修士学位論文

題目

# クローンセットに対する主要編集者の分析法の提案と調査

指導教員
井上 克郎 教授

報告者
辻　健二

平成 30 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 29 年度 修士学位論文

クローンセットに対する主要編集者の分析法の提案と調査

辻　健二

## 内容梗概

コードクローンとは，互いに一致もしくは類似したコード片のことである．一般的に，あるコードクローンが編集されると，同じクローンセット（コードクローンの関係となっている全てのコード片の集合）に属している他のコードクローンも同様に編集される必要がある．しかし，異なる開発者がクローンセット内の各コードクローンの編集を担当している場合は，コードクローンに対する一貫した編集が困難である．この問題を防ぐために，開発者間でのコードクローンの編集情報の共有を支援するツールが開発されている．このツールは，開発者のうちの誰かがコードクローンの箇所を編集した際に，その変更情報を開発者にメールで通知，もしくは Web ベースでの可視化を行うものである．しかし，このツールは複数の開発者でコードクローンを編集する体制を前提としたものであり，実際のソフトウェア開発現場でのコードクローン編集管理が複数人によって行われているとは限らない．コードクローンの編集を管理している主要な開発者が存在しているソフトウェア開発においては，その開発者管理するコードクローンに対する一貫していない編集を把握することができるため，編集情報の共有は必要ないと考えられる．従って，ツールの有用性を示すためには，コードクローンを編集している開発者の人数について調査する必要がある．

既存研究では，コードクローンの編集を行う開発者が単独であるか複数であるかを調査し，コードクローンが主に同一の開発者によって編集されていることが分かった．しかし，この研究は編集割合の最も多い開発者の情報のみ焦点を当てているためにそれ以外の開発者の存在が考慮されていない．

また，ある成果物に対して最も編集を行っている者の編集割合を示す値として，Ownership というメトリックが提案されている．このメトリックを用いた研究は存在しているが，クローンセットに対して適用した事例は無い．

この問題点を解決するために，本研究では，Ownership メトリックをクローンセットに対して適用した新たなメトリックを提案する．このメトリックを用いることで，クローンセットに対する主要な開発者が存在するかどうかに加え，その担当割合を把握することができる．この値が低いと，そのクローンセットには主要な開発者がおらず，複数人によって編集されているということがわかる．

本研究では，3 つのオープンソースソフトウェアに対して，そのクローンセットに対する編集者とその編集人数についての調査を行った．調査にあたり，クローンセットに対して Ownership を適用した新たなメトリックを提案し，それを用いている．調査の結果，オープンソースソフトウェアの中には，主要な編集者がおらず，複数人でクローンセットに対して編集を行って開発を進めているプロジェクトが存在していることがわかった．また，複数人で編集を行っていると判定できたクローンセットの中に，今後欠陥を含む危険性の高い編集が行われているものを確認することで，クローンセットの編集情報を共有することが有用である場面が存在することを確認した．

## 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)
Ownership メトリック

目 次

## 1　まえがき

ソフトウェアの保守工程における大きな問題の一つとして，コードクローンが指摘されている．コードクローンとは，ソースコード中に存在する互いに一致または類似した部分を持つコード片のことであり，主に既存コードのコピーペーストによって生成される [9]．また，互いにコードクローンとなっているコード片の集合のことをクローンセットという．

コードクローンとなっているコード片が編集された場合，同じクローンセットに属している他のコードクローンに対して，一貫した編集が行われる必要がある．しかし，開発者が編集しようとしているコード片はコードクローンであるということに気付かずにそのコード片のみの修正に留まってしまい，他のコード片を無視してしまう可能性は非常に高い [12, 14]．この問題を防ぐためには，システムにかかわる全ての開発者の間で，コードクローンの編集情報が共有される必要がある．

山中らは開発者間でのコードクローンの編集情報の共有を支援するため，Clone Notifier を開発した [21]．Clone Notifier は，対象システム内のコードクローンが編集された際，登録している開発者全員に対してクローン編集の通知を行うツールである．実際にある企業での，複数人によるソフトウェア開発に対してこのツールを適用した結果，プロジェクトマネジャーが気付かなかったコードクローンの編集を通知することができた．Clone Notifier はコードクローンが複数の開発者によって編集されている開発体制では有用であるといえる．しかし，一般的なソフトウェア開発において，コードクローンが単独または複数の開発者によって編集されているか，に関しての調査は十分には行われていない．コードクローンの編集情報共有の有用性を確かめるためには，コードクローンを編集する開発者の人数に関する調査が必要である．

Harder はコードクローンの編集を行う開発者が単独か複数かを調査しており，その結果コードクローンは主に同一の開発者によって編集されていることが分かった [8]．しかし，この研究は編集割合の最も多い開発者の情報のみ焦点を当てているためにそれ以外の開発者の存在が考慮されていない．

これらの問題点を解決するために，本研究は，コードクローンに対する編集を行う開発者の数および編集割合を，Ownership[5] というメトリックを用いて調査した．Ownership とは，ある成果物に対して行われた全ての編集のうち，最も多く編集した開発者が行った編集の割合を示すメトリックであり，開発者がその成果物を担当する強さの度合いを示すものとなっている．本研究で Ownership メトリックを用いることで，コードクローンの編集を担当している開発者が単独か複数かを判断できるだけでなく，その編集割合についても理解することが期待できる．Ownership メトリックを用いて，いくつかのオープンソースソフトウェアを対象に各クローンセット内のコードクローンの編集回数や最も多く編集を行った開発者

の編集割合を計測した結果，コードクローンに対して主要な開発者を持たず，複数人によって編集されるコードクローンを多数含んでいるプロジェクトが存在していることが分かった．また，低い Ownership を示すコードクローンの中には，今後欠陥を含む危険性の高い編集が行われているものがあることが確認できた．

以降，2 章では，本研究の背景について述べる．3 章では，Ownership をクローンセットに適用することによる編集回数の分析手法について述べる．4 章では，実際に行った調査内容についての説明を行う．5 章では，調査結果を示し，結果についての考察を行う．最後に 6 章で本研究のまとめと今後の課題について記述する．

## 2　背景

本章では，本研究の背景として，コードクローン，バージョン管理システム，コードクローンの編集管理，コードクローンの編集者分析について述べる．

### 2.1　コードクローン

コードクローンとは，ソースコード中に存在する，互いに一致または類似する部分を持つコード片のことを指す．また，互いにコードクローンとなるコード片の対はクローンペアとよび，互いにコードクローンとなっているコード片の集合をクローンセットとよぶ．コードクローンが存在するプログラムでは，あるコード片に欠陥が含まれていた場合，そのコード片に対して行った修正を，対応する全てのコードクローンに対して行うかどうか検討しなければならない．特に大規模なソフトウェアを対象とする保守作業においては，大量のソースコードの中からコードクローンを探し出し，その全てに対して修正の検討を行うことは，高い修正コストを必要とする．また，あるコード片を書いた編集者と別の編集者が修正を行おうとする場合，特にこの修正コストは高くなる．そのため，コードクローンはソースコードを編集する上で常に意識しておく必要があり，コードクローンの検出，管理は開発コスト削減のための重要な課題となっている．

#### 2.1.1　コードクローン検出手法

ソースコード中からコードクローンを検出するために，さまざまな類似度に基づく手法が提案されている [18, 16]．

Baker は，コードクローンを検出する手法を提案し，コードクローン検出ツール Dup を開発した [2]．Dup はトークンの類似度に基づいて，ユーザ定義名や予約語を除く一致箇所のコードクローンを検出する．Jiang らは抽象構文木の類似度に基づいてコードクローンを検出する DECKARD を開発した [11]．DECKARD は，入力されたソースコードを抽象構文木に変換し，その後の各部分木を特徴ベクトルに変換する．また，LSH(Locality-Sensitive Hashing) アルゴリズム [10] を用いて，特徴ベクトル間の類似度を求めることによって，コードクローンの検出を行うことができる．Kamiya らはソースコードのトークン列の類似度に基づいてコードクローンを検出する CCFinder を開発した [13]．CCFinder は，入力されたソースコードをトークン列に変換する．このとき，変数名や識別子名などのユーザ定義名を特殊文字に置き換えることによって，ユーザ定義名や予約語が異なるコード片でもコードクローンとして検出することができる．CCFinder は検出精度が高いことで知られており，さまざまな研究や会社で利用されている．しかし，大規模のソースコードが入力された場合，コードクローンを検出するために，膨大な時間がかかるという短所を持っている．

この問題を解決するために，Choi らは CCFinder の前処理や後処理を行うことでコードクローンの検出時間を減らす手法を提案した [6]．彼らの手法は，各入力ファイルの MD5 ハッシュ値 [17] を計算し，そのハッシュ値が一致したファイルの集合を完全一致ファイルセットとして検出する．また，特殊なハッシュ値のファイルや，各完全一致ファイルセットから選ばれた 1 つのファイル (代表ファイル) を CCFinder の入力とし，コードクローンの検出を行っている．その後，代表ファイルからクローンが検出された場合，そのファイルのクローン情報に基づいて，マッピングを行い同じ完全一致ファイルセットに属するファイルクローンの位置を特定している．

本研究では Choi らの手法を拡張し，ファイル単位でのコードクローンを検出した．つまり，各入力ファイル中の全トークンを，一つのトークン列に連結した．その際，変数名や識別子名などのユーザ定義名を特殊文字に置き換えることにより，ユーザ定義名によるソースコードの差分がクローン検出の妨げにならないようにしている．その後は，Choi らの手法と同様，各変換された入力ファイルの MD5 ハッシュ値の計算を行い，各ファイルのハッシュ値に基づきファイルクローンを検出した．

### 2.1.2 コードクローンに対する編集の危険性

2.1 節で説明したように，コードクローンに対する編集は，一貫性が保たれているのが望ましい．しかし，意図的に一貫性を解消して開発を進めるといったケースも存在するため，一貫していない編集が必ず危険性を伴うとは一概には言えない．そこで，一貫していない編集の中でも，よりバグを含む危険性の高い編集について焦点を当てる必要がある．Late Propagation[3, 4] と呼ばれる，コードクローンに対する一貫していない編集の組み合わせが，より危険な編集の例として挙げられている．

Late Propagation とは，2 つのコードクローンの組であるクローンペアに対して一貫性を破壊する編集が行われたあとに，一貫性を回復する編集が行われる一連の現象を指す．図 1 は，ファイル中に存在するコード片 A,B のクローンペアに対して発生した Late Propagation を図示したものである．まずフェーズ 1 での編集で，クローンペアのうちコード片 A だけが編集されたことにより一貫性が失われ，コードクローンでは無くなっている．フェーズ 2 は，一貫性を回復することのない編集が繰り返される段階である．このフェーズでは全く編集が行われない場合もある．そして，最後にフェーズ 3 でコード片 B が編集されることによって一貫性が回復し，2 つのコードクローンが再度クローンペアに戻っている．Late Propagation を引き起こす一貫性を破壊する編集は，後に一貫性を回復する編集が行われるということから，本来同時に修正すべきものであったと考えられる．そのため，Late Propagation を経験しているクローンペアは，そうではない編集を受けているクローンペアと比較して，よりバグが混入しやすいことが知られている [1, 20]．Late Propagation は，フェーズごとに行う

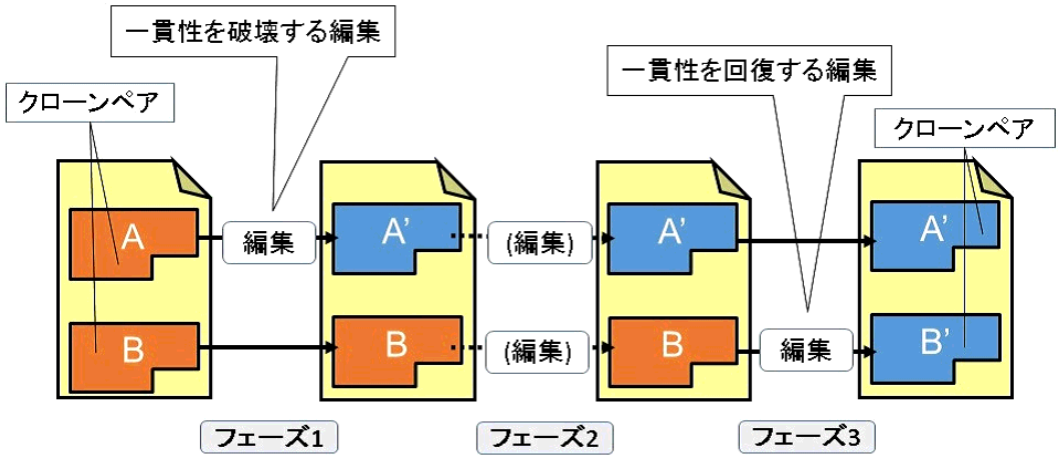


図 1: Late Propagation 発生の流れ

編集先のクローン片によって，8 つの種類に分類される．その分類を表 1 に示す．図 1 の例は，フェーズ 1 で A が編集され，フェーズ 2 では編集が行われず，フェーズ 3 で B が編集されているため，この一連の編集は LP1 に分類される. 既存研究では，これらの編集タイプのうち，LP7,LP8 が起きている箇所は，特にバグを含む危険性が高いと言われている [20].

表 1: Late Propagation 分類表

| タイプ | フェーズ 1 で<br>編集されたコード片 | フェーズ 2 で<br>編集されたコード片 | フェーズ 3 で<br>編集されたコード片 |
|---|---|---|---|
| LP1 | A | -,A | B |
| LP2 | A | B,AB | B |
| LP3 | A | -,A | AB |
| LP4 | A | B,AB | A |
| LP5 | A | -,A,B,AB | AB |
| LP6 | AB | -,A,B,AB | A |
| LP7 | AB | -,A,AB | AB |
| LP8 | A | -,A | A |

### 2.1.3 コードクローンの編集管理

2.1 節で説明したように，あるコード片にバグが存在することが判明した場合，そのコード片に対するコードクローンにも同様のバグが存在する可能性が考えられる．コードクローンが編集された際，そのコードクローンの編集に関わる全ての開発者がその編集内容について把握することで修正漏れやバグの混入などを防ぐことができると考えられる．

山中らは，コードクローン編集の情報を開発者間で共有することで開発を支援するツール Clone Notifier[21] を開発した．Clone Notifier は，バージョン間でコードクローンの差分情報を検出し，その情報をツールに登録している開発者に対してメールを送ることでコードクローンに対する編集を通知するツールである．Clone Notifier は企業でのソフトウェア開発への適用実験が行われている．その結果，プロジェクトマネジャーや開発者らが意図していなかったコードクローン編集を検出し，それを通知することができた．Clone Notifier は複数人でコードクローンを編集する開発体制を前提としたものである．しかし，オープンソースソフトウェアなど，開発現場でのコードクローン編集が複数人によって行われているとは限らない．従って，コードクローンを編集している開発者の人数の調査が必要である．

## 2.2 バージョン管理システム

バージョン管理システムとは，ファイルの編集履歴を記録，管理することができるシステムである．代表的なものとして,CVS(Concurrent Version System)，Apache Subversion，Git などが挙げられる．バージョン管理システムを用いることで，ファイルを編集した開発者やその編集日時，編集内容等を記録することができ，過去の記録に基づいてファイルを過去の記録時点での状態に復元したり，編集内容の差分を表示，取得することが可能となっている．バージョン管理システムにおいて，ファイルの変更情報を保存しているデータベースをリポジトリと呼び，その変更単位のことをコミットと呼び，リポジトリに対してファイルの変更情報を記録する操作をコミットするという．ソフトウェア開発においては，主にソースコードの変更情報を管理するのに用いられている．ソフトウェアに不具合が発生した際の原因箇所の特定や，特定の変更内容の削除，ファイル変更内容をほかプロジェクトへ再利用するなど，バージョン管理システムは開発の現場においても実際に活用されている．

バージョン管理システムは，リポジトリがファイルを管理する方式の違いから，3 つの分類がなされている．ローカルマシン上でファイル管理を行う個別型，クライアント・サーバー形式で管理する集中型，そしてリポジトリを複数持つことのできる分散型である．

ここでは，集中型，分散型それぞれの例として，Subversion，Git についての詳細な説明を行う．

- 集中型バージョン管理システム Subversion

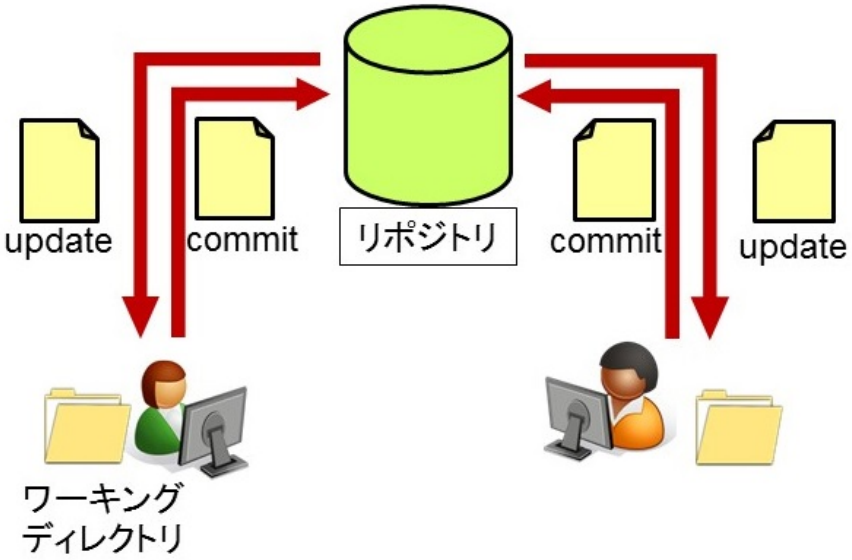


図 2: 集中型バージョン管理システムの概要図

Subversion は，クライアント・サーバ形式でファイル管理を行う集中型のバージョン管理システムである．ファイルの変更履歴はすべて単一のリモートリポジトリで管理されており，リポジトリを利用する編集者のローカル環境には，ある時点での変更履歴のみが保持されているといった状態になる．図 2 に，動作概要図を示す．

リポジトリに対するコミット操作や，リモートリポジトリの変更履歴の取得，過去の変更履歴の表示等は，リモートリポジトリとのネットワーク接続が必要という制約がある．変更履歴の取得には，svn log というコマンドを実行することで，自分の作業コピーの現在の作業ディレクトリ自身と，その内部全てのファイルとディレクトリに関するログメッセージを表示することが出来る．変更単位はリビジョンと呼ばれている．各リビジョンにはコミットした順番に整数値の ID が割り振られ，svn log コマンドのオプションとして ID を指定することで，特定リビジョンの変更履歴を取得することが出来る．

- 分散型バージョン管理システム Git

  Git は，分散型のリポジトリを持つバージョン管理システムである．分散型リポジトリは，リポジトリをサーバ上だけではなく，クライアントであるローカル上にも作成する形体を取る．手元にあるリポジトリにアクセスできるため，場所や時間，ネットワーク環境の有無に関わらず開発を進めることができ，コミット処理も高速に行うことができる．図 3 に，分散型リポジトリの動作概要図を示す．

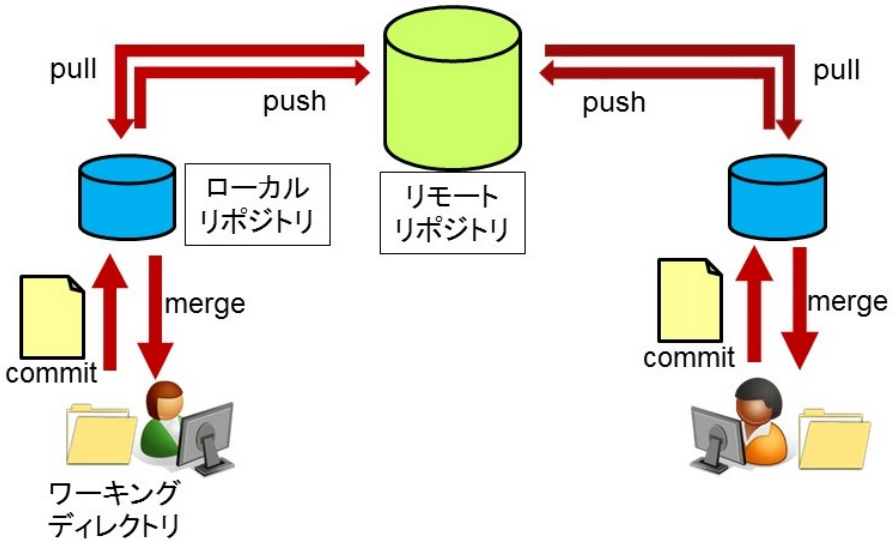


図 3: 分散型バージョン管理システムの概要図

本研究では，ファイルをコミットした開発者を，そのファイルの編集者としている．バージョン管理システムのうち，多数のオープンソースソフトウェア開発で用いられている Git を対象に，ファイルの編集者とその回数の調査を行った．

### 2.3 コードクローン編集者の分析

コードクローンの編集者とは，コードクローンを生成，もしくは編集した開発者のことを指す．Harder はコードクローン編集者を分析し，同じクローンセット内のコードクローンの編集を行った開発者が単独かまたは複数かを調査した [8]．彼の研究では，トークン単位で編集を担当している開発者の特定を行い，その情報を用いてコードクローンの編集者を決定している．この研究の結果，対象とした全てのソフトウェアにおいて，単独で編集されているクローンセットの数が複数人のものよりも多いことが明らかになった．しかし，この研究では，クローンセット全体に対して行われた編集の割合については考慮されておらず，クローンセットの編集に対する一貫性についての考察がなされていない．

### 2.4 Ownership メトリック

ある成果物に対して，その責任の度合を示した Ownership というメトリックが提案されている [5]．Ownership メトリックとは，ある成果物を編集した回数が最も多い編集者の編集割合を示したものである．成果物に対して行われた全編集回数の総計を $CO_{total}$，そのうち最も回数の多い開発者の編集回数を $CO_{max}$ とすると，Ownership の値は，式 1 で求める

ことができる．

$$Ownership = \frac{CO_{max}}{CO_{total}} \tag{1}$$

編集者の人数を数えるだけではなく，Ownership メトリックを用いる利点として，以下のメリットが挙げられる．

- Ownership メトリックの値を見て，関わっている開発者が単独か複数かを直感的に判別可能である．値が 1 であれば単独の開発者による編集であり，そうでなければ複数の開発者が編集に関わっていることが容易に理解できる．

- 複数人での編集であった場合，Ownership メトリックの値はその分担の度合いを反映することが可能である．例えば，Ownership が 0.25 であった場合，複数の開発者が編集に関わっていること以外にも，そのコンポーネントに対して，全ての開発者の編集回数が 1/4 以下であることが把握できる．

本研究では，コードクローンに対する編集者の数を調査するために，Ownership メトリックをコードクローンに対して適用した新たなメトリックを定義して，それを用いた調査を行っている．Ownership メトリックを適用することで編集割合を示し，コードクローンの編集を担当している開発者が単独か複数かを判断できるだけでなく，その編集割合についても理解することができる．

## 3　クローンセットに対する編集回数の分析手法

本章では，Ownership メトリックを用いることによるクローンセットに対する編集回数を分析する手法について述べる．

### 3.1　*CSF-Ownership* および *CST-Ownership* の定義

本研究では，Ownership メトリックをクローンセットに適用した新たなメトリックである *CSF-Ownership* を定義する．あるソフトウェアの，リポジトリ中に含まれるファイルクローンセット中の全てのファイルに対して，編集が行われているコミット回数の総計を $CSF_{total}$ とする．この総計のうち最多の編集者による全編集回数を $CSF_{max}$ とする．このときクローンセットの *CSF-Ownership* の値を式 2 によって定義する．

$$CSF\text{-}Ownership = \begin{cases} \frac{CSF_{max}}{CSF_{total}} & (CSF_{total} > 0) \\ 0 & (CSF_{total} = 0) \end{cases} \tag{2}$$

この値は，編集者がクローンセットに含まれるソースコードに対して編集を行った割合を示している．クローンセットに対する編集の回数について扱うため，編集回数が 0 となるクローンセットも存在する可能性がある．そのようなクローンセットは *CSF-Ownership* が 0 であると定義している．この式を元に，クローンセット毎に *CSF-Ownership* を算出する．計算結果が 0.5 を上回る場合，つまり，クローンセットに対して行われた総編集回数の半数よりも多く編集した編集者が存在している場合に，そのクローンセットには主要な開発者が存在するものとみなす．

ファイル単位での編集回数を数える際，例えば同じ編集量の作業に対して，コミット回数が極端に多い，もしくは少ない編集者が存在した場合，そのファイルの *CSF-Ownership* の値が大きく変動してしまう恐れがある．この問題を解決するには，粒度の細かい編集の回数を扱うことで，より成果物に対する責任度合いを正確に表現する必要がある．そこで，クローンセットに対するファイル単位での編集回数ではなく，トークン単位での編集回数に着目して Ownership を適用したメトリックである *CST-Ownership* を定義する．あるファイルクローンセット中の全てのファイルクローンについて，トークン毎の編集回数の総計を $CST_{total}$ とする．この総計のうち，最多のトークンを編集した編集者による全編集回数を $CST_{max}$ とする．このときクローンセットの *CST-Ownership* の値を以下の式によって定義する．

$$CST\text{-}Ownership = \begin{cases} \frac{CST_{max}}{CST_{total}} & (CST_{total} > 0) \\ 0 & (CST_{total} = 0) \end{cases} \tag{3}$$

### 3.2 クローンセットに対する編集回数

本研究では，クローンセットに対するファイル単位およびトークン単位での編集回数を数える必要がある．本節では，クローンセットに対する編集回数についての定義を行う．

#### 3.2.1 ファイル単位での編集回数

ファイル単位での編集回数について説明する．ファイルに対する編集の種類として，生成，削除，リネーム，内容修正がある．本研究では，ファイルに対して行われたこれらの編集を 1 回として数える．これらの編集情報は，バージョン管理システムを用いることで取得することができる．例えば，Git のコマンドである git log コマンドを用いることで，各コミットで行われた一連の履歴を新しい順に表示することができる．また，オプションやファイル名を指定することで，ファイルが編集を受けたコミットのリストや，ファイルが受けた編集の種類についても取得することができる．あるファイルに対して全てのコミットで行われた編集回数の合計を，そのファイルのファイル単位での総編集回数とする．クローンセットとなっている全てのファイルの総編集回数の合計が，クローンセットに対するファイル単位での編集回数である．

#### 3.2.2 トークン単位での編集回数

トークン単位での編集回数について説明する．あるコミットでファイルに対して編集が行われた際に，その編集でいくつのトークンが編集されているのかを数える．そのためには，まずソースコードをトークン単位に分割し，そしてファイル編集毎のトークンの編集回数を計測する必要がある．以下でそれぞれの手順の詳細について述べる．

ソースコードをトークン分割するにあたり，本研究では，srcML[15] および cregit[7] というツールを用いている．srcML とは，ソースコードのトークンを識別子し，各トークンに対して構文要素としてタグ付けするという表現形式を取るマークアップ言語である．srcML は，C,C++，C#,Java 言語に対応しており，これらの言語で書かれたソースコードをトークンに分割することが出来る．cregit は，ソースコードのトークンに対して，そのトークンに対して，最も多く編集している開発者を求めることができるツールである．cregit は，トークン分割の操作において srcML を使用しており，cregit の機能を用いることで，バージョン管理システム中のソースコード及び開発履歴を，srcML を実行してトークン分割された状態での差分に書き換えることができる．図 4 は，Java のソースコードを，cregit を用いてトークン分割した例を示している．

トークン分割されたソースコードに対するトークンの編集回数について記述する．図 5 の左側には，トークン分割する前の Java ソースコードについての編集差分情報が記されてい

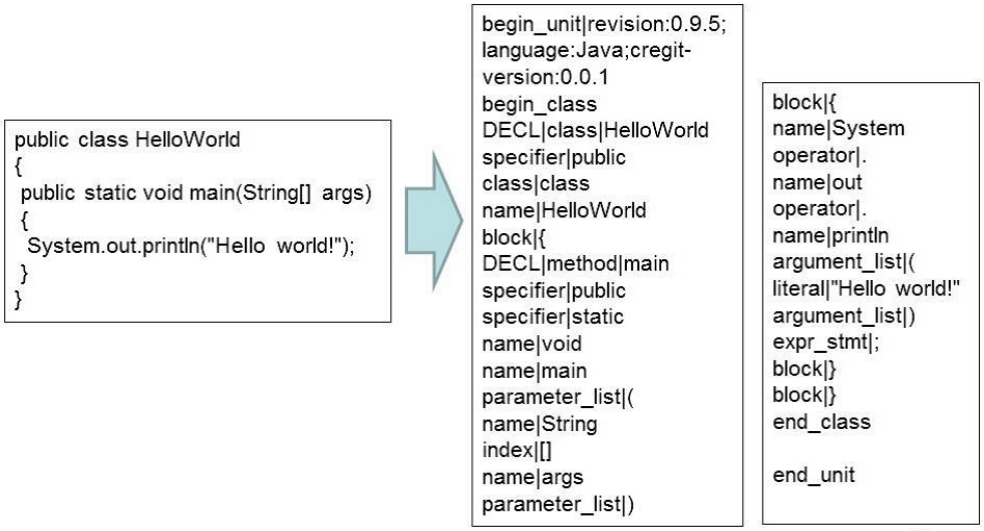


図 4: ソースコードのトークン分割例

る．ここでは，”Hello world!”を画面に表示する部分の文字列を”Hello”に修正して，さらに 3 回繰り返すために新たに変数宣言と for 文を追加している，といった編集内容である．差分情報の開始部分は@@で囲まれており，記述されているファイル内容の編集前及び編集後の範囲が行番号で記されている．図 5 の左側の場合は，編集前の 2 行目から 6 行分と，編集後の 2 行目から 9 行分の内容を表しているということになる．また，編集箇所は行毎に管理されており，削除された行は行頭に-が，追加された行は行頭に+が表示されている．図 5 の編集例では，1 文が削除され，4 つの行が追加されたと読み取れる．しかし，実際の編集作業としては，”Hello world!”を画面に表示する文に対しては削除ではなく，修正と新たなトークンの追加を行っていると考えられる．

図 5 の右側は，トークン分割後のソースコードに対する編集の差分を記述したものである．この表示形式では，1 トークンあたり 1 行で表示されているため，行に対する編集の差分をそのままトークンに対する編集と捉えることが出来る．また，”Hello world”を”Hello”と修正した箇所に対しては，一箇所の修正に対して 2 行に渡って出力を行っていることがわかる．トークンに対する修正箇所については，削除と追加が連続して行われているものとされている．そこで，本研究では，この差分情報を示した箇所の，追加行と削除行の数の総計を，そのファイルに対するトークンの編集回数と定義する．ただし，連続した削除行の直後に連続した追加行が現れている箇所は，削除行数分だけ修正が行われているものと考えられるため，その箇所の追加行分の計測は行わないものとする．また，そのソースコードに対するリネームについては，そのファイルのトークンが全く編集されていないため，編集回数としてカウントせず，リネーム後のソースコードに対する編集回数を引き続き計測するものと

する．

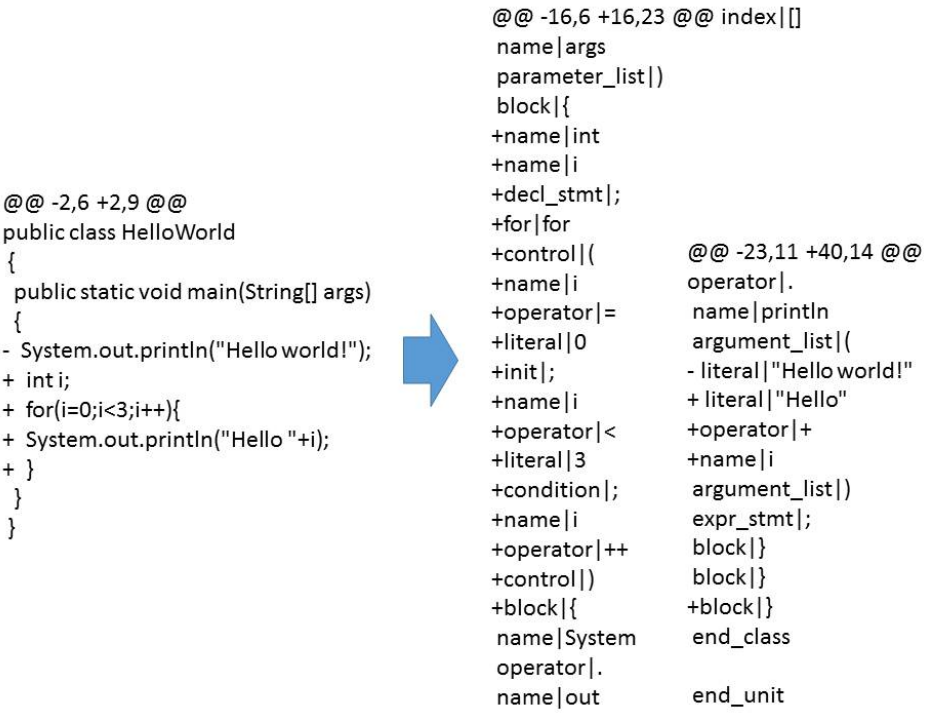


図 5: トークン分割されたソースコードの編集差分情報例

## 4　調査手順

本章では，開発履歴を用いて *CSF-Ownership* および *CST-Ownership* を算出するために実際に行った調査手順を述べる．図 6 は本研究の調査実験の手順の概要を示している．図 6 で表しているように，まず最初に開発履歴を記録しているソースコードリポジトリの，ある時点でのソースコードを対象にファイルクローンセットの検出を行う．次に，リポジトリのコミットログから，ファイルクローンセットに含まれているファイルに対して行われた編集の履歴を分析する．最後に，Ownership メトリックをファイルクローンセットに適用した *CSF-Ownership* および *CST-Ownership* を，上記で求めたファイルクローンセットとファイルの編集者，編集回数を用いて計算する．

本研究で調査実験の対象としたシステムは，バージョン管理システムである Git を用いて開発履歴が管理されているものであり，かつ Java 言語で記述されているオープンソースソフトウェアのプロジェクトである．その中でも，ある程度長い開発期間を経ており，ソースコード量が十分な 3 つのプロジェクトに対して調査を行った．対象プロジェクトの詳細を以下の表 2 に示す．

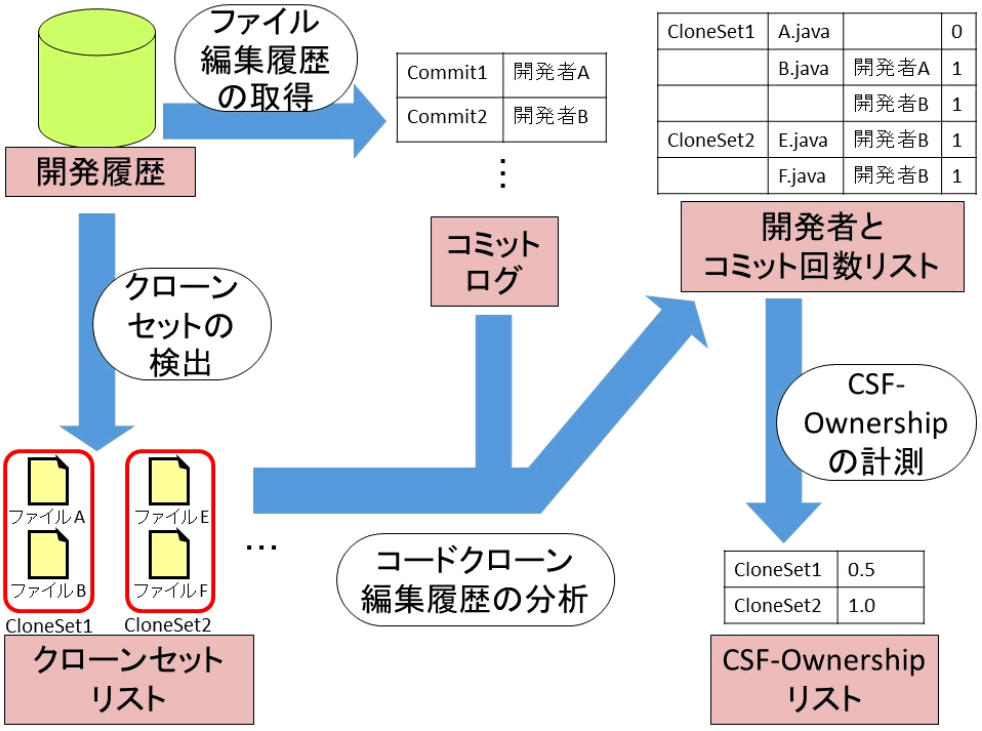


図 6: *CSF-Ownership* の調査手順

以下では，図 6 に基いて，調査の各手順について詳細に述べる．

表 2: 調査対象のソフトウェア

| | wildfly | Apache Ant | eclipse.jdt.core |
|---|---|---|---|
| クローン検出バージョン | 7.0.0.Alpha1 | 1.9.0 | 4.0.0 |
| 調査開始コミット日時 | 2010 年 11 月 4 日 | 2013 年 3 月 6 日 | 2015 年 6 月 24 日 |
| 調査終了コミット日時 | 2017 年 8 月 7 日 | 2017 年 7 月 23 日 | 2017 年 10 月 4 日 |
| コミット数 | 22604 | 790 | 1373 |

### 4.1 クローンセットの検出

対象となるソフトウェアの Git リポジトリから，コードクローンとなっているファイルクローンセットを検出する．検出するコードクローンの単位について，コード片単位でコードクローンを扱うと記述言語にイディオムを多く含むことがあり，検出されるコードクローンはコピーペーストによって生成された以外のものも含まれてしまう．一方でファイル単位のコードクローンは，コピーペーストによって作成されている可能性が高いと考えられる．よって，本研究ではファイルクローンを対象として調査を行った．また，本研究で扱うコードクローンは，変数名や関数名などのユーザ定義名と，パッケージ名，コメント部分を除いて，完全に一致するファイル単位でのコードクローンである．

図 6 の左下には，クローンセットの検出によって得られたファイルクローンリストを図示している．CloneSet1 は，ファイル A とファイル B が同じファイルクローンセットに属しているコードクローンである．CloneSet2 は，ファイル E，ファイル F がコードクローンとなっているクローンセットである．

調査対象のクローン検出は過去の特定のバージョン時点でのリポジトリに対して行った．これは，特定のバージョンをリリースした直後のコミットの中には，バグの修正などによるコードクローンに対する編集が多く含まれると考えられるためである．調査対象としたソフトウェアと，そのクローン検出に用いた対象のバージョンについて，表 2 に記述する．

### 4.2 ファイル編集履歴の取得

本調査では，コードクローンの生成する過程ではなく，コードクローンに対して行われた編集の回数を取得するのが目的である．したがって，コードクローンを検出したバージョン以降のコミットから，ファイル編集履歴を取得した．各システムに対する調査期間について，表 2 の調査開始コミット日時から調査終了コミット日時までに行われたコミットでの編集履歴を対象に調査を行った．対象システムの開発履歴から得られる編集履歴であるコミットログを取得し，各ファイル毎の編集者とその編集回数を取得する．

ファイル単位での編集回数を取得するにあたって，調査対象のファイルがどのコミットで編集されているかを調べることができるコミットログを取得する必要がある．本研究では，コミットログを取得するために git log コマンドを用いた．git log コマンドとは，Git リポジトリにおけるコミットの履歴を閲覧するのに用いられるコマンドのことである．このコマンドを用いることにより，あるコミットにおけるコミット ID やコミット日時，コミットを行った編集者やコミットコメントなど，様々な情報を得ることが出来る．また，オプションを用いることで，コミットの詳細なデータを確認できたり，出力形式を見やすい形に指定することもできる．

```
commit 15ac02089a84b992ce4246ed1e6cee1ba255d578
Author: Tsuji <k-tsuji@ist.osaka-u.ac.jp>
Date:   Thu Dec 20 15:27:30 2012 -0600

    Changed a file

M       src/NewClass.java

commit 403017e5e972635a95514926494ab41458191af2
Author: Tsuji <k-tsuji@ist.osaka-u.ac.jp>
Date:   Wed Dec 19 09:05:00 2012 +0000

    Changed many files

M       src/Main.java
A       src/NewClass.java
R060    src/Orig.java src/Renamed.java
D       src/Error.java
```

図 7: git log --nane-status コマンドの出力例

図 7 は，git log コマンドに，--name-status オプションを付けて実行した出力結果の例を示している．コミット ID やコミットした開発者 ID の他に，そのコミットで編集されたファイル名のリストが得られる．ファイル名のリストでは，各ファイルがどのような編集を受けているかを確認できるように英字がファイル名の前に記されており，英字はそれぞれ，M が内容の修正，A がファイルの追加，R がリネーム，D が削除を表している．本研究では，ここに表示されているファイルが，そのコミット内でファイル単位での編集を受けているものとする．この編集情報を特定期間中の全てのコミットから取得することで，その期間中に開発者が編集したファイルとその回数を計測した．

### 4.3 トークン編集履歴の取得

ファイル単位での編集履歴の取得とは異なり，トークンに対する編集履歴を取得するためには，ソースコードに対してトークン分割を行わなければならない．そこで，トークンに対する編集履歴を取得する方法として，本研究では，ソースコードそのものに加えて，リポジトリ中で管理されているファイルの差分情報をトークン分割することで各トークンに対する編集回数を数えるという手法を用いている．また，調査対象とするクローンセットについては，ファイル単位での編集回数計測の際に用いたファイルクローンセットと同じものとする．リポジトリ中のソースコードをトークン分割するために，cregit とよばれるツールの srcml2token という機能を用いる．srcml2token は，ソースコードを入力とし，トークンとその構文要素に分解し，1 トークン 1 行の形式に書き換えて出力する機能である．この機能を，bfg-repo-cleaner[19] いうツールを用いて，リポジトリ内の全ての履歴にあるソースコードに対して適用することでトークン分割する．bfg-repo-cleaner は，リポジトリの履歴を書き換えるツールである．ソースコードの変更に合わせてリポジトリの内容を書き換えることで，編集内容の履歴をトークンレベルで得ることができるようになる．履歴のトークン分割後のリポジトリをビューリポジトリと呼ぶ．ファイルが編集された際の差分を取得する方法として，本研究では，git log コマンドに，対象となるファイルの編集内容を編集前後の差分形式で表示するオプションである-p [ファイル名] を用いている．このコマンドをビューリポジトリに対して実行することで，特定のファイルの，トークン単位での編集履歴を取得した．

### 4.4 コードクローン編集履歴の分析

上記の方法で，各ファイルクローンのファイル単位，もしくはトークン単位での編集履歴を取得できる．これらの編集履歴と，クローン検出の手順で求めたクローンセットとを用いることで，クローンセット中のファイルに対する編集回数をファイル単位，トークン単位それぞれで求めることが出来る．

このとき，コミットに含まれるファイルのリネームについては，ファイル単位での編集回数には含めるが，内容は編集されていないとみなすため，トークン単位での編集回数には数えないものとする．リネーム後のファイルに対する編集回数は元のファイルへの編集とみなして計測を行う．

この手順で，対象とするクローンセットに含まれる全てのファイルについて，編集した全開発者とそのファイル単位またはトークン単位での編集回数を求めた．

### 4.5 メトリックの計算

クローンセットに含まれる各ファイルに対する全ての編集者とその編集回数を用いて，*CSF-Ownership* と，*CST-Ownership* を求めることが出来る．最初に，各クローンセットに対する総編集回数を求める．クローンセット毎に，コードクローンとなっている全てのファイルへのファイル単位，トークン単位での総編集回数を合計することで求めることができる．この回数が，式 (2) の $CSF_{total}$ もしくは式 (3) の $CST_{total}$ に該当する．次に，そのクローンセットを最も多く編集した開発者による編集回数を特定する．各開発者のクローンセットに対する編集回数は，そのクローンセットを構成する各クローンファイルへの編集回数の合計とするので，その中で最も編集回数の多いものが，式 (2) の $CSF_{max}$ もしくは式 (3) の $CST_{max}$ に該当する．これらの値を用いることで，*CSF-Ownership* と，*CST-Ownership* を算出した．

## 5　調査結果と考察

本章では，本研究で行った *CSF-Ownership* および *CST-Ownership* の調査結果を示し，それに基づいて考察を行う．

### 5.1　*CSF-Ownership* の調査結果と考察

表 3 は，3 つの調査対象ソフトウェアについての，*CSF-Ownership* の値をまとめたものである．

表 3: *CSF-Ownership* の調査結果

| *CSF-Ownership* | wildfly | Apache Ant | eclipse.jdt.core |
|---:|---:|---:|---:|
| $CSF\text{-}Ownership = 0$ | 0 | 1 | 749 |
| $0 < CSF\text{-}Ownership \leq 0.5$ | 9 | 0 | 1 |
| $0.5 < CSF\text{-}Ownership < 1.0$ | 4 | 0 | 1 |
| $CSF\text{-}Ownership = 1.0$ | 1 | 2 | 35 |
| total | 14 | 3 | 786 |

wildfly については，*CSF-Ownership* が 0.5 以下のクローンセットが 9 個あり，14 個あるクローンセットのうち半数以上に主要な開発者が存在しないという結果となった．このようなプロジェクトに対しては，コードクローンの編集管理を支援するツールは有用であるといえる．

Apache Ant については，元々クローンセットは 3 つしか存在していなかったが，そのうちの一つについては，クローンセット中に 16 個ものファイルクローンを持っているものが検出されている．このクローンセット中のファイル全てに対して 1 度編集された記録が残っているが，これらの編集は全て，同じ編集者が同じコミット内で行ったものであり，編集内容も全て同じものであるため，*CSF-Ownership* の値は 1.0 となっている．その他のクローンセットに対しても一貫した編集が保たれており，特定の開発者によるコードクローンの編集管理が行われていることが伺える．

eclipseJDT については，殆どのコードクローンに対して全く編集がされていないことがわかった．編集を受けたクローンセットは 37 個あるが，そのうち 2 個については *CSF-Ownership* の値が 0.5 を下回っている．このような値が確認できたため，eclipseJDT については基本的にはクローンセットに対する主要な開発者が存在し管理がなされているが，コードクローンの編集管理が行き届いていない可能性のあるクローンセットが存在すると言える．

### 5.2 *CST-Ownership* の調査結果と考察

表 4 は，3 つの調査対象ソフトウェアについての，*CST-Ownership* の値をまとめたものである．

表 4: *CST-Ownership* の調査結果

| *CST-Ownership* | wildfly | Apache Ant | eclipse.jdt.core |
|---:|---:|---:|---:|
| $CST\text{-}Ownership = 0$ | 2 | 1 | 761 |
| $0 < CST\text{-}Ownership \leq 0.5$ | 8 | 0 | 1 |
| $0.5 < CST\text{-}Ownership < 1.0$ | 3 | 0 | 1 |
| $CST\text{-}Ownership = 1.0$ | 0 | 2 | 23 |
| total | 14 | 3 | 786 |

トークン単位での *CST-Ownership* は，3 つの対象システムとも，分布の内訳がファイル単位の *CSF-Ownership* の内訳とほぼ変わらなかった．ファイル単位での編集割合とトークン単位での編集割合に変化が無いということは，各編集者とも 1 コミット当たりの編集量に大きな差が見られなかったということが言える．

表 4 の wildfly に存在している *CST-Ownership* の値が 0 になっているクローンセットが存在している．これらのクローンセットは，*CSF-Ownership* の値は 0 になってはいない．このようなケースが見られるのは，ファイル単位での編集回数ではファイルのリネームを 1 回の編集としているが，トークン単位での編集回数ではリネームを数えていないためで，リネームのみが行われたクローンセットについてはこのような結果となっている．

### 5.3 調査結果の詳細

eclipse.jdt.core のクローンセットの多くが，*CSF-Ownership*，*CST-Ownership* の 2 つの値とも 0.0 もしくは 1.0 であったが，2 つのクローンセットは *CSF-Ownership* の値が 0.5 を下回っていた．つまり，これらのクローンセットには主要な開発者が存在していないため，クローンセットの編集管理を支援する必要があると考えられる．このうち，*CSF-Ownership* の値が 0.22 と特に低く，*CST-Ownership* の値も 0.5 以下であったクローンセットについて，その編集者と編集内容に着目した詳細な調査を行った．

図 8 は，eclipse.jdt.core から検出されたクローンセットに対して行われた編集のうち，コードクローンに対する一貫していない編集を抜粋した概要図である．このクローンセットは，2 つのファイルによって構成されたコードクローンである．ここでは便宜的にファイル A,B と呼び，編集を行った開発者を開発者 a，開発者 b，開発者 c とする．初めはコードクロー

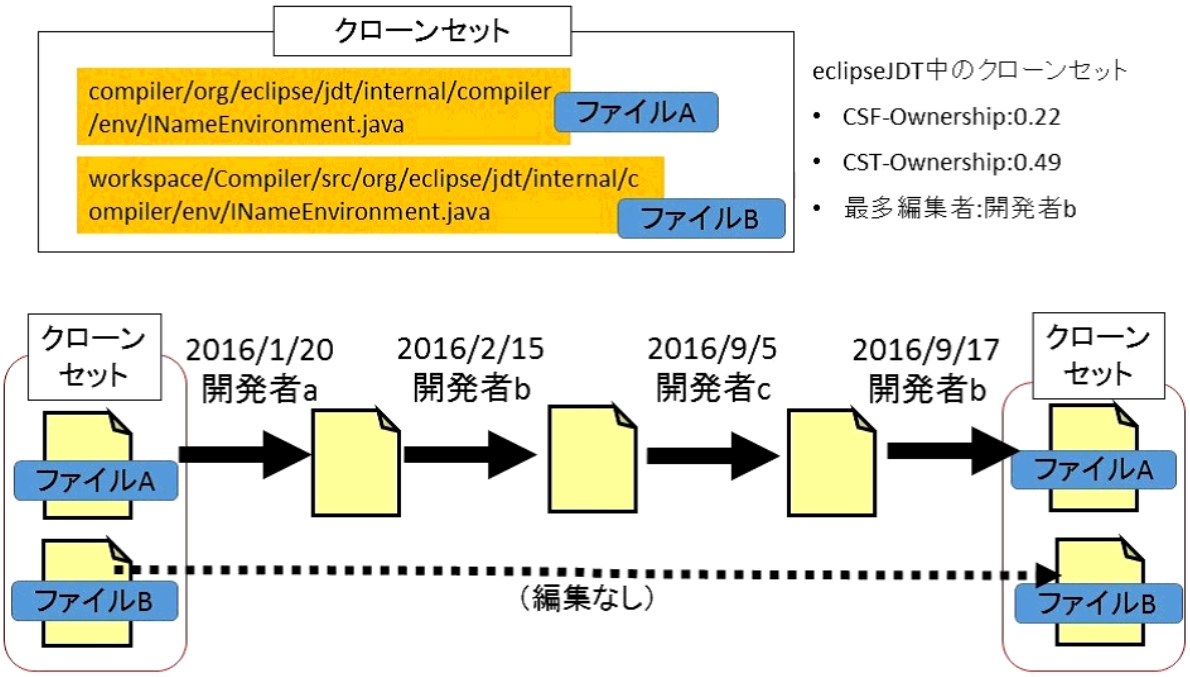


図 8: クローンセットへの一貫していない編集の概要

ンであった両ファイルだが，ファイル A が，開発者 a によって編集される．このとき，ファイル B に対しては編集が行われていなかったため，2 つのファイルはコードクローンでは無くなった．その後は，開発者 b，開発者 c らによって編集を受けていたが，続く開発者 b の編集によって，ファイル A,B は再びクローンセットに戻っている．このような編集は，Late Propagation のタイプ 8 に該当するものである．Late Propagation は，そうではないコードクローンに対する編集よりも，バグを含む危険性の高いといわれている現象である．さらに，Late Propagation の中でも，特にタイプ 7 とタイプ 8 の編集が行われている箇所には，バグを含む危険性が高い．*CSF-Ownership* の値の低いクローンセットの中に，バグを含む危険性の高い編集が行われているものを確認することができた．

### 5.4 妥当性への脅威

クローンセットに対する編集について，低いメトリックの値が出たクローンセットに対する編集の危険性と，高いメトリックの値を示したクローンセットに対する編集の危険性については，網羅的な調査を行ったわけではない．そのため，提案したメトリックの低さから危険な編集を特定することができるかどうかは不明である．

また，本研究では，Git で管理されており，Java で書かれたソフトウェアを対象に調査を

行ったが，他の言語で書かれたソフトウェアに対する調査では，主にトークン単位での編集回数の割合に関して結果が変わる可能性がある．

## 6　まとめと今後の課題

本研究では，ソフトウェア開発におけるクローンセットの編集管理の必要性を調べるために，3 つのオープンソースソフトウェアに対して，そのクローンセットに対する編集者とその編集人数についての調査を行った．調査にあたり，コードクローンに対して Ownership を適用した新たなメトリックである *CSF-Ownership* と，*CST-Ownership* を提案し，それを用いて．調査の結果，オープンソースソフトウェアの中には，主要な編集者がおらず，複数人でクローンセットに対して編集を行って開発を進めているプロジェクトが存在していることがわかった．また，複数人で編集を行っているコードクローンの中に，今後欠陥を含む危険性の高い編集が行われているものが確認できた．今後の課題としては，有用性の検証が挙げられる．本研究で用いたメトリックからクローンセットに対する一貫していない編集を見つけることができたが，メトリックの大小と編集の危険性についての関係性を明らかにできてはいない．そのため，編集内容について網羅的に調査を行い，例えば低い *CSF-Ownership* および *CST-Ownership* を示すクローンセットに対する編集が高い危険性となっている割合を検証により確認できれば，提案メトリックの有用性を示すことができると考えられる．

## 謝辞

本研究において，適切なご助言およびご指導を賜りました，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 教授に心より感謝いたします．研究における有用なツールをご紹介頂いた他，私の入院や就職活動に関してのご相談に親身になって応じてくださった事が本当に励みになりました．本当にお世話になりました．

本研究において，随時適切なご助言およびご指導を賜りました，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 松下 誠 准教授に深く感謝いたします．研究グループや研究室でのイベントを主導してくださるだけでなく，発表練習などの場で議論を纏め，また学生とも気さくに接して頂けることで，気軽に発言できる環境を作って下さり，快適な環境で研究生活を送ることができました．心より感謝いたします．

本研究において，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の 春名 修介 特任教授には，研究内容のご指摘のみならず，発表における慣習などについてもご指導賜りました．のみならず，就職活動においても親身になって相談に応じて下さったこともありました．私が将来の憂いなく研究に励むことができたのは春名先生のおかげであると思っています．本当にありがとうございました．

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の 神田 哲也 特任助教には，研究生活についてのご指導だけでなく，私生活における相談も含めて数多くのアドバイスを賜りました．ツールの導入や実装の際には特にお世話になりました．本当にありがとうございました．

本研究において，多大なるご助言およびご指導を賜りました，名古屋大学大学院情報学研究科附属組込みシステム研究センター 吉田 則裕 准教授に深く感謝いたします．私の研究態度に対して度々ご叱責を受けましたが，吉田先生の研究内容への的確なご指摘を数々頂けたことで，私の研究はこのように形にできました．研究で行き詰まっても，先生に相談することで再度進めることができたという機会は幾度となくありました．最後まで頼りになる先生でした．改めて，本当にありがとうございました．

本研究において，終始適切なご助言およびご指導を賜りました，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研究室 崔 恩瀞 助教に深く感謝いたします．お忙しい中でも研究に対して数多くのご指摘を頂いただけでなく，生活態度におけるご叱責やご指摘も数多くして頂けました．私がこうして論文を書き上げる段階まで至ることができたのは，崔助教のおかげであると，心より感謝いたします．本当にありがとうございました．

大阪大学大学院大学院情報科学研究科コンピュータサイエンス専攻の事務職員である 軽部 瑞穂 氏には，研究生活において非常に多大なご気遣い，ご支援を頂きました．特に，私が病気で療養中の間も，適宜連絡を下さったこと，非常に感謝しております．長い休学期間

からスムーズに復学でき，様変わりした研究室にも私が馴染むことができたのは，軽部氏のご支援があったからこそであると確信しております．心より深く感謝しております．本当にありがとうございました．

最後に，ご指導，ご助言を頂き私を支えてくださった，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] Lerina Aversano, Luigi Cerulo, and Massimiliano Di Penta. How clones are maintained: An empirical study. In *Software Maintenance and Reengineering, 2007. CSMR'07. 11th European Conference on*, pp. 81–90. IEEE, 2007.

[2] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Proc. of WCRE '95*, pp. 86–95, 1995.

[3] Liliane Barbour, Foutse Khomh, and Ying Zou. Late propagation in software clones. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pp. 273–282. IEEE, 2011.

[4] Liliane Barbour, Foutse Khomh, and Ying Zou. An empirical study of faults in late propagation clone genealogies. *J. Softw. Evol. and Proc.*, Vol. 25, No. 11, pp. 1139–1165, 2013.

[5] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. Don't touch my code!: Examining the effects of ownership on software quality. In *Proc. of ESEC/FSE '11*, pp. 4–14, 2011.

[6] Eunjong Choi, Norihiro Yoshida, Yoshiki Higo, and Katsuro Inoue. A clone detection approach for a collection of similar large-scale software products. *IEICE Technical Report*, Vol. 112, No. 275, pp. 117–121, 2012.

[7] Daniel M German. cregit: identifying contributors of source code. `https://events.static.linuxfound.org/sites/events/files/slides/cregit-2.pdf`.

[8] J. Harder. How multiple developers affect the evolution of code clones. In *Proc. of ICSM '13*, pp. 30–39, 2013.

[9] 肥後芳樹, 楠本真二, 井上克郎. コードクローン検出とその関連技術. 電子情報通信学会論文誌, Vol. J91-D, No. 6, pp. 1465–1481, 2008.

[10] P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proc. of STOC '98*, pp. 604–613, 1998.

[11] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. DECKARD: Scalable and accurate tree-based detection of code clones. In *Proc. of ICSE '07*, pp. 96–105, 2007.

[12] Lingxiao Jiang, Zhendong Su, and Edwin Chiu. Context-based detection of clone-related bugs. In *Proc. of ESEC-FSE '07*, pp. 55–64, 2007.

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

[14] Zhenmin Li, S. Lu, S. Myagmar, and Yuanyuan Zhou. Cp-miner: finding copy-paste and related bugs in large-scale software code. *Software Engineering, IEEE Transactions on*, Vol. 32, No. 3, pp. 176–192, 2006.

[15] Michael L. Collard. srcML. `http://www.srcml.org/`.

[16] Dhavleesh Rattan, Rajesh Bhatia, and Maninder Singh. Software clone detection: A systematic review. *Inform. Softw. Tech.*, Vol. 55, No. 7, pp. 1165 – 1199, 2013.

[17] R. L. Rivest. The MD5 message digest algorithm. Internet RFC 1321, 1992.

[18] C. K. Roy and J. R. Cordy. A survey on software clone detection research. School of Computing TR 2007-541, Queen ' s University, Vol. 541, , 2007.

[19] rtyley. BFG Repo-Cleaner by rtyley. `https://rtyley.github.io/bfg-repo-cleaner/`.

[20] Suresh Thummalapenta, Luigi Cerulo, Lerina Aversano, and Massimiliano Di Penta. An empirical study on the maintenance of source code clones. *Empirical Software Engineering*, Vol. 15, No. 1, pp. 1–34, 2010.

[21] Yuki Yamanaka, Eunjong Choi, Norihiro Yoshida, Katsuro Inoue, and Tateki Sano. Applying clone change notification system into an industrial development process. In *Proc. of ICPC '13*, pp. 199–206, 2013.