リポジトリマイニング可能なコードクローン版管理システムの提案

畑 秀明[1,a)]　肥後 芳樹[1,b)]　楠本 真二[1,c)]

概要：ソフトウェアのソースコード中には，同一または類似したコード片，コードクローンが存在する．こういったコードクローンには，不具合の温床となるものだけでなく，保守性の面で好ましいものもあるとの報告が近年されている．例えば，長期間安定したコードクローンは除去する必要のないものが多い．こういった新しい知見の発見と活用のためには，コードクローンの履歴管理や分析が重要である．これまでのコードクローン履歴分析の研究では，コードの変更のみに着目した分析が行われている．一方，リポジトリマイニング（開発履歴のデータマイニング）の研究では，コードだけでなくプロセスや人的属性に関する履歴情報から有用な知見が報告されている．そこで，本稿では，こういった多様な面でのリポジトリマイニングが可能であるコードクローン版管理システムを提案する．6 つのオープンソースソフトウェアへ適用し，その有用性を報告する．

Code Clone Version Control System for Mining Rich Clone Histories

Hata Hideaki[1,a)]　Higo Yoshiki[1,b)]　Kusumoto Shinji[1,c)]

Abstract: In source code, there exist code clones, that is duplicated or similar code fragments. Recent empirical studies reported that not all code clones are harmful but some code clones have positive impact on software maintenance. For example, we do not need to eliminate long-lived clones. To find more effects of clones and manage clone evolutions, it is important to analyze clone histories and manage them. State-of-the-art studies of clone evolution concentrate only on code-related histories. However, the research area of mining software repositories has found the usefulness of not only code-related but process-related and developer-related histories. Therefore, we propose code clone version control systems that enable us to mine rich clone histories including code-related, process-related, and developer-related histories. We applied our systems to 6 open source software repositories to show the usefulness.

1. はじめに

ソースコード中に存在する同一または類似したコード片をコードクローンと呼ぶ．不用意なコピーアンドペーストによるコードクローンの生成は，ソフトウェアの保守を困難にするといわれており，多くの研究が行われてきた [19], [21].

一方，実証的な分析からコードクローンの負ではない面を指摘する以下のような報告がある．

- コードクローンの生成，すなわち類似コードの作成は適切なコードパターンの実現のような好ましい現象であることが多々あり，保守性の面で好ましいものも多い [8], [15] .
- コードクローンと不具合の相関を調査した場合，多くの不具合はコードクローンとの相関が小さい．また，クローンとなるコード片の方が，クローンとならないコード片と比べて不具合との相関が小さい [13].
- 長期間存在する安定したコードクローンがあり，それらは除去する必要がない [2], [9].

特に，近年コードクローンの履歴を対象とした分析で新たな知見が報告されており，多くの実証的な分析を行うことが重要である．こうした開発履歴の分析は，リポジトリマイニングと呼ばれ多くの研究が行われている [23]．特に

[1] 大阪大学
Osaka University, Suita, Osaka 565–0871, Japan
[a)] h-hata@ist.osaka-u.ac.jp
[b)] higo@ist.osaka-u.ac.jp
[c)] kusumoto@ist.osaka-u.ac.jp

不具合予測の分野では，近年リポジトリマイニングの有用性が注目されている．マイニングの対象として，コードの履歴だけでなく，プロセスや人的属性の履歴についてのデータも重要とされている [18]．しかしコードクローン履歴の研究では，コードに着目した研究しか行われていない．

そこで本稿では，コード履歴だけでなく，コードクローンに関わる開発履歴のプロセスや人的属性に関する情報についてのリポジトリマイニングが可能なコードクローン版管理システムを提案する．提案するシステムを 6 つのオープンソースソフトウェアへ適用し，評価を行った．また，プロセスや人的属性に関するコードクローン履歴のマイニングを実現した．

以降，2 章でリポジトリマイニングやコードクローンの履歴分析に関する関連研究を議論し，3 章でコードクローン版管理システムを提案する．4 章でオープンソースソフトウェアへの適用結果とリポジトリマイニングの分析結果を報告し，5 章で議論する．最後に 6 章でまとめる．

2. 関連研究

2.1 リポジトリマイニング

開発履歴が蓄積されたソフトウェアリポジトリに対するデータマイニング（リポジトリマイニング）は，実際のソフトウェア開発における有用な知見の発見が期待され，近年の実証的ソフトウェア工学で多くの研究が行われている [23]．近年のソフトウェア工学の動向としても，実証的証拠（empirical evidence）を発見，収集することが重視されている [10]．

ソースコードの版管理システムを対象としたリポジトリマイニングが盛んな研究テーマに不具合予測がある [18]．不具合予測は，予測モデルの構築にソフトウェアメトリクスを用いる [22]．このソフトウェアメトリクスとして，リポジトリマイニングで得られる開発履歴メトリクスが注目されている．開発履歴メトリクスとしては，ソースコードの変更履歴に注目したコード関連のもの，モジュールの変更回数や過去の不具合修正の有無といったプロセス関連のもの，開発した組織や開発者数などの開発組織関連のもの，分散開発に注目した地理的位置関係に関連するものなどが提案され，多くのソフトウェア開発プロジェクトへの適用からその有効性が報告されている [18]．

こういった成果から，リポジトリマイニングではコードの履歴情報だけでなく，開発プロセスや人的属性など様々な観点での実証的証拠の収集が，重要となると考えられる．

2.2 コードクローンの履歴分析

開発履歴を分析して，コードクローンの変遷を調査する研究が近年数多く行われている [12]．単一の版に対するコードクローン分析と比べて，複数の版におけるコードクローン分析では，コードクローンとなったソースコードがどのように変更されたかといった有用な特性が得られることから，注目されている．

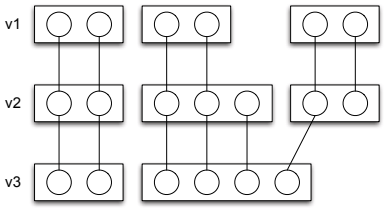


図 1 コードクローン履歴
Fig. 1 Code clone histories

ここでコードクローンの履歴の例を**図 1** に示す．図の丸がコード片を表し，それを囲む四角がコードクローンの関係にあることを示す．それぞれの四角，つまりコードクローン関係にあるコード片の集合をクローンセットと呼ぶ．また，水平方向の層はある版でのコードクローンの様子を表す（ここでは v1 から v3 までの版がある）．丸を結ぶ線は，版間でのコード片の変遷を表す．図 1 では，版 v1 の左側に存在するクローンセットが版 v3 まで存在すること，版 v1 の真ん中に存在するクローンセットには版 v2 でコード片の追加があり，版 v3 では残りのクローンセットと合流していることがわかる．コードクローンの履歴に対しては，コード片に対する変更だけでなく，クローンセットの変遷などが分析対象となる．

コードクローン履歴を分析した初期の研究には Kim らの成果が挙げられる [9]．Java 言語で書かれたオープンソースソフトウェアを対象としたクローンセット変遷の分析から，36 から 38% のクローンセットはセット内で一貫した変更が行われていること，49 から 64% のクローンセットは除去が簡単ではない，特に長期間存在するコードクローンは除去の必要がないこと，また，48 から 72% のクローンセットはわずかの期間で消滅することを報告している．

コードクローンセットに対する一貫しない変更，すなわちクローンセット内の一部のコード片に行った変更を残りのコード片に行わないことは，修正漏れなどの不具合混入の要因と考えられる．Juergens らは，2 つの社内プロジェクトと 1 つのオープンソースプロジェクトを対象とした大規模な分析から，一貫しない変更が不具合に結びつきやすいと報告している [7]．一方 Göde らは，多くのコードクローンセットはあまり変更されず，一貫しない変更はあまりないことを報告している [2]．

Thummalapenta らは，C 言語と Java 言語で書かれた 4 つのオープンソースプロジェクトとのコードクローン履歴分析から，クローンセットに対する変更は一貫して直ちに行われること，コードクローンをテンプレートとして用いるのはソフトウェアシステム内の共通する現象であること，クローンの粒度やサイズ，範囲，プログラミング言語といった特徴はコードクローン変遷のパターンに影響を与

えないこと，また，一貫した変更を遅れて行う変更（late propagation）は不具合修正の場合が多いことを報告している [15].

従来のコードクローン履歴分析では，コードの履歴情報のみを分析対象としている．本研究は，開発プロセスや人的属性の観点でもコードクローン履歴を分析することを目指す．

2.3 コードクローンの履歴管理

Duala-Ekoko らは，CloneTracker というコードクローン履歴管理システムを提案している [1]．CloneTracker は Java のメソッド内のブロックに対するコードクローン履歴を追跡するシステムで，Eclipse のプラグインとして実装されている．

Nguyen らは，コードクローンの履歴管理だけでなく，クローンセットに対して一貫した変更への支援を行う JSync というシステムを提案している [11]．システムのデータベースには，コード片，コードクローン関係，コードクローンへの変更などの情報を保存している．

コードクローンの履歴管理は，コード片のモジュールの履歴だけでなく，クローンセットというモジュール集合の履歴も管理する必要があり，挑戦的な課題である．上述の 2 つのシステムは，コードに関する履歴情報を保持しているが，2.1 節で議論した開発プロセスや人的属性などの面でのリポジトリマイニングは難しい．

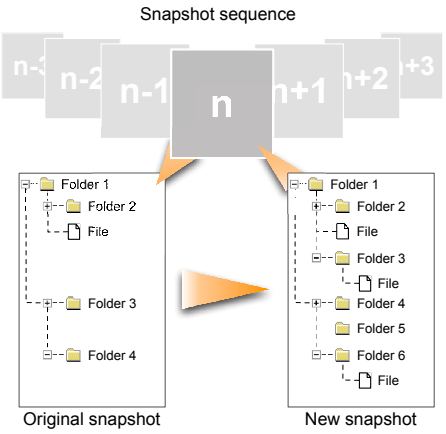
3. コードクローン版管理システム

3.1 コードクローン履歴の追跡

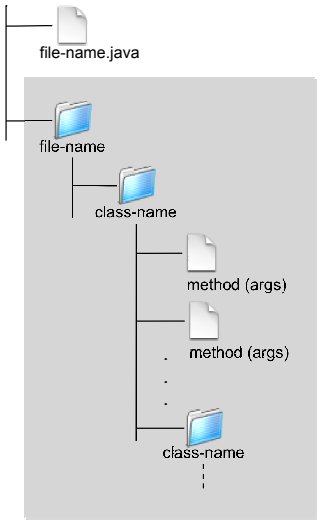
Zibran らはコードクローン履歴を追跡する技術を以下のようにまとめている [16].

(1) 各リビジョンでコードクローンを特定し，連続するリビジョンのクローン間で対応付けを行う．
(2) 最初のリビジョンでコードクローンを特定し，各クローンを以降のリビジョンにマッピングする．
(3) インクリメンタルなコードクローン検出ツールでクローン特定とリビジョン間の対応付けを行う．
(4) 各リビジョンでコードクローンを特定する．連続するリビジョン間で関数の対応付けを行い，その情報を用いてクローン間の対応付けを行う．

(1), (4) のアプローチは，注目する全リビジョンに対して任意のコードクローン検出ツールを用いてコードクローンを特定することができるが，クローン間や関数間の対応付けに時間がかかる．そのため分析可能なリビジョン数が限られる．(2)，(3) のアプローチは高速であるため，多くのリビジョンを対象とできる．しかし，(2) では 2 番目以降のリビジョンで発生したクローンを特定できない．また，(3) は特定のツールに依存し，非インクリメンタルなコードクローン検出ツールで特定可能なクローンを特定できない可能性がある．



(a) スナップショットの再構成



(b) ディレクトリ構造

図 2 Historage のアイディア
Fig. 2 Key idea of Historage

3.2 要求

版管理システムは，コードの変更履歴だけでなく，変更日時，開発者名，コミットログなどの開発プロセスや人的属性に関する情報をも保持している．この情報や，さらには障害管理システムや開発者データベースなどの情報を組み合わせることで多様なリポジトリマイニングが可能になっている．

コードクローンに関しても同様のリポジトリマイニングを可能にしたい．コードクローン版管理システムについての要求を以下に示す．

(1) 全リビジョンでコードクローンを特定する．
(2) 全てのコードクローンの履歴を保持する．
(3) コードクローンの履歴を追跡できる．
(4) 開発プロセスや人的属性に関する情報を保持する．

3.3 キーアイディア：Historage

3.1 節で紹介した既存技術の課題や，3.2 節で掲げた要

求に対するキーアイディアは，先に提案した細粒度履歴管理リポジトリ，Historage である [3]．Historage は，ソースコード版管理システム Git の仕組みを活用して実現した [3], [17]．

Git はソースコードファイルの履歴を，(1) 各版のスナップショットを独立に保存し，(2) 連続するスナップショット間の差分からファイルに対する変更操作を推定する，という仕組みで管理する．あるコミットの前後のスナップショット間での，ファイルの有無や中身の違い，ファイル名や存在するパスの違いから，そのコミットで特定のファイルに対して追加，削除，修正，名前変更，移動といった変更操作が行われたと報告する．また，関連する全スナップショット間で同様の処理を行い，特定のファイルに対する全履歴を出力する．

Historage は，細粒度なモジュールである Java のメソッド履歴を管理するシステムである．履歴を追跡したいメソッドをそれぞれ 1 つのファイルとして各スナップショットで保存することで，上述の Git の仕組みを活用してメソッドの全履歴を管理する．**図 2**(a) に示すように，既に通常のソースコードの履歴が蓄積された Git リポジトリの各スナップショットを再構成することで，既存のファイルレベルの履歴管理システムをメソッドレベルの履歴管理システムにする．

このスナップショットの再構成では，1 つのソースコードファイルに対して図 2(b) の灰色部分に示すようなディレクトリ構造を作成する．各メソッドはそのメソッドシグネチャをファイル名とし，ソースコードのファイル名（の拡張子を除去）を名前とするディレクトリに格納する．モジュール履歴で最も難しい課題は，モジュールの中身の変更と名前変更や移動が同時に行われた場合のモジュールの追跡である．オープンソースソフトウェアでの適用実験で，メソッドテキストの類似度が 30% 以上であれば 95% 以上の精度でメソッドの追跡ができることを確認している．これは実用上十分な精度である[*1]．

Historage の基本的なアイディアは，履歴の管理を行いたいモジュールをディレクトリ構造に配置し，全スナップショットで保存させることで Git の仕組みで履歴を追跡するというものである．Historage では，Java のメソッドが履歴を管理したいモジュールであった．本研究で対象とするコードクローンでは，コードクローンとなるモジュールとクローンセットの履歴管理が必要である．これらをディレクトリ構造に配置し，スナップショットを保存することができれば Historage と同様のアプローチで実現することができる．

3.1 節で議論した既存技術と比べると，本アプローチは全リビジョンでコードクローンの特定を行うが，明示的に対応付けを行って保存しないので高速である．対応付けは履歴を出力するときに Git によって行われる．

[*1] Git は類似度 50% をデフォルトの閾値にしており，それより低い類似度でも高い精度で追跡ができているから

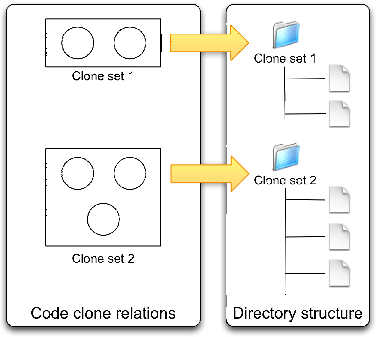


図 3 コードクローン関係を保持したディレクトリ構造
Fig. 3 Directory structure representing code clone relations

3.4 提案システム：CloneHistorage

コードクローンの履歴管理を行うシステム CloneHistorage を提案する．**図 3** に示すように，それぞれのクローンセットごとにディレクトリを作成し，その中にコードクローンとなるモジュールを配置させる．これによって，コードクローン関係を保持したディレクトリ構造となる．図 3 の左側に示す 2 つのモジュールを含むクローンセット 1 と 3 つのモジュールを含むクローンセット 2 は，図 3 に右側に示すディレクトリ構造となる．このようなディレクトリ構造にすることによって，各ファイルに対する変更を分析することでコードクローンとなるモジュールの履歴が，ディレクトリに対する変更を分析することでクローンセットの履歴が得られる．

CloneHistorage の実現には，以下の 2 点を決定する必要がある．

モジュールの粒度と識別　モジュールの粒度には，ファイル，メソッド，ブロックなどが考えられる．これはコードクローン検出手法にも依存する．対応するファイル名を定める必要がある．

クローンセットの識別　対応するディレクトリの名前が必要である．

図 3 に示したように，クローン検出対象のモジュールとクローンセットは，それぞれファイルとディレクトリに対応させる．このときそれぞれに名前が必要である．この名前付けには以下の要求がある．

- それぞれを識別するために，1 つのスナップショットで唯一の名前をつける必要がある．
- スナップショット間で追跡可能な名前である必要がある．同じまたは対応する要素に同じ名前を付けることができれば追跡が容易である．名前変更の検出は，スナップショット間の異なる名前の要素間で行われる．そのため，対応しない要素に同じ名前を付けてはならない．例えば，版 v1 で要素 1 に A，要素 2 に B と名

前を付け，版 v2 で要素 1 に B，要素 2 に C と名付けると，正しく履歴を追跡できない．Git は同じ名前の要素を追跡するため，B と名付けられた要素を追跡すると版 v1 で要素 2，版 v2 で要素 1 という誤った履歴となる．このため，連番などの単純な名前付けは適切でない．

以上の要求に加えて，履歴を理解するときには，分かりやすい名前である方が望ましい．そのため，適切な名前付けは CloneHistorage の大きな課題の 1 つである．

2.3 節で紹介した，JSync は，クローンセットに名前を付けて識別していない [11]．また，CloneTracker はクローンセットに連番を振って識別している [1]．クローンセットに名前をつけて管理することは，我々の知る限り行われていない．

3.5 実装

3.5.1 コードクローン検出対象モジュール

コードクローン検出対象となるモジュールには，Java のメソッドを選択した．これは，ファイルレベルと比べて細粒度な分析ができ，メソッドシグネチャを名前とすれば 3.4 節で示したモジュールの名前付けに対する要求を満足できるからである．加えて，先の研究でメソッドの履歴追跡が実用上十分な精度で行えることを確認できている [3], [17].

コードクローン検出には，検出ツール Scorpio を用いた [6], [20]．Scorpio はソースコードをプログラム依存グラフとしてモデル化し，類似する部分グラフの特定からコードクローンセットを検出する．プログラム依存グラフを用いたコードクローン検出は，検出の精度が高く，ソースコード上で非連続な要素群に対してもコードクローン関係を検出ができる．また，Scorpio は計算コストを削減する工夫によって，他のプログラム依存グラフを用いたコードクローン検出ツールと比べてスケーラビリティが高い．さらに，Scorpio はメソッドごとにプログラム依存グラフを作成するので，メソッド間のコードクローン検出ができる．本稿の Scorpio でのコードクローン検出では，検出する同型部分グラフの頂点数の最小数を 10 とした．

3.5.2 クローンセットの識別

クローンセットの名前は，属するコード片の処理を表すものであれば理解しやすく望ましい．しかし，ソースコードの処理を人が理解できるよう分かりやすくまとめるのは，挑戦的な課題であり難しい [14].

事前のコードクローン分析から，クローンセットとなるコード片を含むメソッドには，同じ名前や似た名前のものが多いことを確認した．メソッド名はメソッドの処理を表しており，クローンセットに含まれるメソッド名を参考にすることは妥当である．そこで以下の手順でクローンセットの名前を決定する．

(1) クローンセット内の全メソッド名が一致した場合，そのメソッド名をクローンセット名とする．
(2) 全メソッド名は一致しないが，全メソッド名に共通部分がある場合，異なる部分を @ に置換したものをクローンセット名とする．例えばクローンセット内に含まれるメソッドが `bindGridView` と `bindListView` というメソッド名である場合，クローンセット名は `bind@View` とする．
(3) (2) で共通部分が 3 文字未満の場合（共通部分が無い場合も含む），メソッド名が長い順，同じ場合は辞書順に 2 つ選び，2 つのメソッド名の最長共通部分文字列 (longest common subsequence) をクローンセット名とする．Java のメソッド名は慣習として単語の最初が大文字の複合語であるので，単語に分解して単語単位で最長共通部分文字列を取得する．例えば，クローンセット内の 3 つのメソッド名が `dismissAllOtaDialogs` と `dismissAllDialogs`，`onDisconnect` の場合，クローンセット名は，先の 2 つのメソッド名の最長共通文字列から `dismissAllDialogs` となる．
(4) (3) で最長共通文字列が 3 文字未満の場合，選択した 2 つのメソッド名をつなげたものをクローンセット名とする．
(5) (1) から (4) までのいずれかで，何らかの名前が得られる．しかし，他のクローンセット名と名前の衝突が起こる可能性がある．衝突が起こった場合には，scorpio が出力するクローンセットのハッシュ値をクローンセット名とする．これは，コードクローン検出時に内部で保持するオブジェクトから計算したものであり，分かりやすい名前ではないが，スナップショットで唯一の名前という要求は満足する．

3.6 再構成するディレクトリ構造

図 2(a) に示したようにスナップショットを再構成する．まず，3.5.2 節で決定したクローンセット名を名前とするディレクトリを作成する．そのディレクトリ内にクローンセットに属するコード片をもつメソッドを入れる．そのメソッドは，属するファイルのパスと図 2(b) に示したディレクトリ構造で配置する（/クローンセット名/path/to/java/ファイル名/クラス名/method() となる）．

同一のメソッドが複数のクローンセットに属する場合は，全てのクローンセット内に配置する．コードクローンとならなかったメソッドは，クローンセット名 `none` 内に配置する．これによって非コードクローンの履歴分析も行える．

4. 適用実験

提案手法を Java で書かれたオープンソースのソフトウェアリポジトリに適用する．**表 1** にリポジトリのデータを示

表 1 対象リポジトリデータ
Table 1 Target repository data.

リポジトリ	期間	コミット数	最新版のメソッド数
EMF Compare	5 年	1,141	12,022
Struts 2	6 年 2 ヶ月	1,632	13,822
Browser	3 年 7 ヶ月	4,083	2,363
Camera	3 年 4 ヶ月	3,537	991
Music	3 年 1 ヶ月	672	534
Phone	3 年 4 ヶ月	3,271	1,341

表 2 クローンセット名前付けの結果
Table 2 Results of naming clone sets.

リポジトリ	(1) 一致	(2) 共通	(3) LCS	(4) 結合	(5) ハッシュ
EMF Compare	22	18	1	2	4
Struts 2	13	75	1	5	4
Browser	2	5	0	0	1
Camera	2	5	1	0	1
Music	1	4	0	2	0
Phone	1	4	1	0	0

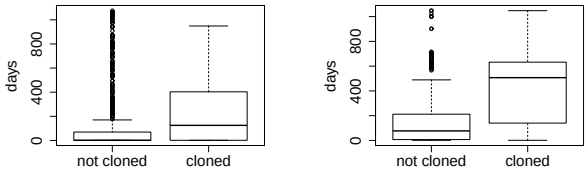
す．Eclipse から 1 つ（EMF Compare[*2] ），Apache から 1 つ (Struts 2[*3])，Android から 4 つ (Browser[*4] ，Camera[*5] ，Music[*6] ，Phone[*7])，計 6 つのリポジトリを対象にそれぞれ CloneHistorage を構築する．開発履歴は 3 年から 6 年ほど蓄積されている．Android から取得したリポジトリは，最新版のメソッド数が 500 から 2,000 程度の小規模なものであり，Eclipse と Apache から取得したリポジトリは，最新版のメソッド数が 12,000 以上のやや規模の大きなものである．

3.2 節で示した要求について，全てのリビジョンでコードクローンを特定し，保存しているので要求 (1) と (2) を満たす．また，要求 (3)：履歴の追跡については，Historage でメソッドレベルのモジュールを追跡可能であることを確認している [3]．そこで以降では，クローンセットの履歴追跡と，要求 (4)：開発プロセスや人的属性に関する情報の保持について確認する．

4.1 クローンセットの履歴追跡

クローンセットの履歴は，CloneHistorage においてディレクトリの履歴に対応させている．このディレクトリ名，すなわちクローンセット名はコードクローン関係にあるメソッド名を参考に決定する．3.5.2 節に示す (1) から (5) の順にクローンセット名を決定するが，番号が若いほど元のメソッド名との一致度合いが高く分かりやすい名前といえる．特に (5) のハッシュ値は，わかりにくいだけでなく，スナップショット間でも異なる値となり履歴の追跡が難しくなるため好ましくない．

実際にどのように名前付けが行えたかを調査する．**表 2** は，最新版にあるクローンセットに対する名前付け結果の分類である．多くのクローンセットには，手順 (1) と (2) で得られる人目にも分かりやすい名前が付けられている．一方，好ましくない名前である (5) のハッシュ値はわずかしかないことが確認できる．以上より，提案するクローンセットの名前付け手法は多くのクローンセットに好ましい名前を付けられたと判断できる．この評価から，多くのクローンセットの履歴が容易に追跡可能であることを確認した．ハッシュ値の名前を持つクローンセットの履歴も容易ではないが可能である（クローンセットに含まれるメソッドの履歴を追跡して，集合の履歴をまとめる）．



(a) Camera (b) Music
図 4 クローン有無とメソッドの存在日数
Fig. 4 Method existing periods with/without clones

4.2 開発プロセスや人的属性のマイニング

CloneHistorage を用いて，コードクローンに関するリポジトリマイニングを行う．コードクローンとなるメソッドの履歴とクローンセットの履歴を分析する．構築した CloneHistorage は，Git リポジトリであるのでこれまでのリポジトリマイニングと同様の分析ができる．

まず，開発プロセスメトリクスの一つである，モジュールの存在日数を計測した．**図 4** に，コードクローンを含むメソッドと含まないメソッドの存在日数をボックスプロットにまとめた（(a) Camera と (b) Music の結果）．この結果からは，クローン関係にあるメソッドの方がないものと比べて長期間存在していることが確認できる．今回の分析では，長期間存在する安定したクローンセットが多かったとわかる．メソッドが，クローンセットでないディレクトリ none からクローンセットのディレクトリに移動した場合は，履歴の途中でコードクローンとなったことを意味する．本稿の分析では，ほとんどのメソッドが作成時にコードクローンとなったのに対して，途中からコードクローン

[*2] http://git.eclipse.org/c/emfcompare/org.eclipse.emf.compare.git
[*3] https://github.com/apache/struts2
[*4] https://github.com/android/platform_packages_apps_browser
[*5] https://github.com/android/platform_packages_apps_camera
[*6] https://github.com/android/platform_packages_apps_music
[*7] https://github.com/android/platform_packages_apps_phone

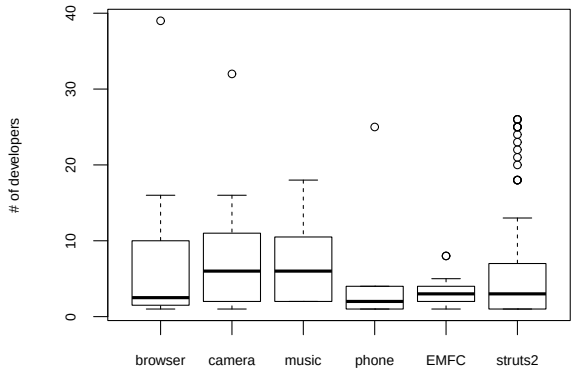


図 5 クローンセットのメソッドを変更した開発者数
Fig. 5 Number of developers who changed methods in clone sets

となったものも一部あることを確認した．

次に，人的属性として開発者数を計測した．**図 5** は，ソフトウェアごとにクローンセットを変更した開発者数をまとめたボックスプロットである．Camera や Music は，多くの開発者に変更されているのに対して，Phone のクローンセットを変更した開発者数は少ないことが確認できる．我々の知る限り，コードクローンの履歴に関して人的属性に関する分析を行ったのは，これが初めてである．

こうした多様な分析は，CloneHistorage を用いるからこそ容易にできる．

5. 議論

5.1 クローンセットの名前衝突

本稿では，クローンセットの名前付けが重複した場合，ハッシュ値を用いることで唯一の名前となるよう実装した．しかし，この名前は履歴を分析する場合にわかりやすくない．今後の課題として，より適切な名前付けが課題となる．

クローンセットの名前衝突が起こるのは，以下の 2 つの場合がある．

- 無関係のクローンセットに，偶然同じ名前付けを行った．
- クローン検出では別々のクローンセットとされたが，人が見た場合には一つのクローンセットと判断できる．

前者の場合に対応するためには，3.5.2 節に示した名前付け手順の工夫が必要である．一方，後者の場合は同一のクローンセットとして扱った方が好ましいと思われる．コードクローン検出時の閾値調整や他のコードクローン検出ツールの再実行などを行い，クローンセットの再評価を行うことが必要と考える．どんなクローンセットの履歴情報が必要かという要求にも依存する問題である．

5.2 機能

コードクローン関係にあるメソッドは，Git 上で通常のファイルと同様に管理されているので，Git に備わった履歴出力に関する全てのコマンドが利用できる．

一方，クローンセットの履歴はディレクトリの履歴に対応させている．Git 上でディレクトリの履歴は，内包するファイルに対する変更の履歴として得られる．これは，クローンセットの履歴として期待する出力である．しかし，ファイルは名前変更後も追跡可能であるのに対して，ディレクトリの名前変更に対して Git は対応していない．これは，版管理システムが各ファイルの版管理を対象としており，ディレクトリの版管理を対象としていないからである．そのため，クローンセットに名前変更が起こった場合の履歴出力には Git 備え付けのコマンドだけでは十分でない．4.2 節の適用実験では，名前変更があった場合は変更前の追跡をしていない．ディレクトリの名前変更や移動検出に対応 [4] したコマンドの実装を行うことは今後の課題である．

5.3 スケーラビリティ

CloneHistorage の構築は，全コミット時のスナップショットを再構成するため，コミット数とファイル数が多いほど時間を必要とする．適用実験では CloneHistorage の構築に，最も規模の小さい Music で 6 時間，最も規模の大きい Struts 2 で 4 日掛かった（Xeon 2×2.26GHz プロセッサ，16GB メモリの Mac Pro で実行）．CloneHistorage は一度構築すれば以降容易にリポジトリマイニングが可能になる．しかしその構築のスケーラビリティは高くない．これは，現状全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からである．各コミットでは一部のファイルしか変更しないことが多いため，変更したファイルだけを追加で処理するインクリメンタルなコードクローン検出 [5] を行えば実行時間を大幅に減らせる可能性がある．

6. おわりに

コードクローン履歴の管理と分析は，ソフトウェア進化に対する知見の発見と保守性向上の面から重要な課題であり，近年注目されている．一般的な開発履歴のデータマイニングは，ソフトウェア工学の多くの研究テーマで有用な知見を報告しており，コードクローンの履歴に対しても期待される．しかし，コードクローンの履歴分析は，コード片とクローンセット（コードクローン関係にあるコード片の集合）履歴の追跡や，データベースの構築などが必要となる難しい課題である．この難しさから，現状のコードクローン履歴の研究は，版管理システム CVS を分析対象としていた初期のリポジトリマイニングの研究フェーズに似ている．リポジトリマイニングの研究は，版管理システム

の発展（Subversion や Git）からより簡単に履歴分析ができるようになり，コード履歴だけでなく，プロセスや人的属性などの多様な面からのデータマイニングの研究が行われ，多くの有用な知見が報告されてきた．

本稿では，コードクローン履歴に対して，これまでのリポジトリマイニングと同様に多様な面からの分析を可能にするため，リポジトリマイニング可能なコードクローン版管理システム CloneHistorage を提案した．Eclipse，Apache，Android から選択した 6 つのオープンソースソフトウェアに対して適用し，評価とリポジトリマイニングを行った．今後の課題には，CloneHistorage 構築のスケーラビリティ向上，履歴分析を容易にするコマンドの実装，クローンセットの名前付け方法の改善などがある．

謝辞 本研究は，日本学術振興会科学技術研究費補助金特別研究員奨励費（課題番号：23・4335），日本学術振興会科学研究費補助金基盤研究 (A)(課題番号：21240002)，萌芽研究 (課題番号：23650014，24650011)，若手研究 (A)(課題番号：24680002) の助成を得た．

参考文献

[1] Duala-Ekoko, E. and Robillard, M. P.: Tracking Code Clones in Evolving Software, *Proc. of 29th Int. Conf. on Softw. Eng.*, ICSE '07, pp. 158–167, (2007).

[2] Göde, N. and Koschke, R.: Frequency and risks of changes to clones, *Proc. of 33rd Int. Conf. on Softw. Eng.*, ICSE '11, pp. 311–320, (2011).

[3] Hata, H., Mizuno, O. and Kikuno, T.: Historage: Fine-Grained Version Control System for Java, *Proc. of 3rd Joint Int. and Annual ERCIM Workshops on Principles of Softw. Evolution and Softw. Evolution Workshops*, IWPSE-EVOL '11, pp. 96–100, (2011).

[4] Hata, H., Mizuno, O. and Kikuno, T.: Inferring Restructuring Operations on Logical Structure of Java Source Code, *Proc. of 3rd Int. Workshop on Empirical Softw. Eng. in Practice*, IWESEP '11, pp. 17–22, (2011).

[5] Higo, Y., Yasushi, U., Nishino, M. and Kusumoto, S.: Incremental Code Clone Detection: A PDG-based Approach, *Proc. of 18th Work. Conf. on Reverse Eng.*, WCRE '11, pp. 3–12, (2011).

[6] Higo, Y. and Kusumoto, S.: Code Clone Detection on Specialized PDGs with Heuristics, *Proc. of 15th European Conf. on Softw. Maintenance and Reengineering*, CSMR '11, pp. 75–84, (2011).

[7] Juergens, E., Deissenboeck, F., Hummel, B. and Wagner, S.: Do code clones matter?, *Proc. of 31st Int. Conf. on Softw. Eng.*, ICSE '09, pp. 485–495, (2009).

[8] Kapser, C. J. and Godfrey, M. W.: "Cloning considered harmful" considered harmful: patterns of cloning in software, *Empirical Softw. Eng.*, Vol. 13, pp. 645–692, (2008).

[9] Kim, M., Sazawal, V., Notkin, D. and Murphy, G.: An empirical study of code clone genealogies, *Proc. of 5th Joint Meeting of the European Softw. Eng. Conf. and the ACM SIGSOFT Symp. on the Found. of Softw. Eng.*, ESEC/FSE-13, pp. 187–196, (2005).

[10] MacDonell, S., Shepperd, M., Kitchenham, B. and Mendes, E.: How Reliable Are Systematic Reviews in Empirical Software Engineering?, *IEEE Trans. Softw. Eng.*, Vol. 36, pp. 676–687, (2010).

[11] Nguyen, H. A., Nguyen, T. T., Pham, N. H., Al-Kofahi, J. and Nguyen, T. N.: Clone Management for Evolving Software, *IEEE Trans. Softw. Eng.*, Vol. 99, No. PrePrints, (2011).

[12] Pate, J. R., Tairas, R. and Kraft, N. A.: Clone evolution: a systematic review, *Journal of Software Maintenance and Evolution: Research and Practice*, pp. 1–23, (2011).

[13] Rahman, F., Bird, C. and Devanbu, P.: Clones: What is that smell?, *Proc. of 7th IEEE Work. Conf. on Mining Softw. Repositories*, MSR '10, pp. 72–81, (2010).

[14] Sridhara, G., Hill, E., Muppaneni, D., Pollock, L. and Vijay-Shanker, K.: Towards automatically generating summary comments for Java methods, *Proc. of 25th IEEE/ACM Int. Conf. on Automated Softw.*, ASE '10, pp. 43–52, (2010).

[15] Thummalapenta, S., Cerulo, L., Aversano, L. and Di Penta, M.: An empirical study on the maintenance of source code clones, *Empirical Softw. Eng.*, Vol. 15, No. 1, pp. 1–34, (2010).

[16] Zibran, M. F. and Roy, C. K.: The Road to Software Clone Management: A Survey, Technical report, Department of Computer Science, The University of Saskatchewan, Canada (2012).

[17] 畑　秀明，水野　修，菊野　亨：リポジトリ再構築によるメソッドトレーサビリティの実現，ソフトウェアエンジニアリングシンポジウム 2010 (SES2010)，情報処理学会，pp. 57–62 (2010).

[18] 畑　秀明，水野　修，菊野　亨：不具合予測に関するメトリクスについての研究論文の系統的レビュー，コンピュータソフトウェア，Vol. 29, No. 1, pp. 106–117 (2012).

[19] 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. 29–42, (2011).

[20] 肥後芳樹，楠本真二：実規模ソフトウェアへの適用を目的としたプログラム依存グラフに基づくコードクローン検出法，ソフトウェアエンジニアリング最前線 2008 (2008).

[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 (ソフトウェア工学)，電子情報通信学会論文誌. D, 情報・システム，Vol. 91, No. 6, pp. 1465–1481, (2008).

[22] 阿萬裕久，野中　誠，水野　修：ソフトウェアメトリクスとデータ分析の基礎，コンピュータソフトウェア，Vol. 28, No. 3, pp. 12–28 (2011).

[23] 小林隆志，林　晋平：データマイニング技術を応用したソフトウェア構築・保守支援の研究動向，コンピュータソフトウェア，Vol. 27, No. 3, pp. 13–23 (2010).