Vol.2019-SE-203 No.1
2019/12/12

プロジェクト間クローンに対する変更傾向の調査

石津　卓也[1,a)]　吉田　則裕[2]　崔 恩瀞[3]　徳井 翔梧[1]　井上 克郎[1]

概要：既存のプロジェクトやコードを再利用することは標準化されたシステム開発環境 (エコシステム) を形成し開発の効率化に貢献する．しかしながら, このようなプロジェクト間クローンを変更した場合，再利用ソースコードにおいても同様の変更を検討する必要がある．そこで本研究では，実際の GitHub 上で管理されている C/C++プロジェクト間クローンの変更傾向に関する調査を行った．その結果，一貫した変更は一貫性のない変更に比べて約 1.89 倍存在し，一貫した変更が行われているプロジェクトペアは完全一致していることが明らかになった．一貫した変更に比べて一貫性のない変更のほうが約 1.89 倍存在していることが判明した．プロジェクトの変更理由についても調査し，一貫性のない変更の多くはプロジェクトペアの一方が開発者の開発活動があまり活発ではないために生じていると考察した．また，両方のプロジェクトの開発が活発だった場合，それぞれの機能が独自に拡張されている可能性がある．

キーワード：プロジェクト間コードクローン,OSS, 変更傾向

Change Trends Analysis of Inter-Project Cloning

Takuya Ishizu[1,a)]　Norihiro Yoshida[2]　Eunjong Choi[3]　Shogo Tokui[1]　Katsuro Inoue[1]

1. まえがき

オープンソースソフトウェア (OSS) の開発には高品質なソースコードが開発される．昨今では，多くのソフトウェア開発者が GitHub などのリポジトリ上で公開されている OSS を容易に利用でき，高品質で効率的な開発が可能である．その一方で各々の開発者が既存の OSS を再利用をすることで生じる問題がある．変更された再利用ソースコードを一貫性のない変更が行われることである．例えば，プログラミング言語や依存ソフトウェアの仕様変更，機能拡張，バグの修正のために再利用したソースコードが変更されることがある．このような変更に対して再利用元ソースコードにも一貫した変更を検討することはソフトウェア間の依存関係やバグ修正の観点などから望ましいと考えられるが，一貫した変更が行われるとは限らない．再利用ソースコード間の一貫性のない変更が生じる理由は次の 2 つの理由が考えられる．1 つは再利用ソースコードの変更を検知ができていない可能性があることで，もう 1 つはその検知した変更を反映するべきか否か検討しなければならず開発者にとって負担になっている可能性があることである．開発者は一貫性のある変更が必要になった時，変更に伴う保守作業全体について考慮して一貫した変更を行うべきか否かを検討する必要があるが，必ずしも行われるとは限らない．

しかしながら，この理由を調べるためにはプロジェクト間にまたがる再利用ソースコードの変更に関する実際の傾向について詳細に把握する必要があり，多くの OSS を対象にどのような再利用ソースコードがどの程度変更されているのか，あるいはなぜ一貫性のない変更をしているのかについて調査する必要がある．また，反対に一貫した変更についても同様にどのような再利用ソースコードがどの程度

[1] 大阪大学
Osaka University
[2] 名古屋大学
Nagoya University
[3] 京都工芸繊維大学
Kyoto Institute of Technology
[a)] t-ishizu@ist.osaka-u.ac.jp

変更されているのか調査することはプロジェクト間にまたがる再利用ソースコードの一貫した変更を行う効率的な開発方法を考える上で重要な手掛かりになると考えられる．プロジェクト間の再利用に関する研究はいくつか行われている [1], [6]．これらの研究はともに GitHub 上に存在するプロジェクトにまたがる再利用ソースコードの分布について，コードの類似性に着目して検出するプロジェクト間クローンを検出する手法を用いて調査し，どのようなソースコードが再利用されやすいかについて考察している．

本研究では，Lopes らの研究 [6] のが発見した GitHub 内の再利用ソースコードについて，プロジェクト間で一貫した変更の効率的な開発方法や一貫性のない変更に対する対策を考える目的のためにプロジェクト間クローンの変更傾向について調査した．その結果，918 プロジェクト中 148 プロジェクトのプロジェクト間クローンが変更されており，一貫性のある変更に比べて一貫性のない変更のほうが約 1.89 倍存在していることが判明した．一貫性のある変更は完全一致している，あるいは完全一致に近い類似度を持つプロジェクトペアで起こり，その変更は開発者が必要に応じてリビジョンアップのタイミングで一貫性のある変更が行われていると考えられる．さらに，プロジェクトの変更理由についても調査し，一貫性のない変更の多くはプロジェクトペアの一方が開発者の開発活動があまり活発ではないために生じていることが明らかになった．また，両方のプロジェクトの開発が活発だった場合，それぞれの機能が独自に拡張されている可能性がある．

以降，2 章では変更傾向を行う調査の背景，3 章では変更傾向を調査の目的とその手順，4 章ではその調査結果とそこから得られた考察，5 章ではまとめと今後の課題について述べる．

2. 背景

本章では，再利用ソースコードを調査する目的や手段を説明するにあたって前提となるコードクローンやクローン変更管理システム，プロジェクト間クローンの既存研究について説明する．

2.1 コードクローン

コードクローンとは，プログラムテキスト中の互いに一致または類似したコード片を意味する [9]．また互いに一致または類似したコード片の集合をクローンセットと呼ぶ．コードクローンの存在はソフトウェアの保守作業を困難なものにすると言われている．例えば，あるコード片を変更する場合，そのコードクローンと同じクローンセットに属する他のコードクローンに対しても同様の変更が行われるべきか検討する必要がある．コードクローンにバグ等が含まれていた場合，そのコードクローンとコードクローン関係になっている全てにコード片を見つけて，それらのコード片に対して同一修正を行うかを検討するため，コードクローンはなるべく集約すべきであるという考え方がある．一方で，ソースコードの可読性やモジュールの再利用による機能の凝集性を考慮した場合，すべてのコードクローンを集約すべきではないとも考えられる．そのため，コードクローンの集約はどのようなコードクローンであるのかを開発者がよく見極めて行う必要がある．

一般的にコードクローンが発生する主な要因としてはコピーアンドペーストが挙げられる．開発者が類似処理を開発する際に既存ソースコードをコピーアンドペーストして開発作業を効率化させたいためである．ソースコードの再利用は単一のプロジェクト内だけではなくて，プロジェクトを跨って頻繁に行われる．単一のプロジェクト内で発生するコードクローンをプロジェクト内クローンと呼ぶ．一方で，プロジェクト間で発生するコードクローンをプロジェクト間クローンと呼ぶ．

コードクローンの差異の度合いに応じてコードクローンは以下の 4 つの分類がされている [7]．

タイプ 1　空白やタブ，コーディングスタイル，コメントの有無などの差異を取り除き完全に一致するコードクローン．

タイプ 2　タイプ 1 の差異に加えて，変数名やユーザ定義名，型が異なるコードクローン．

タイプ 3　タイプ 2 の差異に加えて，文の挿入や削除，変更などが行われているコードクローン．

タイプ 4　類似処理ではあるが，構文上の実装が異なるコードクローン．

入力ソースコードからコードクローンを自動で検出する手法はいくつか提案されていて，SourcererCC[*1][8] は，タイプ 3 までのコードクローンを検出可能であり，字句解析に基づいてファイル単位で構文的な類似度が高いコードクローンを検出できる．SourcererCC は Lepes らの研究 [6] などで利用された実績もあり，GitHub の任意のスナップショットに含まれるおよそ 100 万に近いプロジェクト数を対象に 80%以上類似しているファイル単位のコードクローンを検出している．

ここで 2 つの言葉の使い分けに注意したい．1 つはプロジェクト間クローンの類似度である．これはコード片同士の類似度を示していて，クローンペア 1 つにつき 1 つの類似度が算出される．もう 1 つはプロジェクトの類似度である。これはプロジェクト同士の類似度を示していて，プロジェクトペア 1 つにつきそれぞれのプロジェクトの類似度が存在するため，2 つの類似度が算出される．例えば，プロジェクト A とプロジェクト B が類似しているかどうかを判断するためには，それらに含まれるファイル数とプロジェクト間クローンとなっているファイル数の比率を基準に判

[*1] http://github.com/Mondego/SourcererCC

断している．例えば，プロジェクト A に含まれる 8 ファイルのうち，4 ファイルがプロジェクト B とプロジェクト間クローンであると認められる場合，プロジェクト A の類似度は $4/8 = 0.5$ となる．また，プロジェクト B のファイル数が 12 ファイルの場合，その類似度は $4/12 = 0.33$ と計算される．

2.2 クローン変更管理システム Clone Notifier

コードクローン変更管理システム Clone Notifier [10] とは，2 バージョン間のコードクローン変更情報を検出し，分類した結果を開発者に通知するシステムである．まず Clone Notifier はコードクローン検出ツール CCFinder [4] を用いて，各バージョンのソースコードからクローンセットを検出し，2 つのバージョン間のクローンセットの変更情報を検出する．最後に，各クローンセットを変更の内容で以下のように分類する．

- *Stable:* 2 バージョン間で全てのコード片が変更されなかったクローンセット
- *Changed:* 2 バージョン間で一部のコード片に対して編集、追加、削除の操作が行われたクローンセット
- *New:* 新リビジョンにしか存在しないコードクローンセット
- *Deleted:* 旧リビジョンにしか存在しないコードクローンセット

2.3 大規模プロジェクト間クローンに関する既存研究

プロジェクト間クローンに関する大規模な調査がいくつか行われている [1], [6].

Lopes ら [6] は，GitHub で開発されているおよそ 450 万のフォークしていないプロジェクトを対象にプロジェクト間クローンについて調査し，再利用ソースコードの分布を視覚的に表現する DéjàVu を作成した．この関連研究では GitHub のスナップショットを GHTorrent[2] に対して各プロジェクトの詳細情報（スター数，コミット数，コミッタ，フォークの有無，開発主要プログラミング言語など）を取得し，人気のあるプロジェクトやプログラミング言語の違いなどからプロジェクト間クローン分布の傾向を調査している．その結果，調査対象のおよそ 70%のファイルが既存ファイルの再利用であることを突き止めた．

Gharehyazie ら [1] は，プロジェクト間のコード検索の効率化を目的としてプロジェクト全体でのコード再利用実態を調査している．そのためにコードクローン検出ツール Deckard[3] が検出したプロジェクト間クローンに対して統計学的アプローチといくつかのユースケースから再利用ソースコードの特性について調査している．調査結果，GitHub ではプロジェクト間クローンが広く普及していることや，エコシステムの形成はオニオンモデルとなっていることが明らかになった．すなわち，一つの同じプロジェクトから派生的に再利用がされて，さらにその派生は同一アプリケーションドメイン，異なるアプリケーションドメインと変化していくことが明らかになった．

既存研究 [1], [6] では，GitHub 上に存在するプロジェクト間コードクローンが多いことを明らかにした．このことよりソフトウェア開発において，効率化や高い品質，メンテナンスの容易さの面を考慮して OSS を利用することが選択肢の 1 つであるといえる．そのため，ソフトウェアの進化過程では必ずしも 1 つのソフトウェアで完結した進化をするわけではなく，[1] で述べられているようにオニオンモデルとして拡張的な進化していくと予想される．

一方で再利用ソースコードの変更が加えられればその変更を再利用先 OSS で一貫した変更をするのか検討しなければならない．プログラミング言語や依存ソフトウェアの仕様変更，機能拡張，バグの修正などの理由のために変更が加えられるが，再利用ソースコードのメンテナンスを行う開発者は必ずしも変更内容を一貫した変更をする選択が行う必要はない．

Lopes らは再利用ソースコードに関してプロジェクト間クローンを検出することで統計的な再利用ソースコードの分布に関する大規模な調査を行い，再利用ソースコードの特性について考察しているが，それらの一貫性のある変更について調査した研究はない．そこで本研究では，再利用ソースコードの一貫した変更について注目し，実際に変更されたプロジェクト間クローンを検出してその特性について考察を行う．

3. 調査

本章では，再利用ソースコードを調査する目的や手段について詳細に説明する．

3.1 調査の目的

本研究では以下の 3 つのリサーチクエスチョンに従い調査を行った．

RQ1 GitHub に含まれるプロジェクト間クローンのうち，変更されるコードクローンおよび一貫性のない変更がされたコードクローンの割合はどのくらい存在するのか．

RQ2 一貫した変更が行われたプロジェクト間クローンはどのようにして一貫性を実現しているのか，また，どのようなプロジェクトが一貫性を実現しているのか．

RQ3 変更されたプロジェクト間クローンはどのような理由で変更されていたのか．

RQ1 ではプロジェクト間クローンはどの程度の変更がなされるのかについて調査する．さらにその中で一過性のある変更あるいは一貫性のない変更が行われるプロジェクトの数を調査することで本研究の調査対象の全体的な変更傾向について考察する．RQ2 では，変更されているプロジェク

ト間クローンを有するプロジェクトのうち一貫性のある変更されているものに関して調査する．一貫した変更がされているプロジェクトの傾向を見ることで将来的なメンテナンス作業の効率化に貢献できる可能性がある．RQ3 では，実際に変更されたプロジェクト間クローンがどのような理由で変更されたかについて調査する．特に変更が起こりやすいプロジェクトの傾向の調査を行う．

3.2 調査の手順

リサーチクエスチョン回答のために行った調査の手順について説明する．

3.2.1 対象プロジェクトペアの選択

まず調査対象のプロジェクトは Lopes らの研究で検出したプロジェクト間クローンを含む OSS を対象に調査した．彼らの研究では，対象プログラミング言語を C/C++，Java，Python，JavaScript の 4 つに絞っているが，ソースコードの再利用率が最も高いプログラミング言語は JavaScript で，最も低いのは Java であると示している．本研究では，再利用率が中間である C/C++プロジェクトを調査対象として選択した．

次に、対象プロジゼクトペアを選択するために，まず現在も公開されているプロジェクト[*2]を調べ，DéjàVu が公開しているクローン結果をもとに任意のプロジェクトペア 6900 を選択した．Dejavu が公開しているプロジェクト間の類似度はしきい値が 50%以上に限られている．2 バージョンは 2017 年 1 月 19 日を調査開始日付として 2019 年 9 月 30 日〜10 月 6 日を終了日付として採用した。前者は DéjàVu におけるデータセットの構築日と推測された選択した．なお，正確な日付については Lopes らの研究 [6] には明記されていない．また，後者についてはダウンロード期間でダウンロードしたときに最新のバージョン選択している．

3.2.2 プロジェクト間クローンの検出

プロジェクト間クローンについて，本論文では Lopes ら [6] が公開している結果を用いている．そのため，プロジェクト間クローンの検出は既存研究の手法に準拠している．Lopes らが提案したプロジェクト間クローン検出手法について簡単に説明する．まず GitHub のスナップショットに対して，主要プログラミング言語が C/C++で開発されているフォークのないすべてのプロジェクトをダウンロードし，すべての対象ソースコードに対してファイルハッシュ値，トークン列ハッシュ値を算出する．次に，ファイルハッシュ値が一致する場合，ソースコードが一致していると判定し，トークン列ハッシュ値が 80%以上類似していればプロジェクト間クローンであると判定している．コードクローン検出ツール SourcererCC はこれらの判定を行うことができる．

3.2.3 Clone Notifier の拡張

調査のために，本研究では Clone Notifier に 2 点の改変を加えた．1 点目は，コードクローン検出ツールを CCFinder から SourcererCC に変更した．なぜなら，CCFinder では検出できなかったタイプ 3 のコードクローンを SourcererCC は検出できるからである．2 点目は，本研究において詳細な分析を行うための，分類の拡張である [5] ．Kim らが定義したクローンセットの進化パターンを参考に，クローンセットの分類の 1 つである Changed Cloneset にラベルを追加した．

- *Add* 追加されたコード片がある
- *Subtract* 削除されたコード片がある
- *Shift* 別のクローンセットから移動してきたコードクローンが含まれる
- *Consistent* 全てのコード片に同様の編集が行われている
- *Inconsistent* 編集されたコード片と編集されていないコード片が同時に存在する

3.2.4 Clone Notifier によるバージョンにまたがるプロジェクト間クローンの検出

2.2 節で説明した Clone Notifier はダウンロードしてきたプロジェクトからコードクローンを検出し，2 バージョン間の変更を管理できる．本研究では，3.2.1 節で選択したプロジェクトペアを 1 つのプロジェクトとみなして，それぞれの調査対象期間の調査開始と終了日付のコミットを 2 バージョンとみなす．各バージョンごとにプロジェクトペアのコードコーンを検出する．2 バージョンのそれぞれで検出したバージョン間クローンに関する情報を永続化する際に，その各ファイルが属する本来のプロジェクトに関して，プロジェクト間にまたがるようなバージョン間のコードクローンを抽出する．そして，その永続化したプロジェクト間クローンに関する情報に基づいて 3.1 節で述べた 3 つのリサーチクエスチョンにこたえる．なお，この一連の調査プロセスを自動化した際にリポジトリを clone することを失敗する，あるいは SourcererCC の実行時間が非常に長くなる場合を想定して 1 時間のタイムアウトを設けたため，永続化できたプロジェクトペア数は 918 となった．

4. 調査結果

本章では，リサーチクエスチョンに沿って調査結果を報告しその考察を行う．

4.1 RQ1：変更プロジェクト数および一貫しない変更が行われたプロジェクト数

表 1 は，プロジェクト間クローンが存在するプロジェクトについて，プロジェクトが含むクローンセットの変更情報で分類した結果を示す．ここでの変更とは Clone Notifier

[*2] http://dejavu.ics.uci.edu/index

の分類でいう *Changed*，*New* および *Deleted* を指す．ただし，この表からわかるように、今回の対象プロジェクト間クローンにおいてはいずれも *Chganged* または *Stable* に分類され，*New* および *Deleted* は検出されなかった．そのため，表中では *Changed* と *Stable* の分類しか表記していない．

また，表 1 の縦の項目はプロジェクトの類似度で分類している．その分類は既存研究 [6] に従い，「100%」，「80%」，「50%」を目安に 5 つの区分に分けていて，それらの項目は排他的な関係となっている．表 1 の項目を上から順番に説明する．まず「完全一致」とは，2 つのプロジェクトペアが完全に一致していることを指している．その数は 70 あるが，注意したいのは集計はプロジェクト数で行っているので完全一致の項目では 35 ペアの完全一致したプロジェクトペアについて表示していることになる．次の「100%」という項目は，ペアプロジェクトに対して包含されているプロジェクト数を示している．すなわち，プロジェクト A_1 と A_2 において，A_1 が 100%であるとき，A_1 のファイルは A_2 に完全に含まれているが，その反対では部分的にしか含んでいないことを意味している．最後に，この，「100%」項目から下の項目 3 つについては「100%」，「80%」，「50%」をしきい値に範囲ごとの分類している．

表 1 から、調査対象プロジェクトの傾向では，918 プロジェクトのうち 148 プロジェクト（約 16.1%）が変更されていることが分かる．変更されたプロジェクト数の割合が最も高い分類は「完全一致」で約 31.4%であり，最も低い分類は「100%」で約 5%だった．実際の数で比較すると変更プロジェクトが最大であるのは「80%未満 50%以上」の 60 プロジェクトで，最小であるのは「100%」の 5 プロジェクトであった．

表 2 は変更されていた各プロジェクトに含まれていたクローンセットがどのような変更であるのか調査しした結果を示す．ただし，紙面の幅の関係で表 2 の各項目名を省略した表記を用いているので説明する．まず，横軸では変更されたプロジェクト間クローンの分類として Kim らの研究 [5] に従い，「*ADD*」，「*ADD(CONsistent)*」，「*ADD(INConsistent)*」，「*CONSistent*」，「*INConsistent*」，「*SHIFT(INConsistent)*」，「*SHIFT*」の 7 項目としている．また，縦の項目はクローンセットが属するプロジェクトペアのそれぞれの類似度を，括弧内が類似度のしきい値を表しており，ハイフンでそのしきい値の組み合わせを示している．

表 2 の「(100)-(100)」は完全一致するプロジェクトを示していて，その多くが *consistent* な変更であることが分かる．完全一致をしていないプロジェクトの組み合わせでは *consistent* な変更よりも *inconsistent* な変更な多いこともわかった．

変更されたプロジェクト間クローン全体の傾向としては

表 1 プロジェクト間クローンが存在するプロジェクト数の変更情報

	All	*Changed*	*Stable*
完全一致	70	22	48
100%	99	5	94
100%未満 80%以上	205	31	174
80%未満 50%以上	318	60	258
50%未満	226	30	196

一貫した変更に比べて一貫していない変更がおよそ 1.89 倍存在していることが分かった．また，一貫した変更が行われているプロジェクトペアは完全一致していることが多いということが判明した．

RQ1 の答え

RQ1. ではプロジェクト間クローンがどの程度変更されているのかについて調査している．その答えはおよそ 16%であり，決して多い数ではないが，無視できるほどの小さな数でもないと考えられる．その変更プロジェクトのうち，一貫していない変更のほうが一貫した変更よりも多く，一貫した変更が行われているプロジェクトペアは完全一致しているという特徴がある．

4.2 RQ2：一貫性のある変更の手段

RQ2 を答えるために表 2 で示した完全一致しているプロジェクトペア中の一貫した変更をしているプロジェクトペア 10 組すべてについて目視で確認した．これらのプロジェクトを調査したとき，以下の 2 つのコードクローン間で一貫的な変更が行われた理由があることが判明した．

(1) 対象プロジェクトが開発用と公開用の 2 つのリポジトリに分岐していて，定期的に開発者がマージしている．
(2) GitHub の仕様によって，プロジェクトのリネーム後であっても旧 URL からリダイレクトされる．

後者の要因は調査方法の問題点であり，本質的な一貫した変更は前者である．前者に該当するプロジェクトは 2 組存在し，どちらも CRAN[*3]に公開されているプロジェクトだった．これらのプロジェクトペアは一方が開発用リポジトリで，もう一方が公開用リポジトリとなっている．開発用リポジトリでは盛んにコミットが行われ，リビジョンが更新されるタイミングでマージが行われていた．公開用リポジトリは Read Only となっており，認可された開発者しか変更を加えることはできない．更新のタイミングは開発者の任意のタイミングで行われていると考えられる．また，一貫性のない変更が行われていた完全一致するプロジェクト間でも同様にミラーとなっているリポジトリを発見している．

また，表 2 中に完全一致していないにもかかわらず一貫

[*3] https://cran.r-project.org/

表 2　プロジェクト間クローンの変更に関するクローンセットの内訳

	ADD	ADD(CON)	ADD(INC)	CONS	INC	SHIFT(INC)	SHIFT	SUM
(100)-(100)	1	0	0	80	0	0	0	81
(100)-(100:80)	0	0	0	0	0	0	0	0
(100)-(80:50)	0	0	0	0	0	0	0	0
(100)-(0:50)	0	0	0	0	7	7	0	14
(100:80)-(100:80)	1	0	2	3	58	0	2	66
(100:80)-(80:50)	0	0	0	0	14	0	0	14
(100:80)-(0:50)	0	0	0	0	23	3	0	26
(80:50)-(80:50)	0	1	0	14	63	0	0	78
(80:50)-(0:50)	0	0	0	1	11	0	0	12
(0:50)-(0:50)	0	0	0	0	0	0	0	0
SUM	2	1	2	93	176	10	2	

した変更が行われているクローンセットが 18 つ確認できた．これらのうち 1 つのプロジェクトは，それらの類似率はお互いに 97%である．これらが一貫した変更をしているのも開発用と公開用の 2 つのプロジェクトに分岐していることが理由だった．これらはリビジョンの更新時などに定期的にマージされていると予想され，DéjàVu がプロジェクトクローンを検出時には一部一貫した変更がなされていなかったと予想できた．

残りの 17 のプロジェクト間クローンをもつ 5 組のプロジェクトペアについて，観察して考察した結果，次の理由が考えられる．

(3) プロジェクトペアにおける共通の編集者がかかわっているソースコードが含まれているため，同様の修正が行われている．

5 つのプロジェクトペアについて，それぞれのプロジェクトペアを持つ所有者は同じであった．すなわち，これらのそれぞれの一貫した変更は同じ編集者が同時に変更すべきと判断した結果を反映していると考えることができる．

さらに 1 つのプロジェクトについては一貫した変更と一貫性のない変更の両方が含まれていた．一貫性のない変更は実行結果の出力ファイルのデフォルトパスを変更するものであったが，それに相当する処理内容がもう一方のクローン片に含まれておらず，そのプロジェクト間クローンは機能的には処理内容が異なるクローンであると考えられるため，SourcererCC の誤検出であると考えられる．

完全一致するプロジェクトペアを検出する上で，URL のリダイレクトは注意すべき点であるといえる．今回の調査でリポジトリ名等を変更した影響で URL が変更される時，GitHub の仕様で変更前の URL でもリダイレクトされることが判明した．これは分析時にプロジェクトごとに固有の ID を割振る際に URL に対して割り当てているために生じている分析の問題点だと考えられる．

RQ2 の答え

RQ2. では一貫した変更を行っているプロジェクト間クローンについて調査することを目的としている．その答えは次のとおりである．完全に一致しているプロジェクトペアの場合は，公開用リポジトリと開発用プロジェクトをそれぞれ持ち，リポジトリの所有者によって一貫した変更を行うかどうか管理されている．また，完全に一致していないプロジェクトペアの場合は，その修正に同一の編集者が携わっているため一貫した変更が管理されている．

4.3　RQ3：再利用ソースコードの変更理由

まず RQ3 に答えるために，表 2 で示した変更されたプロジェクト間クローンの変更について目視で確認し，次のような理由でプロジェクト間クローンの変更が行われていると判断した．

- 動作環境やライブラリに依存したバージョンアップ
- プロジェクトがもつ機能の拡張・修正

動作環境には Visual Studio などの IDE が含まれる．Visual Studio は頻繁にアップデートされるが，C/C++ではこの Visual Studio の拡張を目的とした開発が存在する．C/C++プロジェクトでは Visual Studio を判別するマクロ_MSC_VER などがある．このような動作環境のバージョンアップに依存して変更が行われるプロジェクトを確認した．また動画ファイルフォーマット Matroska を編集するライブラリに依存するプロジェクトにおいても，そのバージョンアップに依存して変更が行われているなど，ライブラリのバージョン依存な変更を確認した．これら以外の修正理由としては，バグ修正や機能拡張といったものであると判断した．ただし，一部の修正についてはその理由の判別が難しくここでは言及しないものとする．

まず一貫した変更について，4.2 節の調査結果から完全一致あるいは完全一致に近い類似度を持つプロジェクトペアまたは同じ開発者が含まれるプロジェクトペアは一貫し

た変更が行われやすいと考えられる．完全一致しているプロジェクトペアは一貫した変更が行われる意図があって変更を反映していることが 4.2 節の調査で確認できている．同じく同じ開発者が携わっているプロジェクトペアについても，変更すべき箇所を検討することで一貫した変更が行われている可能性はある．

また，一貫性のない変更が行われている 53 プロジェクトペアについて，それぞれプロジェクトの開発活動がどの程度活発なのか調査した．その結果，調査期間 2017 年 1 月 19 日と 2019 年 9 月 30 日〜10 月 6 日の間に変更が一度も行われていないプロジェクトが 41 プロジェクト含まれていた．すなわち，一貫性のない変更が行われるプロジェクトペアの多くは，一方のプロジェクトの開発活動があまり活発ではないと考えることができる．また，このような一方のプロジェクトの開発活動があまり活発ではないプロジェクトペアのうち，所有者が同じリポジトリが 1 組存在した．一方は 2013 年頃まで開発が行われ，それ以降まったく開発されていなかった．もう一方は 2014 年頃から 2018 年頃まで開発が継続して続けられていた．このような開発がずっと前に止まっているプロジェクトは古いバージョンのリポジトリと考えることができる．

一貫性のない変更が行われたプロジェクトペアのうち，調査期間内に両方のプロジェクトに変更が加えられた残りの 12 プロジェクトペアについても考察する．それらの変更は分岐的なプロジェクトの進化が起きている可能性が高いと考えられる．例えば，所有者名が異なるがプロジェクト名が同じあるプロジェクトペアにおいて，一方は動作環境であるハードウェアに依存した変更が加えられて，もう一方は入力できるフォーマットの種類の増加に関する機能拡張が行われていた．このように開発者が望んだ用途を実現するための変更が独自に加えられている．

RQ3 の答え

RQ3. ではプロジェクトペアの変更理由を考察する．非一貫性のない変更の多くは，プロジェクトペアの一方が開発者の開発活動があまり活発ではないために生じている．また，両方のプロジェクトの開発が活発だった場合，それぞれの機能が独自に拡張されている可能性がある．また，動作環境やライブラリに依存したバージョンアップも変更理由として一部存在している．

5. まとめと今後の課題

本研究では，Lopes らの研究 [6] が発見した GitHub 内の再利用ソースコードについて，プロジェクト間で一貫した変更の効率的な開発方法や一貫性のない変更に対する対策を考える目的のために GitHub に含まれる再利用ソースコードの変更に関するリサーチクエスチョンを 3 つ立てて，それらにこたえるためにプロジェクト間クローンを検出し調査した．その結果，完全一致しているプロジェクトペアは一貫した変更が行われており，開発者が必要なタイミングで同期していると考えられる．一貫した変更は，同じ開発者がそれらのプロジェクトに携わっていても行われる．また，一貫した変更に比べて一貫性のない変更のほうが約 1.89 倍存在していることが判明した．プロジェクトの変更理由についても調査し，一貫性のない変更の多くはプロジェクトペアの一方が開発者の開発活動があまり活発ではないために生じていると考察した．また，両方のプロジェクトの開発が活発だった場合，それぞれの機能が独自に拡張されている可能性がある．

また，本研究を通して今後の再利用ソースコードの分析における課題には以下が挙げられる．

- 調査プログラミング言語の拡大
- 調査リポジトリ数の拡大
- エコシステム周辺における変更の伝播に関する考察
- リダイレクトされるプロジェクトの排除
- コードクローン検出手法の精度向上

参考文献

[1] Mohammad Gharehyazie, Baishakhi Ray, and Vladimir Filkov. Some from here, some from there: Cross-project code reuse in github. In *Proceedings of 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pp. 291–301. IEEE, 2017.

[2] Georgios Gousios. The ghtorent dataset and tool suite. In *Proceedings of the 10th working conference on mining software repositories*, pp. 233–236. IEEE Press, 2013.

[3] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. DECKARD: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on Software Engineering*, pp. 96–105. IEEE Computer Society, 2007.

[4] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, Vol. 18, No. 7, pp. 654–670, 2002.

[5] Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy. An empirical study of code clone genealogies. In *Proc. ESEC/FSE*, Vol. 30, pp. 187–195, 2005.

[6] Cristina V Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. Déjàvu: a map of code duplicates on github. *Proceedings of the ACM on Programming Languages*, Vol. 1, No. OOPSLA, pp. 84:1–84:28, 2017.

[7] Chanchal K Roy, James R Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of computer programming*, Vol. 74, No. 7, pp. 470–495, 2009.

[8] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. Sourcerercc: Scaling code clone detection to big-code. In *Proceedings of 2016 IEEE/ACM 38th International Conference on Software*

Engineering (ICSE), pp. 1157–1168. IEEE, 2016.
[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 D, Vol. 91, No. 6, pp. 1465–1481, 2008.
[10] 山中裕樹, 崔恩瀞, 吉田則裕, 井上克郎, 佐野建樹. コードクローン変更管理システムの開発と実プロジェクトへの適用. 情報処理学会論文誌, Vol. 54, No. 2, pp. 883–893, 2013.