、相関がみられた 3つのソフトウェアについても、正負にはばらつきがあり、かつ強い相関ではなかった。

この結果から、RQ3 には **No** と回答することができる。

表 2 修正回数と生存期間に対するスピアマンの順位相関係数

Table 2 Spearman's rank correction coefficients.

ソフトウェア	ρ	p -value
Ant	-0.004515	0.9117
ArgoUML	-0.4130	2.2×10^{-16}
CloneTracker	-0.05487	0.6617
Carol	0.3320	4.736×10^{-8}
DNSJava	0.08566	0.08473
JabRef	-0.1393	0.0002124

6.4 RQ4: コードクローンは生存期間の前半に修正されやすい傾向にあるか

図 7 に各ソフトウェアの各系譜に加えられた修正の時期を示す。グラフの x 軸は系譜を表しており、図 6 と同様に生存期間の降順に並べられている。グラフ中の各点は 1 つの修正を表しており、y 軸はそれぞれの修正が加えられた時期を表している。y 軸は各系譜の生存期間が 100 になるように正規化した値となっており、0 に近い位置に点が

表 3 修正時期に関する χ^2 検定Table 3 χ^2 -test for timing of modifications.

ソフトウェア	前半	後半	χ^2	p-value
Ant	155	101	5.3407	0.02083
ArgoUML	540	305	32.7572	1.044×10^{-8}
CloneTracker	15	6	1.2228	0.2688
Carol	71	54	0.9042	0.3417
DNSJava	169	115	4.804	0.02839
JabRef	133	98	2.3707	0.1236

存在する場合は、生成後すぐに修正されたことを示している。反対に y 軸の値が大きい位置に点が打たれている場合、生存期間の終盤に修正が加えられたことを示唆している。またグラフ右側の数値は、それぞれのソフトウェアについて、その時期に加えられた修正が全体に占める割合（百分率）を表しており、たとえば一番下の数値（Ant の場合、28.9%）は生成直後から生存期間の 20%が経過するまでの間に加えられた修正の割合を示している。なお、ここでは生存期間の途中で加えられた修正の時期を分析することを目的としているため、消滅に関わる修正は考慮していない。

次に、生存期間の前半と後半に加えらる修正回数に差があるか否かを χ^2 検定を用いて判定した。その結果を表 3 に示す。表 3 の“前半”の列は生存期間の 50%が経過するまでの間に、“後半”の列は生存期間の 50%が経過した後に加えられた修正の回数をそれぞれ表している。いずれのソフトウェアについても前半の方が後半よりも修正回数が多いという結果となった。また、Ant, ArgoUML, および DNSJava については有意水準 5%で有意な差となった。

以上より、RQ4 には Yes と回答することができる。ただし、生存期間の後半に加えらる修正の回数は生存期間の前半と比較して少ない傾向にあるものの、後半においても相当数の修正が加えられており、コードクローンが生存期間の後半において安定しているとはいえない。したがって、生成後一定期間が経過したコードクローンへの対処が効果的でないとは必ずしもいえない。しかしながら、生存期間の前半の方が修正される回数がやや多いため、コードクローンへの対処は早期であるほど高い効果を発揮する可能性が高い。

7. 考察

7.1 生存期間が長く、修正回数が多いコードクローン

1 章で述べたように、生存期間が長く、かつ修正される回数の多いコードクローンに注力することが、効率的なコードクローンへの対処を実現するうえで重要である。そこで、そのようなコードクローンがどの程度存在するのかを分析した。その結果を表 4 に示す。表中の“生存期間”の列は生存期間が長い系譜の数を、“修正回数”の列は修正回数が 2 回以上の系譜の数をそれぞれ表しており、また“生存期間&修正回数”の列は生存期間が長く、かつ 2 回以

表 4 全期間の半分以上生存し、かつ複数回修正されている系譜の数
Table 4 The number of long-lived genealogies which are modified multiple times.

ソフトウェア	生存期間	修正回数	生存期間&修正回数 (全系譜に占める割合)
Ant	223	98	52 (7.27%)
ArgoUML	336	490	48 (1.74%)
CloneTracker	18	4	4 (6.06%)
Carol	54	59	30 (11.63%)
DNSJava	9	127	4 (0.99%)
JabRef	102	92	23 (3.27%)
計	742	870	161 (3.28%)

上修正された系譜の数を表している。なおここでは、‘生存期間が長い’か否かの判別基準として、Kim らの研究 [10] で用いられている基準と同様に、‘分析期間の半分以上生存しているかどうか’を用いた。表 4 より、生存期間が長く、かつ複数回修正されるようなコードクローンは全体の約 3.3%存在していた。

また、各系譜の生存日数と修正回数の関係を表した散布図を図 8 に示す。図の横軸は系譜の生存日数を、縦軸は修正回数をそれぞれ表している。表 4 では分析期間の半分以上生存しているものを生存期間の長い系譜としている。しかし、図 8 を見ると、その基準には満たないものの 1 年を超えるような生存日数を有し、かつ複数回修正されている系譜が存在していることが分かる。その一方、生存日数が数日から数十日程度であり、かつ複数回修正されている系譜が多数存在することも確認できる。

このような結果から、生存期間が長く、かつ複数回修正されるという両方の特性を持つコードクローンは、いずれか一方の特性を持つコードクローンと比較して少ないことが分かる。したがって、すべてのコードクローンに対して平等に対処を行うことは効率的とはいえず、双方の特性を持つ対処すべきコードクローンを慎重に選択する必要があるといえる。

図 9 に生存期間が長く、修正回数の多いコードクローンの例を示す。この例は CloneTracker に存在するコードクローンであり、リビジョン 3 で生成されて以降、最終リビジョンまで生存していた。このコードクローンは計 6 回修正されており、修正が加えられたリビジョンは 17, 19, 24, 121, 234, および 244 となっている。このコードクローンに対する修正漏れは存在していなかったものの、このように生存期間が長く修正回数の多いコードクローンは修正漏れの要因となる危険性が高い。早期にこのようなコードクローンへの対処を行うことで、修正漏れを効率的に防ぐことができると考えられる。

7.2 結果の妥当性

本節では、本研究の結果の妥当性を脅かすおそれのある

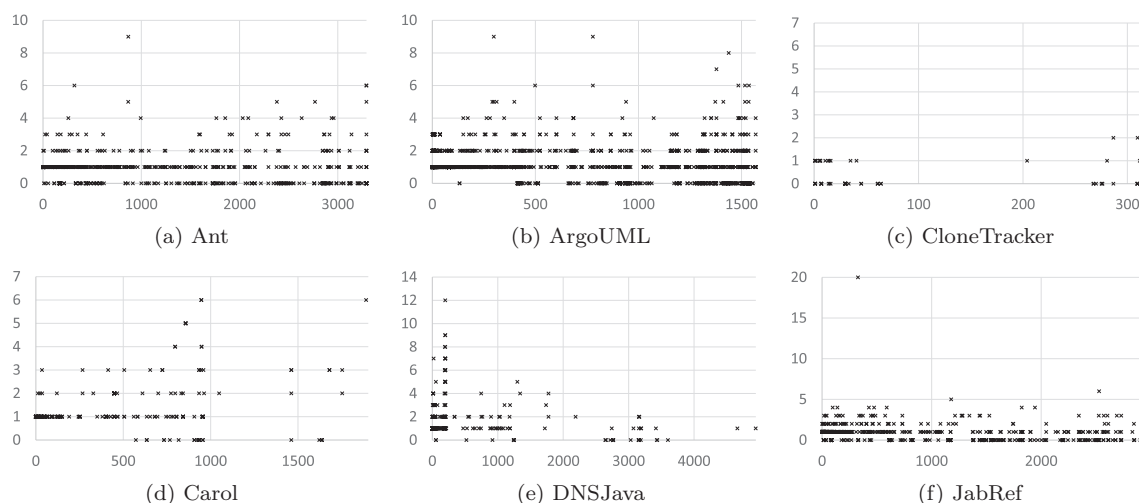


図 8 各系譜の生存日数と修正回数

Fig. 8 Scatter plots between the number of modifications and alive days.

```

9: public class FinallyBlockInfo extends BlockInfo {
15:     public FinallyBlockInfo(String core, List<String> types) {
16:         super(BlockType.FINALLY, core);
17:         this.types = new LinkedList<String>();
18:         this.types.addAll(types);
19:         StringBuilder builder = new StringBuilder();
20:         for (String type : types) {
21:             builder.append(type + " ");
22:         }
23:         this.concatenatedTypes = builder.toString();
24:         this.crdElement = new BlockCRD(btype);
25:     }
40: }

```

図 9 生存期間が長く、かつ修正回数が多いコードクローンの例

Fig. 9 An example of long-lived clone which is modified multiple times.

要因について述べる。

対象ソフトウェア

今回の調査では、Java で記述されたオープンソースソフトウェアのみを対象としている。そのため、他の言語で記述されたソフトウェアや商用ソフトウェアを対象と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が導かれる可能性がある。また、コードクローンの含有率も実験の結果に影響を与える可能性が考えられるため、コードクローン含有率が極端に高い、もしくは極端に低いソフトウェアに対して実験を行った場合、本研究における実験結果と異なる結果が得られる可能性がある。

コードクローンの検出方法

本研究では、調査に要する時間的コストを低減させるために、ブロック単位のコードクローン検出手法を用いている。そのため、字句単位の検出手法などの他の検出手法を用いて調査を行った場合、本研究と異なる結果が得られる可能性がある。また本研究で用いたコードクローン検出手法はサブブロックや変数名などの正規化を行っている。正規化の方法を変更することで、調査結果に影響が及ぶ可能性がある。さらに、自動生成されたコードはコードクローンとなりやすいことが指摘されているが [4]、本研究では

自動生成されたコードに対する特別な前処理を行っていない。したがって、自動生成によって作りこまれたコードクローンを分析対象から除くことで、異なる結果が得られる可能性がある。

コードクローンの追跡

本研究ではコードクローンの追跡に CRD を用いた。CRD はソースコードに対する変更に近いコードクローン追跡手法であるが、CRD を用いた場合でも、コード片が別のファイルに移動した場合など、位置情報が変わるような変更が行われた場合、コードクローンを適切に追跡できない可能性がある。

対処を要するコードクローンの特性

本研究では、対処を要するコードクローンの特性として、生存期間が長いこと、ならびに複数回修正されることをあげ、調査の対象とした。しかし、コードクローンを構成するコード片の長さや、コードクローンを構成するコード片の数、コードクローンに対する修正漏れの多寡など、今回着目した特性以外のものも対処を要するコードクローンを考えるうえで重要な特性となりうる。したがって、本研究で着目した特性以外のものを分析対象とすると、異なる結果が導かれる可能性がある。ただし、“対処すべきコードクローンとは何か”という点については統一された見解は存在しておらず、議論の余地が残されている。

8. あとがき

本研究では、コードクローン追跡手法 CRD を用いてコードクローンの進化の特性を分析した。分析を通じ、これまでの調査で報告されている、‘コードクローンの多くは生存期間が短い’、および‘複数回修正されるコードクローンは少ない’という知見が成り立つことを改めて確認するとともに、コードクローンの生存期間の長さや修正回数に相関がみられないこと、およびコードクローンは生存期間の前

半において修正されやすい傾向にあることを明らかにした。また、生存期間が長く、かつ修正される回数の多いコードクローンが全体の約 3.3% 程度であることを示した。

今後の課題として、異なる言語で記述されたソフトウェアに対する分析を行うことや、異なるコードクローン検出手法を用いて分析を行うことがあげられる。さらに、生存期間が長く、かつ修正される回数の多いコードクローンを予測することが今後の重要な課題である。本研究を通じて注力すべきコードクローンの割合が高くないことが明らかになった。すなわち、効率的にソフトウェア保守を行うためには対処するコードクローンを慎重に選択する必要があるといえる。しかし、本研究は対処すべきコードクローンの予測、あるいは特定には至っていない。現存するコードクローンが今後長く生存するのか、また頻繁に修正されるのかを予測することで、対処すべきコードクローンの選択を支援することができる。また、本研究の調査でコードクローンは生存期間の前半に修正されやすい傾向にあることが明らかとなったため、現在あるコードクローンが今後どの程度の期間生存するのかを予測することができれば、そのコードクローンが生存期間の前半にいいのか、あるいは後半にいいのかを知ることが可能となり、コードクローンへの対処を検討するうえでの指標となると考えられる。

謝辞 本研究は、日本学術振興会科学研究費補助金基盤研究 (S) (課題番号: 25220003)、挑戦的萌芽研究 (課題番号: 24650011)、特別研究員奨励費 (課題番号: 25・1382)、文部科学省科学研究費補助金若手研究 (A) (課題番号: 24680002)、および独立行政法人情報処理推進機構技術本部ソフトウェア高信頼化センターが実施した「2012 年度ソフトウェア工学分野の先導的研究支援事業」の助成ならびに支援を受けて行われた。

参考文献

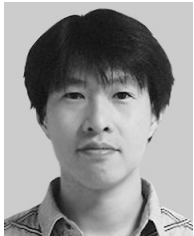
- [1] 井上克郎, 神谷年洋, 楠本真二: コードクローン検出法, コンピュータソフトウェア, Vol.18, No.5, pp.47–54 (2001).
- [2] Koschke, R.: Frontiers on Software Clone Management, *Proc. Frontiers of Software Maintenance in the 24th International Conference on Software Maintenance*, pp.119–128 (2008).
- [3] 肥後芳樹, 楠本真二, 井上克郎: コードクローン検出とその関連技術, 電子情報通信学会論文誌, Vol.91-D, No.6, pp.1465–1481 (2008).
- [4] 神谷年洋, 肥後芳樹, 吉田則裕: コードクローン検出技術の展開, コンピュータソフトウェア, Vol.28, No.3, pp.28–42 (2011).
- [5] 肥後芳樹, 吉田則裕: コードクローンを対象としたリファクタリング, コンピュータソフトウェア, Vol.28, No.4, pp.42–56 (2011).
- [6] Zibran, M.F. and Roy, C.K.: The Road of Software Clone Management: A Survey, Technical Report, Department of Computer Science, University of Saskatchewan, Canada (2012).
- [7] 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).
- [8] Li, Z., Lu, S., Myagmar, S. and Zhou, Y.: CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code, *IEEE Trans. Softw. Eng.*, Vol.32, No.3, pp.176–192 (2006).
- [9] Jiang, L., Mishnerghi, G., Su, Z. and Glondou, S.: DECKARD: Scalable and Accurate Tree-based Detection of Code Clones, *Proc. 29th International Conference on Software Engineering* (2007).
- [10] Kim, M., Sazawal, V., Notokin, D. and Murphy, G.C.: An Empirical Study of Code Clone Genealogies, *Proc. 13th International Symposium on Foundations of Software Engineering*, pp.187–196 (2005).
- [11] Göde, N. and Koschke, R.: Frequency and Risks of Changes to Clones, *Proc. 33rd International Conference on Software Engineering*, pp.311–320 (2011).
- [12] Rahman, F., Bird, C. and Devanbu, P.: Clones: What is that Smell?, *Proc. 7th Working Conference on Mining Software Repositories*, pp.72–81 (2010).
- [13] Monden, A., Nakae, D., Kamiya, T., Sato, S. and Matsumoto, K.: Software Quality Analysis by Code Clones in Industrial Legacy Software, *Proc. 8th International Symposium on Software Metrics*, pp.87–94 (2002).
- [14] Hotta, K., Sasaki, Y., Sano, Y., Higo, Y. and Kusumoto, S.: An Empirical Study on the Impact of Duplicate Code, *Advances in Software Engineering*, Vol.2012 (2012).
- [15] Harder, J. and Göde, N.: Cloned Code: Stable Code, *Journal of Software: Evolution and Process* (2012).
- [16] Krinke, J.: Is Cloned Code More Stable than Non-Cloned Code?, *Proc. 8th International Working Conference on Source Code Analysis and Manipulation*, pp.57–66 (2008).
- [17] Duala-Ekoko, E. and Robillard, M.P.: Clone Region Descriptors: Representing and Tracking Duplication in Source Code, *ACM Trans. Software Engineering and Methodology*, Vol.20, No.1, pp.3:1–3:31 (2010).
- [18] Thummalapenta, S., Cerulo, L., Aversano, L. and Penta, M.D.: An Empirical Study on the Maintenance of Source Code Clones, *Empirical Software Engineering*, Vol.15, No.1, pp.1–34 (2010).
- [19] Göde, N.: Evolution of Type-1 Clones, *Proc. 9th International Working Conference on Source Code Analysis and Manipulation*, pp.77–86 (2009).
- [20] Saha, R.K., Asaduzzaman, M., Zibran, M.F., Roy, C.K. and Schneider, K.A.: Evaluating Code Clone Genealogies at Release Level: An Empirical Study, *Proc. 10th Working Conference on Source Code Analysis and Manipulation*, pp.87–96 (2010).
- [21] Krinke, J.: A Study of Consistent and Inconsistent Changes to Code Clones, *Proc. 14th Working Conference on Reverse Engineering*, pp.170–178 (2007).
- [22] Göde, N. and Harder, J.: Oops! ... I Changed It Again, *Proc. 5th International Workshop on Software Clones*, pp.14–20 (2011).
- [23] Kapser, C.J. and Godfrey, M.W.: “Cloning considered harmful” Considered Harmful: Patterns of Cloning in Software, *Empirical Software Engineering*, Vol.13, No.6, pp.645–692 (2008).
- [24] Juergens, E., Deissenboeck, F., Hummel, B. and Wagner, S.: Do Code Clones Matter?, *Proc. 31st International Conference on Software Engineering*, pp.485–495 (2009).

- [25] 吉田則裕, 崔 恩瀾, 肥後芳樹, 楠本真二, 井上克郎: 産学連携による携帯端末用ソフトウェアに対する品質改善の取り組み, ソフトウェアエンジニアリングシンポジウム, pp.9:1-9:6 (2012).



堀田 圭佑 (学生会員)

平成 22 年大阪大学基礎工学部情報科学学科卒業. 平成 24 年同大学大学院情報科学研究科博士前期課程修了. 現在, 同研究科博士後期課程在学中. 日本学術振興会特別研究員 DC. コードクローン分析に関する研究に従事. 電子情報通信学会, IEEE 各会員.



肥後 芳樹 (正会員)

平成 14 年大阪大学基礎工学部情報科学学科中退. 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. 電子情報通信学会, IEEE 各会員.



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. 電子情報通信学会, IEEE, JFPUG 各会員.