コードクローン変更過程における開発者のインタラクションとソフトウェア品質の関係

久木田 雄亮
和歌山大学大学院 システム工学研究科
kukita.yusuke@g.wakayama-u.jp

大平 雅雄
和歌山大学 システム工学部
masao@sys.wakayama-u.ac.jp

要旨

本稿では，コードクローン追跡ツール *CCT (Code Clone Tracer)* を用いて，コードクローンの変更に関与した開発者らとソフトウェア品質との関係について分析する．*CCT* は，コードクローンの作成・利用過程における人的影響を調査することを意図して設計されたコードクローン追跡ツールである．分散型バージョン管理システム *Git* を対象としたコードクローン追跡機能に加え，不具合管理システムと連携することで不具合を混入したコミットとその作成者を特定し分析するための機能を備えている．本稿では，*3* つのオープンソース開発プロジェクト（*RxJava, c:geo, Jackson databind*）を対象として行った社会ネットワーク分析の結果について報告する．

1. はじめに

長きに渡るソフトウェアシステムの改良・保守の過程では，生産性の向上を目的として既存コードが頻繁に再利用されることがある．その結果,「同一の」または「類似する」コード片（本稿ではコードクローンあるいは単にクローンと呼ぶ）がソースコード全体に遍在することも少なくない．ソースコード全体にコードクローンが多数存在する状況においてコードクローンに変更を加える必要性が生じた場合，変更箇所が多岐に渡ることから多くコストが必要となったり，変更漏れにより新たな不具合を混入してしまう恐れがある．すなわち，改良・保守の初期段階では有益な手段であった既存コードの再利用が，結果的にその後の生産性や品質を低下させる原因となる可能性がある．

このような懸念を払拭するために，コードクローンの**検出手法** [1, 5, 10] や**分析手法** [6, 7, 9] がこれまで盛んに研究されてきた．これら既存研究の多くは主に，ある時点のソフトウェアシステムに含まれるコードクローンとソフトウェアシステムの品質，あるいは，コードクローンの変更がソフトウェアシステムの品質に与える影響に着目している．コードクローンの存在により，あるソースコードに対する変更が他のソースコードに対してどのように影響するかを把握することが困難になるため，結果として品質が低下する可能性が高いと考えられていたためである．しかしながら，これまでの膨大な量の研究を通じて，コードクローンの存在と品質に明確な関係が認められる事例とそうでない事例が混在して報告されており，未だ統一的なコンセンサスは得られていない [15]．

そのため，特に近年では，ソフトウェアシステムの改良・保守の過程でコードクローンがどのように拡散していくのかを詳細に分析するための**クローン追跡**手法 [4, 8, 14] に注目が集まっている．ソフトウェアシステムの開発において広く利用されているバージョン管理システムに記録された全変更履歴に基づいて，リビジョンを 1 つずつ辿りながらコードクローンの変更により不具合が混入される過程を正しく理解し，不具合の混入を未然に防ぐための対策につながると期待されている．

本稿では，我々[12, 16] が提案したコードクローン追跡ツール CCT (Code Clone Tracer) を用いて，コードクローンの変更に関与した開発者らとソフトウェア品質との関係について分析を行い，プロジェクトにおいてコードクローンを作成することは悪影響を与えるものであるかをコードクローンに関係する Issue や開発者間の繋がりから確認する．CCT は，コードクローンの作成・利用過程における人的影響を調査することを意図して設計さ

れたコードクローン追跡ツールである．分散型バージョン管理システム Git を対象としたコードクローン追跡機能に加え，不具合管理システムと連携することで不具合を混入したコミットとその作成者を特定し分析するための機能を備えている．本稿では，3 つのオープンソース開発プロジェクト (RxJava, c:geo, Jackson databind) を対象として行った社会ネットワーク分析の結果について報告する．

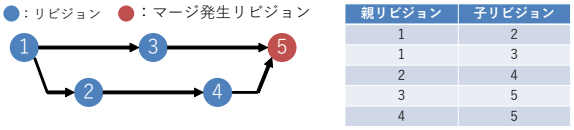
本論文の構成は以下の通りである．続く 2 章では，まず，コードクローン追跡ツール CCT (Code Clone Tracer) の実装について詳述する．3 章では，CCT の処理結果を用いて，コードクローン変更過程における開発者のインタラクションとソフトウェア品質との関係を調査するための分析方法について述べる．4 章では，3 つのオープンソース開発プロジェクト (RxJava, c:geo, Jackson databind) を対象として行った社会ネットワーク分析の結果を示し，分析結果を考察する．最後に 5 章において，本稿をまとめるとともに本研究の今後の課題を示す．

2　コードクローン追跡ツール CCT

本章では，コードクローンの作成・利用過程における人的影響を調査するため実装したコードクローン追跡ツール CCT (Code Clone Tracer) について述べる．CCT は，数千から数万リビジョンからなるソフトウェアシステムに存在するコードクローンの追跡を想定しており，高速にクローンを特定することができる粗粒度なクローン特定方法 [13] を用いてクローンを追跡する [1]．また，不具合管理システムと連携することで不具合を混入したコミットとその作成者を特定する機能を備えている．以降では，コードクローン追跡機能とコードクローン分析機能のそれぞれについて詳細に説明する．

2.1　コードクローン追跡機能

分散バージョン管理システム Git を対象としたコードクローン追跡を実現するために，リポジトリに保存されているすべてのリビジョンを対象に，(1) リビジョンの親子関係の特定，(2) 各リビジョンに含まれる全コード片の特定，(3) コードクローンの特定，(4) コードクローンの追跡の 4 種類の処理を順に行う．以下ではそれぞれの処理について説明する．



親リビジョン	子リビジョン
1	2
1	3
2	4
3	5
4	5

図 1: リビジョンの親子関係の特定

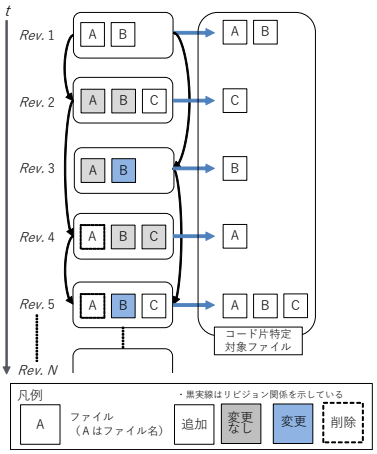


図 2: コード片特定対象ファイルの決定（リビジョンの親子関係は図 1 と同じ）

(1) Git におけるリビジョンの親子関係の特定：CCT は分散型バージョン管理システム Git を対象とするため，従来の集中型バージョン管理システムを対処としたクローン追跡 [4, 8, 14] のように時系列でリビジョンを 1 つ 1 つ順に辿る方法は適用できない．そのため，以下の手順でリビジョンの親子関係を特定する（図 1）．まず，分析対象リポジトリに含まれるリビジョンの情報を取得する．次に，リビジョン毎に親リビジョンと子リビジョンを特定し（図 1 左），各リビジョンの親子関係を記録する（図 1 右表）．リビジョンの親子関係を整理しておくことで，図 1 の *Rev.*1 や *Rev.*5 のように，複数の親リビジョンや子リビジョンを持つリビジョンが参照可能になる．

(2) 各リビジョンに含まれる全コード片の特定：ファイル変更情報を基にして各リビジョンのコード片特定対象ファイルを決定し（図 2），[13] と同様にブロック単位でコード片を特定する（図 3）．

まず，分析対象リポジトリのコミット履歴から，リビジョン番号，コミット日時，コミッタ名，追加・変更・削

[1] 実際，クローンの特定までの処理（コード片抽出，正規化，ハッシュ化）は，CRD (Clone Region Descriptors)[4] ベースのクローン追跡ツール ECTEC (Enhancement of CRD-Based Clone Tracker for Evolution of Clones) [8] の一部を再利用して実装している．

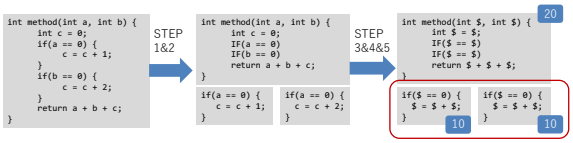


図 3: ブロックの検出・正規化・ハッシュ化の流れ（[13] より抜粋）

除されたファイルについての情報（コミット情報と呼ぶ）を取得する．次に，分析対象リポジトリの最初のリビジョン（リポジトリに初めてコミットされたファイル群から構成されるリビジョン）を調査し，最初のリビジョンを構成する全ファイルを特定する（図 2 における *Rev.*1）．コミット情報を用いて，各リビジョンに加えられたすべての変更（ファイルの追加・変更・削除）を，最初のリビジョンを起点として，リビジョンの親子関係に従って順次遡り最新のリビジョンまでコード片特定対象ファイルを決定する．マージコミット（図 2 における *Rev.*5）では，変更の衝突が発生する場合がある．衝突がない場合，マージ元の 2 つのコミットとの差分情報を取得し，変更が加えられたファイルをコード片特定対象ファイルとする．衝突が発生していた場合，衝突を解消するまでの履歴が抜け落ちていることがあるため，そのリビジョンに存在するファイル情報を基にしてすべてのファイルをコード片特定対象ファイルとする．なお，ソースコード以外のファイルは分析対象ではないため除外する．

コード片特定対象ファイルに対して行うコード片特定処理について以下で説明する．まず，最初のリビジョンを構成する全ファイルに対してブロック単位でコード片を特定する．図 3 の STEP1 でソースコードに対して字句解析，構文解析を行い，STEP2 でブロックの種類，条件式，シグネチャ，クラス名を特定する．ブロックとは，クラスやメソッド，for 文や if 文などのブロック文を指す．ブロック単位でコード片を特定した後，最初のリビジョンを起点として以降のリビジョンを構成するファイルについてもコード片を順次特定する．このとき，最初のリビジョン以降のリビジョンでは，ファイルの変更情報に基づいて，変更が加えられたファイル（図 3 のコード片特定対象ファイル）のみに対してコード片を特定し，変更がなかったファイルのコード片は前のリビジョンのコード片情報を再利用する．

(3) コードクローンの特定：前述の前処理により，すべてのリビジョンに対してブロック単位ですべてのコード片が特定される．コードクローンの特定処理においては，まず，[13] と同様の方法（図 3 の STEP3&4&5）で，特定されたブロックを定められたフォーマットに整形する．STEP3 において，ブロックに含まれる変数名とリテラルは特殊文字にすることで正規化し，変数名やリテラルのみが異なるようなクローンを特定できるようにする．さらに，STEP4 において，正規化されたブロックの文字列をもとにハッシュ値を算出する．そして，STEP5 において，ハッシュ値の一致するブロックをクローンとして特定する．このコードクローン特定処理を前述の前処理によって特定できた全てのリビジョンのコード片に対して行うことで，各リビジョンに存在するコードクローンを特定できる．

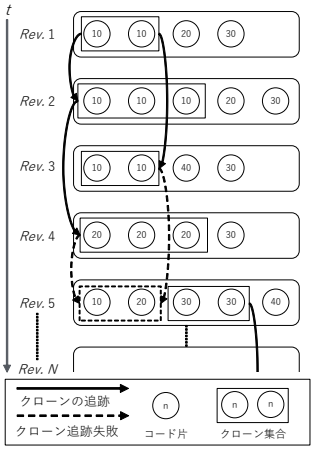


図 4: コードクローンの追跡（リビジョンの親子関係は図 1 と同じ）

(4) コードクローンの追跡：コードクローンの追跡処理では，最初のリビジョンに含まれるすべてのコード片のハッシュ値を比較し，同じハッシュ値を持つコード片の集合をクローン集合としてまとめ，以降のリビジョンでのクローン集合の前後関係を特定できるようにする．具体的には，以下の方法でクローン集合の前後関係を特定しクローン集合をリビジョン間で追跡する（図 4）．

まず，次の子リビジョンに存在しているクローン集合に含まれるコード片のハッシュ値を調べ，直前のリビジョンのクローンのハッシュ値と比較する．ハッシュ値が一致した場合は，前のリビジョンに存在するクローン集合と同じクローン集合であるので要素数を前後のリビジョンで比較する．要素数の増加によるものか，ハッシュ値が変化しない変数名やリテラルの変更によるものかを判

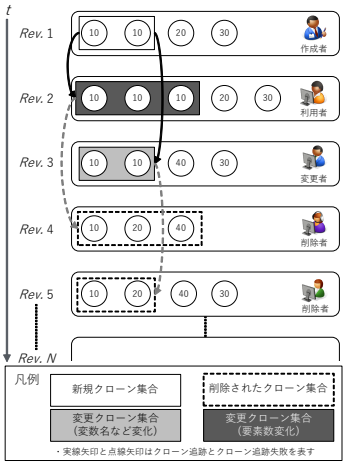


図 5: コードクローンの追跡と開発者の分類（リビジョンの親子関係は図 1 と同じ）

別後，前のリビジョンのクローン集合を親集合，次のリビジョンのクローン集合を子集合と解釈し，クローン集合間にリンクを張る．ハッシュ値だけの比較では追跡漏れが起きてしまうため，ハッシュ値の比較でリンクを張ることができなかったクローン集合に対して [8] で用いられている手法と同様に CRD の類似度比較（CRD はコード片のブロックの種類やソースコードにおける位置情報を含む表記形式で，この文字列が類似するかを比較）によってクローン集合間にリンクを張る．リンクが張られることがなかったクローン集合は，新たなクローン集合が作成されたものと解釈し，該当クローン集合を以降のリビジョンでの追跡対象とする．最後に，前のリビジョンにのみ存在するクローン集合は削除されたものと解釈して追跡対象から除外する．

従来の追跡ツールである ECTEC[8] では全てのコード片に対してリンクを張ることによって，コードクローンの追跡を行っていた．本ツールではコードクローンのみにリンクを張ることによってコードクローンの追跡を実現しているので，コードクローンとは関係のないコード片の追跡判定処理をする必要がなくなり，コードクローンの追跡にかかる処理時間を削減できる．

2.2 コードクローン分析機能

CCT はコードクローン追跡機能に加えて，コードクローンの作成・利用に関与した開発者とコードクローンの品質との関係を時間を遡って分析するための機能を以下の処理によって実現している．

(1) 開発者の特定処理：クローンの作成・利用過程に関与した開発者を作成者（オリジナルのクローンを作成した開発者）・利用者（既存のクローンを他の場所に移植した開発者）・変更者（ハッシュ値の変更を伴わないクローンに対する変更をおこなった開発者）・削除者（ファイルの削除などによりクローンを削除した開発者）の 4 種類に分けて特定し，リビジョンと紐づける（図 5）．

(2) 不具合混入リビジョンとファイルの特定処理：これまで紹介した機能はすべて Git リポジトリから得られる情報に基づいた処理であり，クローンの作成・利用過程に関与した開発者が混入した不具合に関する情報は別途，不具合管理システム（Bugzilla や Jira など）から取得する必要がある．本ツールには，SSZ アルゴリズム [11] を用いて不具合混入コミットを特定し，不具合を混入した開発者を分析する機能を実装してる．

(3) コードクローンと不具合紐づけ処理：本ツールで行ったコードクローン追跡処理と不具合混入リビジョンとファイルの特定処理の結果から得られる開発者と不具合の関係を用いて，コードクローンと不具合を紐づけることによって，クローン・人・不具合の紐付けできる．

3 分析：開発者のインタラクションとソフトウェア品質

本章では，コードクローン変更過程における開発者のインタラクションとソフトウェア品質の関係を調べるために CCT を用いて行う分析と，分析で用いたデータセットについて述べる．

3.1 目的

ファイルの作成・変更に関与する開発者と品質を分析した研究 [3, 2] から，開発者のインタラクションとソフトウェア品質には密接な関係があることが知られている．例えば文献 [3] では，安定して成長している OSS 開発プロジェクトでは潜在的に密なコミュニティが存在しており，そのコミュニティに属する開発者同士が同じファイルの変更に関与していることを明らかにしている．すなわち，特定の開発者からなるグループでの密なインタラクションがソフトウェア品質に寄与することを示唆するものである．一方，文献 [2] では，一人の開発者によって変更されてきたファイルよりも多数の開発者によって変

更が加えられてきたファイルの方が欠陥が含まれやすいことを明らかにしている．文献 [3] とは反対の結果ではなく，開発者の疎なインタラクションを通じて同一ファイルに対する変更が行われるとソフトウェア品質の低下を招くということを示唆したものである．

これらの研究成果は，ファイル単位での分析によりファイルの作成・変更に関与した開発者とソフトウェア品質の関係を示したものであった．そこで本稿では，コードクローン単位の分析でも同様の結果を導くことができるのかどうかを確かめることを目的とする．コードクローンとソフトウェア品質の関係については多数の研究が存在しているが，コードクローンがソフトウェア品質にとって有害である場合とそうでない場合が混在しており，統一的なコンセンサスは未だ得られていない．さらに，コードクローン変更過程における開発者とソフトウェア品質の関係について調べた研究はまだ多くないため，既存研究では示されていないコードクローンの長期的なメンテナンスのための新たな知見を提供できる可能性がある．

3.2 分析方法

本稿では，コードクローン変更過程における開発者のインタラクションを Pajek[2] により可視化し分析する．Pajek とは人と人，人とモノの関係などをネットワークとして表現し分析するためのネットワーク分析ツールの 1 つである．分析では，課題管理システムに報告された課題 (Issue) に関係しているコードクローン（Issue 混入クローン）と Issue 解決のためにコードクローンの変更に関与した開発者との関係を Pajek により可視化する．また，対比のため，Issue の解決とは全く関係のないコードクローン（Issue 未混入クローン）とその変更に関与した開発者との関係を可視化する．すなわち，品質に問題のあるコードクローンの変更に関与した開発者のインタラクションと，品質に問題のないコードクローンの変更に関与した開発者のインタラクションとを比較する．具体的には以下の手順で分析を行う．

(1) Issue 混入クローンと未混入クローンの分別：まず初めに，CCT の処理結果から得られる情報を基に，Issue と紐づいた Issue 混入クローンと，Issue と紐づいていない Issue 未混入クローンとに分ける．

(2) 開発者の特定：次に，各コードクローンを変更（追加，修正，削除）した開発者の情報を CCT の処理結果から取り出す．

(3) ネットワーク情報の作成：(1), (2) の情報を用いてコードクローンとコードクローンの変更に関与した開発者を紐付け，Pajek 形式のネットワークデータを作成する．

(4) コードクローンと開発者の関係の可視化：(3) により Pajek を用いてコードクローンと開発者の関係を無向グラフとして可視化する．

(5) 開発者のインタラクションの可視化：さらに，(4) から同一クローンの変更に関与した開発者のみを紐づけることにより開発者のインタラクションを無向グラフとして可視化する．

3.3 データセット

本稿では，GitHub[3] で活発に活動を行なっている 3 つのオープンソース開発プロジェクト（RxJava, c:geo, Jackson databind）を対象として分析を行う．RxJava はリアクティブプログラミングを支援するライブラリである．c:geo は Android 向けの GPS 機能を使ったゲームのアプリケーションである．Jackson databind は JSON 形式のデータを扱うことを支援するライブラリである．表 1 に各プロジェクトの基本統計量を示す．

表 1 中の「Issue 混入 CC 変更」,「Issue 未混入 CC 変更」とはそれぞれ，Issue 混入クローンの変更に関与した開発者の数と Issue 未混入クローンの変更に関与（Issue とは関連のない理由でコードクローンを変更）した開発者の数を指す．プロジェクト間でばらつきがあるものの，プロジェクトの開発者の約 1 割〜3 割程度がコードクローンの変更に関与していることが見て取れる．なお，Github では Issue の種類をプロジェクト毎に自由に定義するタグで管理する仕様になっておりプロジェクト間に共通のタグが存在しないため，本データセットにおける Issue には不具合に関するものから機能拡張（エンハンスメント）に関するものまで含まれている．特定の種類の課題のみを指すものではないことに注意されたい．Issue の種類を厳密に区別するためにはすべての Issue を目視する必要があるため，本データセットでは区別しないこととしたが，分析結果に影響を与える可能性があり本研究の今後の課題とする．

[2] Pajek, http://vlado.fmf.uni-lj.si/pub%20/networks/pajek/

[3] https://www.github.com/

表 1: 対象プロジェクトの統計量

プロジェクト名		RxJava	c:geo	Jackson databind
対象期間		2014/08/30 ∼2016/06/29	2011/07/11 ∼2016/07/21	2011/12/23 ∼2016/07/21
リビジョン数		1,397	9,821	3,260
開発者数（人）(∗)	開発者総数	73	107	114
	Issue 混入 CC 変更者	18	28	17
	Issue 未混入 CC 変更者	17	27	11
コードクローン（セット数）	CC 合計	289	2,660	1,610
	Issue 混入 CC	104	2,008	1,223
	Issue 未混入 CC	185	652	387
Issue 数（件）	Issue 合計	1,892	5,849	1,307
	CC 関連有 Issue	35	405	241
	CC 関連無 Issue	1,857	5,444	1,066

(∗) 開発者総数とは，プロジェクトでコードを変更したことのあるすべての開発者の数である．直下の 2 項目（Issue 混入/未混入 CC 変更）はコードクローン (CC) の変更に関与した開発者の数を示すもので，それら 2 項目の合計とはプロジェクト全体の開発者の合計は一致しない．

表 1 中,「Issue 混入 CC」,「Issue 未混入 CC」とはそれぞれ，変更されたコードクローンの内，Issue が混入していたものと未混入だったもの数を意味している．それぞれを足し合わせたものが合計と一致する．なお，コードクローンは，クローンとなっているコード片の総数ではなくクローン関係にあるコード片集合（セット数）を示すものである．コードクローンはリビジョン数の違いを考慮してもプロジェクト間で大きな差異があることがわかる．特に，c:geo では 2,008 セットの Issue 混入クローンが存在しており，コードクローンに対する変更の多くは Issue に関連したものである．Jackson databind も同様に，コードクローンの変更の多くは Issue に関連していることが見て取れる．

表 1 中,「CC 関連有 Issue」,「CC 関連無 Issue」とはそれぞれ，報告された Issue を解決するためにコードクローンの変更を伴ったものとコードクローンの変更は伴わなかったものを意味している．それぞれを足し合わせたものが Issue 合計と一致する．RxJava については，報告された Issue に占める CC 関連有 Issue は少ない (35/1,892) が，c:geo および Jackson databind については比較的多くの Issue がコードクローンの変更に関係していることが見て取れる．

4 分析結果と考察

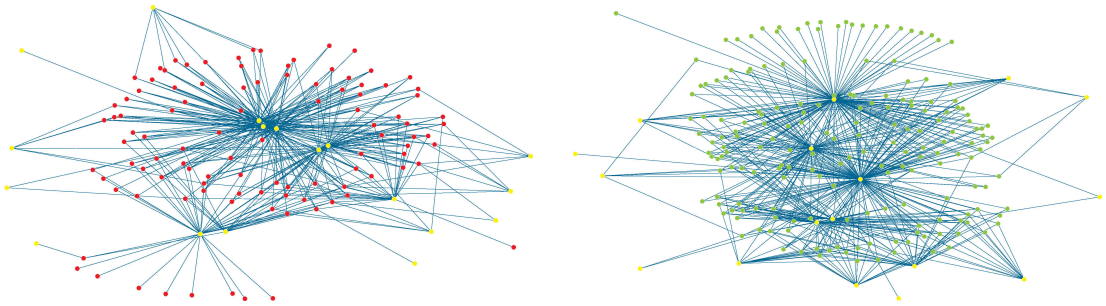
本章では，前章で述べたデータセットに対して Pajek を用いて可視化した開発者のインタラクションをプロジェクト毎に示し結果を考察する．

4.1 概要

Pajek で可視化した開発者のインタラクションを可視化した結果を図 6∼図 11 に示す．すべての可視化結果において，赤色のノードは Issue 混入クローンを，緑色のノードは Issue 未混入クローンを表す．黄色のノードが開発者である．

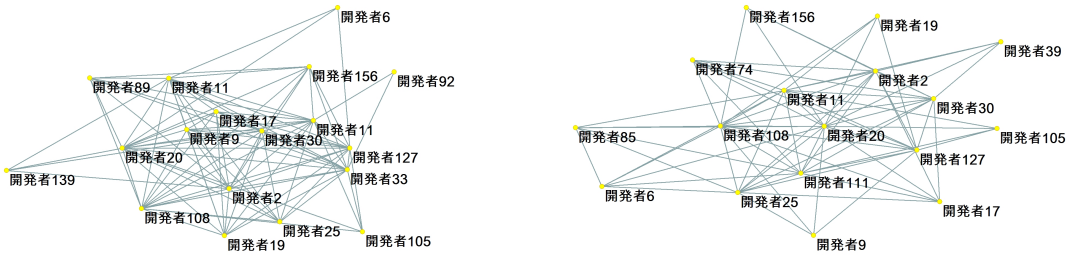
図 6, 8, 10 は，コードクローンを変更した開発者と変更されたコードクローンの間をエッジで結び，コードクローンとコードクローンの変更に関与した開発者の関係（開発者とコードクローンのインタラクション）を，Issue 混入クローンと Issue 未混入クローンとで対比できるように可視化したものである．

図 7, 9, 11 は，図 6, 8, 10 のグラフから，同じコードクローンを変更したことのある開発者間をエッジで結び，コードクローンの変更に関与した開発者同士の関係（開発者間のインタラクション）のみを抽出し可視化したものである．開発者とコードクローンのインタラクションと同様に，Issue 混入クローンと Issue 未混入クローン



(a) Issue 混入（密度：0.603）　　**(b)** Issue 未混入（密度：0.394）

図 6: コードクローンと開発者の関係 (RxJava)



(a) Issue 混入（密度：0.614）　　**(b)** Issue 未混入（密度：0.529）

図 7: 開発者の関係 (RxJava)

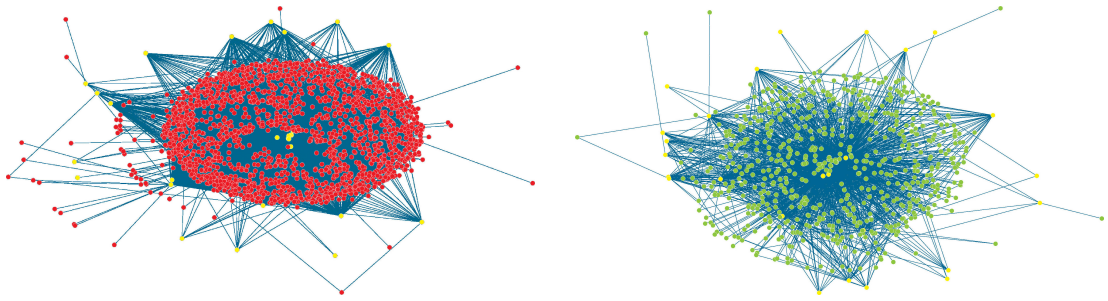
とで対比できるようにしている．開発者間のインタラクションは，開発者同士が同じコードクローンを共同で作成・変更・利用したことを意味する．つまり，図 7, 9, 11 のグラフが密になっているということは，開発者同士が同じコードクローンを共同で作成・変更・利用したことが多いということを示している．各グラフの密度はあるグラフの辺の数とそのグラフの可能な辺の数の最大数で割ったものである．

次節では，これらの可視化結果をプロジェクトごとに考察する．

4.2 RxJava

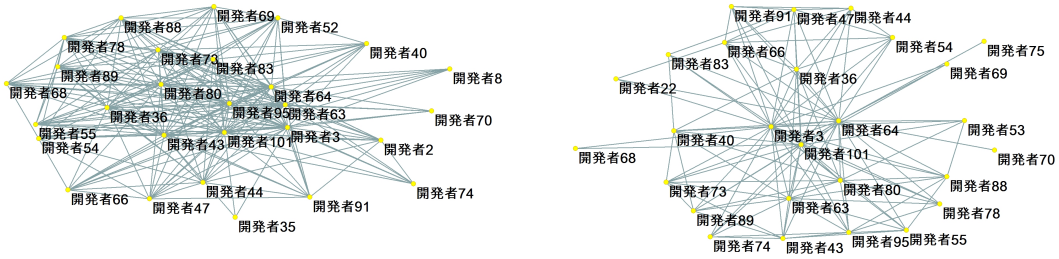
表 1 より，RxJava は他の 2 プロジェクトに比べて変更されたクローンセット数（CC 合計: 289, Issue 混入 CC: 104, Issue 未混入 CC: 185）が少ないことが分かる．図 6 からも変更されたコードクローンの数に違いがあるものの，開発者とコードクローンのインタラクションの基本構造に大きな違いがないことが見て取れる．実際，図 6(a), (b) のどちらも 5, 6 名の開発者が数多くのコードクローンの変更に関与していることが分かる．

一方，図 7 からは Issue 混入クローン数 (104) は Issue 未混入クローン数 (185) よりも少ないにもかかわらず，図 7(a) の方が若干ではあるが密な構造になっていることが分かる．コードクローンは開発担当者の異なる複数のファイルに散在していることが多いが，いずれかのファイル中のコードクローンを変更した場合にはその他のファイル中のコードクローンも変更する必要が生じやすい．特に Issue 混入クローンでは，コードクローンの変更により Issue を混入させる，あるいは，Issue を解決するためにコードクローンを変更する必要があるため，コードクローンを含むファイル間の依存関係が Issue 未混入コードクローンの変更よりも強く現れやすいものと考えられる．結果的に，Issue 混入クローンの変更に関与した開発者同士のインタラクションは，Issue 未混入クローンの変更に関与した開発者同士のインタラクションよりも密になるものと思われる．



(a) Issue 混入（密度：0.891）　　**(b)** Issue 未混入（密度：0.333）

図 8: コードクローンと開発者の関係 (c:geo)



(a) Issue 混入（密度：0.597）　　**(b)** Issue 未混入（密度：0.351）

図 9: 開発者の関係 (c:geo)

4.3 c:geo

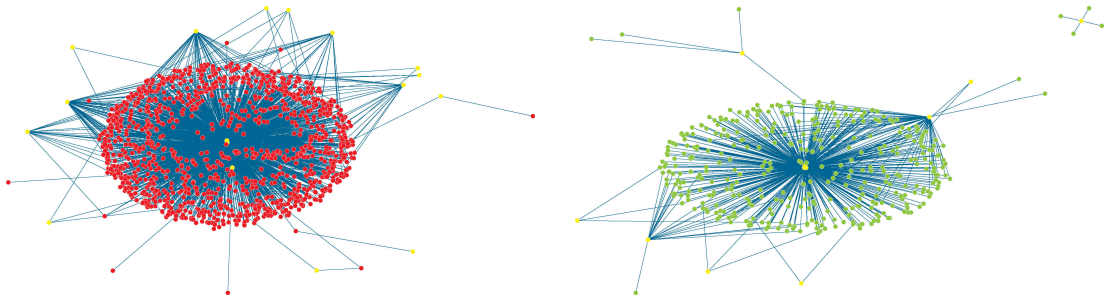
表 1 より，c:geo は他の 2 プロジェクトに比べてリビジョン数が最も多く，変更されたクローンセット数（CC 合計: 2,660, Issue 混入 CC: 2,008, Issue 未混入 CC: 652）が非常に多いことが分かる．一方，コードクローンの変更に関与した開発者自体は十数名多い程度（Issue 混入 CC 変更者: 28, Issue 未混入 CC 変更者:27）であるため，図 8 のグラフではコードクローンのノードとそのエッジがグラフ要素の大半を占める．その結果，開発者とコードクローンのインタラクションの詳細を読み取ることができないが，少なくとも少数の開発者が多数の Issue 混入/未混入クローンの変更に関与していることは見て取れる．RxJava と同様，少数の中心的な開発者がコードクローンの変更において重要な役割を担っていると思われる．

図 9 からは，コードクローンの変更に関与した開発者の数はほぼ同じであるが，Issue 混入クローン数 (2,008) が Issue 未混入クローン数 (652) よりも相当多いため，Issue 混入クローンを変更する場合の方が開発者同士のインタラクションが密になっていることが分かる．Issue 未混入クローンの変更では，Issue 混入クローンの変更とは異なり数名の開発者がより多くの繋がり（エッジ）を持っているが全体としては比較的疎な構造担っていることが分かる．

4.4 Jackson databind

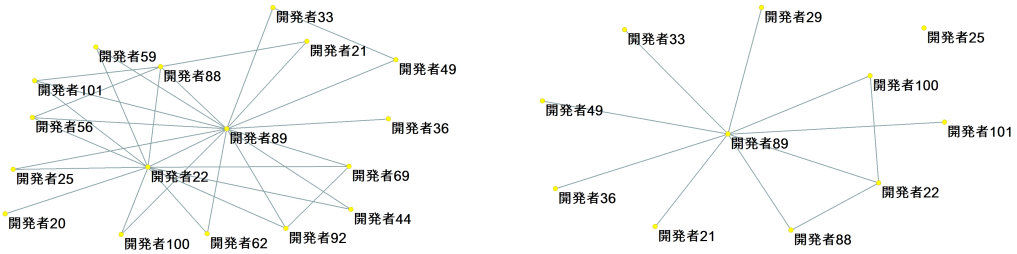
表 1 より，Jackson databind のリビジョン数 (3,260) は c:geo (9,821) の 1/3 程度であることが分かる．一方，Issue 混入 CC および CC 関連有 Issue は 1/2 程度であるため，Jackson databind におけるソースコードの変更は 3 つのプロジェクトの中で最もコードクローンの変更によるものであることが読み取れる．図 10 からは，Issue 混入クローンの変更では 2 名，Issue 混入クローンの変更では 1 名の開発者がほとんどのコードクローンの変更に関与していることが分かる．

図 11 からも，同様のことが見て取れる．Jackson databind では，1,2 名の中心的な開発者が作成したソー



(a) Issue 混入（密度：0.855）　　**(b)** Issue 未混入（密度：1.223）

図 10: コードクローンと開発者の関係 (Jackson databind)



(a) Issue 混入（密度：0.235）　　**(b)** Issue 未混入（密度：0.231）

図 11: 開発者の関係 (Jackson databind)

スコードには多くのコードクローンが含まれており，その他の開発者がそれらコードクローンを利用した結果を表しているものと思われる．Jackson databind においても，Issue 混入クローンの変更の場合の方が 2 名以外の開発者のインタラクションも若干密な構造になっている．Issue 未混入クローンの変更の場合は，1 名の開発者のみが全ての開発者とインタラクションが存在する構造になっており，他の開発者同士の繋がりはほとんど存在しない．

5. まとめと今後の課題

本稿では，コードクローンの編集過程における開発者のインタラクションとソフトウェア品質の関係を理解する事を目的として，我々が提案しているコードクローン追跡ツール CCT (Code Clone Tracer) を用いて，3 つのオープンソース開発プロジェクト（RxJava, c:geo, Jackson databind）を対象とした社会ネットワーク分析を行った．

社会ネットワーク分析ツール Pajek によって可視化した結果，以下の知見が得られた．

- いずれのプロジェクトにおいても少数の開発者のみがコードクローンの編集の大多数に関与しており，Issue の存在の有無に関わらず各プロジェクトにおける開発者とコードクローンのインタラクションの全体構造に大きな違いはなかった（図 6, 8, 10）．
- Issue 混入クローンの変更に関与した開発者同士のインタラクションは比較的密な構造になる（図 7, 9, 11），すなわち，コードクローンに何らかの問題がありそれを解決するためには多くの利害関係者の協力を必要とする．問題のあるコードクローンの編集は多数のソースコードの変更という観点だけではなく，それぞれの担当者との調整や協力という観点からも影響範囲がより大きくなる可能性がある．

本稿では，CCT の処理結果をもとに社会ネットワーク分析を行ったが，まだ基本的な分析を終えた段階であり，より有益な知見を提供するためには今後はさらに分

析を精錬する必要がある．本研究の今後の課題としては以下のようなものが挙げられる．

- Issue の種類を分別する（本稿では不具合なのか機能拡張なのか等を区別していない）．
- Issue の内容を識別するとともに変更されたコードクローンの内容を分析する（本稿では Issue の内容やコードクローンの変更内容を詳細に把握しておらずコードクローンの変更理由は不明なままである）．
- Issue の内容を識別する（本稿では Issue の内容を詳細に把握していないのでコードクローンの変更理由は不明なままである）．
- ファイル単位の従来の分析結果 [3, 2] と比較して，コードクローンの編集過程においてのみ特徴的な現象を抽出する（本稿ではファイル単位での分析は行っていないため，前述の知見はコードクローンの変更に関してのみ観察される事象でない可能性がある）．
- 開発者とコードクローンのインタラクションを時系列に可視化する（本稿では，過去のデータを一度に可視化しているためコンテキストが失われている）．
- 分析対象とする OSS プロジェクトデータを増やす，あるいは，企業のプロジェクトデータを対象にする（本稿では，Github 上の 3 つの OSS プロジェクトを対象にしたのみ）．

謝辞

本研究の一部は，文部科学省科学研究補助金（基盤 (C): 15K00101）による助成を受けた．

参考文献

[1] I.D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proc. of the Int'l Conf. on Software Maintenance (ICSM '98)*, pp. 368–377, 1998.

[2] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu. Don't touch my code!: Examining the effects of ownership on software quality. In *Proc. of the Joint Meeting of the European Software Engineering Conf. and the ACM SIGSOFT Symp. on the Foundations of Software Engineering (ESEC/FSE '11)*, pp. 4–14, 2011.

[3] C. Bird, D. Pattison, R. D'Souza, V. Filkov, and P. Devanbu. Latent social structure in open source projects. In *Proc. of the 16th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering (FSE '08)*, pp. 24–35, 2008.

[4] E. Duala-Ekoko and M.P. Robillard. Clone region descriptors: Representing and tracking duplication in source code. *ACM Trans. on Software Engineering and Methodology (TOSEM)*, Vol. 20, No. 1, pp. 3:1–3:31, 2010.

[5] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Proc. of the IEEE Int'l Conf. on Software Maintenance (ICSM '99)*, pp. 109–118, 1999.

[6] N. Göde and R. Koschke. Frequency and risks of changes to clones. In *Proc. of the 33rd Int'l Conf. on Software Engineering (ICSE '11)*, pp. 311–320, 2011.

[7] J. Harder. How multiple developers affect the evolution of code clones. In *Proc. of the 2013 IEEE Int'l Conf. on Software Maintenance (ICSM '13)*, pp. 30–39, 2013.

[8] Y. Higo, K. Hotta, and S. Kusumoto. Enhancement of crd-based clone tracking. In *Proc. of the 2013 Int'l Workshop on Principles of Software Evolution (IWPSE '13)*, pp. 28–37, 2013.

[9] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *Proc. of the 31st Int'l Conf. on Software Engineering (ICSE '09)*, pp. 485–495, 2009.

[10] 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 (TSE)*, Vol. 28, No. 7, pp. 654–670, 2002.

[11] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *Proc. of the 2005 Int'l Workshop on Mining Software Repositories (MSR '05)*, pp. 1–5, 2005.

[12] 大平雅雄, 久木田雄亮. コードクローンの作成・利用過程における人的影響を調査するための追跡ツールの試作. 情報処理学会 ソフトウェア工学研究会, 第 2016-SE-193 巻, pp. 1–25, 7 2016.

[13] 堀田圭佑, 楊嘉晨, 肥後芳樹, 楠本真二. 粗粒度なコードクローン検出手法の精度に関する調査. 情報処理学会論文誌, Vol. 56, No. 2, pp. 580–592, 2015.

[14] 堀田圭佑, 肥後芳樹, 楠本真二. 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. 情報処理学会論文誌, Vol. 55, No. 2, pp. 947–958, 2013.

[15] 堀田圭佑, 肥後芳樹, 楠本真二. 生成抑止，分析効率化，不具合検出を中心としたコードクローン管理支援技術に関する研究動向. コンピュータソフトウェア, Vol. 31, No. 1, pp. 14–29, 2014.

[16] 久木田雄亮, 大平雅雄. コードクローンの編集過程における開発者の潜在的社会構造に関する分析. 2016 年度情報処理学会関西支部大会 支部大会講演論文集, 第 2016 巻, 9 2016.