修士学位論文

題目

ソースコード編集履歴を用いた
コードクローンリファクタリングの有効性調査

指導教員
肥後 芳樹 教授

報告者
川本 琢人

令和 8 年 2 月 2 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

令和 7 年度 修士学位論文

ソースコード編集履歴を用いたコードクローンリファクタリングの有効性調査

川本 琢人

内容梗概

ソフトウェア開発において，コピーアンドペースト等によって生じる互いに類似したコード片であるコードクローンは，ソフトウェアの保守性を低下させる要因の一つである．コードクローンに対してバグ修正や機能追加等の編集を行う際は，類似するすべてのコードクローンに対して一貫して編集する必要がある．ここでコードクローンを見落としてしまうと，編集に一貫性が無くなり，不整合が生じる．この不整合は，バグの修正漏れや，不十分な機能追加による新たなバグを生む原因となる．

このような問題を解決する手法にコードクローンのリファクタリングがある．コードクローンを一箇所に集約すると，一貫した編集に失敗するリスクを排除できる．コードクローンのリファクタリングの有効性についてはさまざまな議論がある．ソースコード中に残存しているコードクローンに施される編集の少なさを根拠に，コードクローンを積極的にリファクタリングする必要はないとする意見もある．リファクタリングされずに残存しているコードクローンや，リファクタリングの直前，直後にのみ注目した研究は複数見られる一方で，リファクタリングによって集約されたソースコードが経験する編集を長期的に観察した研究は見られない．

本研究では，コードクローンを集約するリファクタリングの有効性を評価するため，集約されたコードクローンのその後の編集履歴を追跡し，調査した．GitHub で管理された 77 件のオープンソースプロジェクトのリファクタリング情報を含む SmartSHARK データセットを対象に調査を行った．メソッドの編集履歴の追跡には，CodeTracker を用いた．データセットから得られた 187 件の集約されたコードクローンの編集履歴を対象に分析した結果，集約されたコードクローンは集約から 1 年以内に平均して 0.78 回の編集を経験していると分かった．これは既存研究で報告されている，集約されずに残存したコードクローンの編集頻度である 0.48 回よりも高い数値であった．

主な用語

コードクローン
リファクタリング

目 次

1　はじめに

コードクローンとは，ソースコード中の互いに一致あるいは類似したコード片をいう [30]．コードクローンに対してバグ修正や機能追加などの編集を行う際には，すべてのコードクローンに対して一貫した編集を行う必要が生じる [30,31]．ここでコードクローンを見落としてしまうと，編集に一貫性が無くなり，不整合が生じる．不整合によって修正したはずのバグが残存したり，新たなバグを生じさせたりしてしまう．

コードクローンが引き起こす保守性の低下を防ぐ方法に，リファクタリング [6] によるコードクローンの集約がある [31]．コードクローンを集約すると，バグ修正や機能追加の編集はただ一箇所を編集するだけで済み，見落としのリスクを回避できる．このリファクタリングは，見落としのリスクを回避できればできるほど，保守性の問題に対して有効なリファクタリングであったといえる．

既存研究では，コードクローンを集約するリファクタリングが実際にどの程度保守性の問題に有効なのか，様々に議論されている．コードクローンは比較的安定しており，無理に集約する必要はないと主張する研究も存在する [18]．また，リファクタリングの編集操作自体に新たなバグを混入させるリスクがあるとの指摘もある [4,5]．コードクローンを集約するリファクタリングの有効性を調査する既存研究は複数存在するが，これらは集約されずに残存しているコードクローンの進化 [13,18,21] や，コードクローン集約の直前，直後における種々のメトリクスの変化 [2] に注目して分析している．一方で，集約されたコードクローンのその後の長期的な変化に注目した分析を行っている研究はみられない．

本研究は，コードクローンを集約するリファクタリングの有効性を，実際のソフトウェアのソースコード編集履歴を用いて評価する．リファクタリングの有効性は次の 2 つの観点から評価した．

- 集約されたコードクローンはどの程度編集されるか．
- 集約によってはじめて必要になる編集はどの程度発生するか．

本研究では，この 2 つの観点を Research Questions に落とし込み，それぞれ考察した．

本研究の主な貢献は以下の通りである．

- 集約されたコードクローンが集約されていないコードクローンよりも頻繁に編集されていると明らかにした．
- コードクローンを集約するリファクタリングに対する評価について，集約後の編集履歴にも編集するべきとの見方を示した．

- 集約されたコードクローンが経験する編集のうち約 15%は，コードクローンを集約してはじめて必要になった編集だと明らかにした．

以降，2 章では，本研究の背景となるコードクローンとリファクタリング，およびリファクタリングの具体的な手法について述べる．3 章では，関連する既存研究を概観し，本研究の動機と Research Questions を提示する．4 章では，調査対象としたデータセットと，メソッドの編集履歴を追跡するための調査手法について述べる．5 章では，分析によって得られた調査結果を示し，具体例を用いて考察を行う．6 章では，本研究の妥当性への脅威について，7 章で結論および今後の課題について述べる．

2　背景

2.1　コードクローン

ソースコード中の互いに一致あるいは類似したコード片をコードクローンという [30]．コードクローンは，ソフトウェアに対する一貫した編集を困難にし，ソフトウェアの開発及び保守に悪影響を与える恐れがある [30,31]．一貫した編集の失敗は，たとえばコードクローンに含まれるバグを修正する場合に発生する．コードクローンにバグが含まれていた場合，このバグを修正するには，すべてのコードクローンに対して一貫して修正を行う必要がある [19]．この場合，修正箇所はソースコード中の複数箇所に散在するため，見落とすおそれがある．一部のコードクローンを見落としてしまうと，コードクローン間に不整合が生じ，バグは残存する．そのため，コードクローンに対する一貫した編集を正しく施すための工夫が求められる．

Roy ら [20] は，コードクローンをその類似度によってタイプ 1 からタイプ 4 の 4 種類に分類した．

タイプ 1. 改行，空白文字，コメント等の違いを除いて一致するコードクローン．

タイプ 2. タイプ 1 の違いに加えて，リテラル，型，識別子の違いを除いて一致するコードクローン．

タイプ 3. タイプ 2 の違いに加えて，文の変更，挿入，または削除の違いを許容して一致するコードクローン．

タイプ 4. 同様な処理を行うが，構文の異なるコードクローン．

2.2　リファクタリング

ソフトウェアの外部的な振る舞いを変えることなく，内部構造を改善する取り組みをリファクタリングという [6]．リファクタリングは，ソースコードの可読性を向上させ，ソフトウェアの保守性を高めるために行われる．コードクローンを集約するリファクタリングは，一貫した編集の失敗を防ぐために有効である [17]．

コードクローンを集約すると，もともと一貫した編集が必要な編集は，単に集約されたコードクローン 1 箇所に対する編集になる．この編集では複数箇所を同時に書き換える必要がなくなるため，見落としは発生しない．本研究では，この編集を一貫性が必要だった編集と呼ぶ．

肥後ら [31] は，コードクローンを集約するための様々なリファクタリングパターンを整理している．この節の残りの部分では，リファクタリングパターンの一つであるメソッドの

抽出と，リファクタリングの結果ソースコードに導入される集約されたコードクローンについて述べる．

2.2.1 メソッドの抽出

メソッドの抽出は，既存のメソッドから特定の部分を切り出し，新たなメソッドとして定義するリファクタリングである [11,24]．主にタイプ 1 およびタイプ 2 のコードクローンに適用できる [11]．コードクローンを新たなメソッドに切り出し，既存のコードクローンをこの新たなメソッドの呼び出しに置き換えて，コードクローンを集約する．本研究では，リファクタリングによって追加された，コードクローンを切り出したメソッド定義を，集約されたコードクローンと呼ぶ．

2.2.2 他のリファクタリングパターンによるタイプ 1，タイプ 2 コードクローンの集約

肥後ら [31] は，メソッドの抽出以外にもクラスの抽出やメソッドの引き上げなどのリファクタリングパターンを挙げている．これらのリファクタリングパターンでも，ソースコードには集約されたコードクローンが追加される．クラスの抽出は，複数のクラスに共通する処理を新たなクラスに切り出すリファクタリングパターンである [11,24]．新たなクラスに含まれるメソッド定義は，複数のクラスに含まれていた共通した処理を集約しているため，集約されたコードクローンといえる．メソッドの引き上げは，サブクラス間で重複するメソッドを親クラスへ移動するリファクタリングパターンである [11,24]．この場合も，親クラスに追加されたメソッド定義は集約されたコードクローンといえる．

2.2.3 タイプ 3 コードクローンの集約

タイプ 3 のコードクローンに対しても，コードクローンを集約する手法が提案されている．この方法の一つに，テンプレートメソッドを用いる方法がある [31]．テンプレートメソッドを用いる手法は，共通の親クラスを持つコードクローンのペアに対して適用できる．コードクローンを集約するメソッドを親クラスに追加し，コードクローンの共通部分は親クラスに引き上げ，異なる処理はそれぞれの子クラスに記述する．堀田ら [32] は，プログラム依存グラフを用いてテンプレートメソッドを作成する手法を提案している．

この他にも，メソッドの抽出リファクタリングをタイプ 3 のコードクローンに拡張する方法が提案されている．メソッドの抽出リファクタリングはタイプ 1 およびタイプ 2 のコードクローンに含まれる差異をパラメータに抽出する [27,31]．タイプ 3 のコードクローンに含まれる文単位の差異を，ラムダ式を用いてパラメータに抽出する方法が提案されている [27]．

2.2.4 コードクローンの予防

ソフトウェアの保守性を高く保つために，コードクローンの発生を未然に防ぐ，あるいは発生した直後に解消するのも有効である．

Yoshida ら [29] は，開発者がメソッドの抽出リファクタリングを行う際，同時にリファクタリングすべきコードクローンを推薦する手法を提案している．開発者が手動でコードクローンを集約する場合，他のコードクローンの存在に気づかず，見落とされたコードクローンが集約されずに残存してしまう場合がある．Yoshida らの提案する手法は，開発者の編集操作を監視し，メソッドの抽出が行われた時点で即座に関連するコードクローンを提示する．この手法により，開発者はコードクローンを見落とさずにリファクタリングできる．また，AlOmar ら [3] は，IDE 上でのコピーアンドペースト操作を監視し，コードクローンが発生した直後にメソッドの抽出リファクタリングを推薦する手法を提案している．この手法により，開発者は意識せずともコードクローンの発生を未然に防止できる．

これらのようなツールを用いて，あるいは手動で同様な操作を行って即座にコードクローンを集約した場合，コードクローンは発生しない．

3　研究の動機と目的

コードクローンの編集頻度やコードクローンを集約するリファクタリングの有効性について，実際のデータに基づいて言及する既存研究は複数存在する．Hotta ら [13] や Krinke ら [18] は，オープンソースプロジェクトのソースコードの編集履歴を調査し，コードクローンと非コードクローンの編集頻度をそれぞれ異なる方法で比較している．この調査は，コードクローンは非コードクローンと比較して編集頻度はより低かったと報告しておりコードクローンの保守コストは高くないと結論付けている．Saha ら [21] は，6 件のオープンソースプロジェクトでコードクローンに施された編集を調査している．この実験データに基づき筆者が算出したところ，タイプ 1 およびタイプ 2 のコードクローンは平均して 0.48 回の編集を経験している．また，AlOmar ら [2] は，開発者がコードクローンの除去を意図して行ったリファクタリングの前後で，ソフトウェアの品質メトリクスがどのように変化するか調査している．この調査によれば，リファクタリングの前後で改善したメトリクスもある一方で，悪化したメトリクスも存在していた．リファクタリングの有効性は単一のメトリクスによって単純に評価できないと結論付けられている．

これらの研究は，コードクローンを集約するリファクタリングの有効性を強く肯定してはいない．しかしこれらの研究は，コードクローンを集約した後のソースコードに実際に起きた編集には関知していない．コードクローンの問題点は，一貫した編集を困難にすることである．コードクローンを集約するリファクタリングの有効性を明らかにするためには，集約されたコードクローンがその後実際に経験する，一貫性が必要だった編集の調査が重要である．本研究は，コードクローンを集約するリファクタリングの有効性を，リファクタリング後のソースコード編集履歴を用いて明らかにする．

3.1　Research Questions

この目的を達成するため，以下に示す 2 つの Research Questions(RQ) を設定した．

RQ1．集約されたコードクローンはどの程度編集されるか．

既存研究では，コードクローンは一般的に安定していると報告されている．リファクタリングによって集約されたコードクローンも同様に安定しているのかを調査する．また，既存研究で報告されている編集回数と比較する．

RQ2．集約によってはじめて必要になる編集はどの程度発生するか．

コードクローンを集約するリファクタリングは，ソースコードに新たなメソッドを追加する．この追加によって，コードクローンを集約せずにそのまま残しておけば発生しなかった編集が発生している可能性がある．このような，集約によってはじ

めて必要になる編集は，ソフトウェアの保守に追加のコストをもたらす．集約によってはじめて必要になる編集がどの程度発生しているか調査する．

次の章では，本研究の調査手法について詳細に述べる．

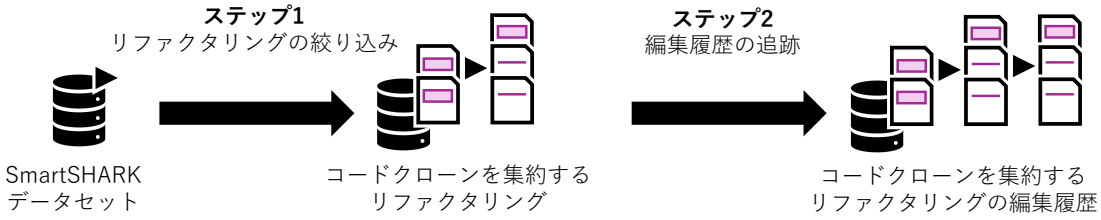

図 1: 調査手法の概観

4 調査手法

本研究では，集約されたコードクローンがその後のソフトウェア進化の過程でどのように編集されていくか調査した．本章では，具体的な調査手法と使用した調査対象のデータセットについて述べる．

4.1 調査手法の概観

調査手法の概観を図 1 に示す．図中の SmartSHARK データセットと CodeTracker は後述する．本研究の調査手法は，2 つのステップに分けられる．

ステップ 1. **リファクタリングの絞り込み**： 調査対象のデータセットから，コードクローンを集約するリファクタリングを絞り込む．

ステップ 2. **編集履歴の追跡**： 集約されたコードクローンの編集履歴を追跡する．

本章の残りの部分では，使用したデータセットと各ステップの詳細を説明する．

4.2 SmartSHARK データセット

集約されたコードクローンのソフトウェア進化の過程を調査するため，ソフトウェア進化の過程に関する様々な情報を格納したデータセットである SmartSHARK [23] を使用する．データセットの規模は以下の通りである．

- プロジェクト数：77 件
- 総コミット数：366,322 件
- 総リファクタリング数：703,260 件

データセットに含まれるすべてのプロジェクトは git でバージョン管理されている．データセットにはリファクタリング検出ツールの RefactoringMiner [25, 26] および RefDiff [22]

によって検出されたリファクタリングの情報も含まれている．リファクタリングの情報は，リファクタリングが施されたコミットとリファクタリングの種別の情報を含む．リファクタリングの種別とは，メソッドの抽出や移動等のリファクタリングの種類を表す文字列である．このリファクタリング検出で用いられている RefactoringMiner は適合率 99.6%，再現率 94%でリファクタリングを検出できるツールであり [25]，SmartSHARK データセットではこの検出結果を使用するにあたり，正確な検出結果と評している [23].

SmartSHARK データセットには，コードクローンの集約以外のリファクタリングも含まれている．次の節で述べるステップ 1 でコードクローンを集約するリファクタリングを取り出す.

4.3 リファクタリングの絞り込み

このステップでは，調査対象の SmartSHARK データセットに含まれるリファクタリングから，コードクローンを集約するリファクタリングのみを取り出す.

まず，コードクローン，即ちソースコードの重複を取り除こうとするコミットで行われたリファクタリングに注目する．リファクタリング情報は、施されたリファクタリングの種別の情報のほか，その編集が行われたコミットの情報，特にコミットメッセージが含まれている．コミットメッセージに “duplicat*” という文字列が含まれているコミットで行われたリファクタリングに絞り込んだ．さらに，より確実にコードクローンを集約するリファクタリングを絞り込むため，検出されたリファクタリングの種別にも注目した．リファクタリングの種別がメソッドの抽出かクラスの抽出であるリファクタリングに絞り込んだ．これらの絞り込みによって，最終的に調査対象のコードクローンを集約するリファクタリングを 345 件得た.

4.4 編集履歴の追跡

このステップでは，コードクローンを集約したリファクタリングから，集約されたコードクローンの編集履歴を得る.

バージョン管理システムを用いるとファイル単位の編集履歴が得られる．Git であれば git log コマンドを用いてファイルの編集履歴を容易に得られる．しかし，メソッドの編集履歴を正確に得るのは困難である．Hora らによると，大規模な Java システムのメソッドの編集のうち，10%から 21%で git log コマンドや Diff ツールによる編集履歴の取得が困難である [12]．編集履歴の取得を困難にする編集の例として，メソッドのリネーム [12, 16] や移動 [10, 12]，抽出 [12] およびインライン化 [7, 12] が挙げられる．取得が困難な編集は，その前後でメソッドの編集履歴を分断してしまう [7, 12]．このような編集があってもできるだけ

編集履歴の分断を起こさない手法が複数提案されている [8–10, 15]．本研究では，メソッドの編集履歴を得るために，CodeTracker [15] を使用する．

CodeTracker [15] は，リファクタリング検出ツールの RefactoringMiner [1, 25, 26] を拡張したツールである．編集履歴を取得するメソッド位置は次の情報から表される．以降，これらの情報の組をメソッド位置と呼ぶ．

- リポジトリの URL
- 時点を表すコミット
- メソッドを含むファイルのパス
- メソッドの開始行

CodeTracker は，編集履歴を調べるメソッドの位置と，編集履歴の開始位置となるコミットを入力にとり，与えられたメソッドの編集履歴を得る．出力は編集のリストとして表される編集履歴である．編集は次の情報から表される．以降，これらの情報の組を編集と呼ぶ．

- 編集が施されたコミット
- 編集前のメソッド位置の情報
- 編集後のメソッド位置の情報
- 編集の種別のリスト

編集の種別とは，RefactoringMiner が検出するリファクタリングの種別に単なるメソッド本体の変更を表す “Body Change” を加えた，編集の種類を表す値である．RefactoringMiner が検出するリファクタリングの種別は現在 100 種類を超える [1, 25, 26]．全種別の紹介は省略するが，たとえばメソッドのリネームは “Rename”、メソッドを定義するクラスの変更は “Container Change” と分類される．同一の編集内で複数種類のリファクタリング，あるいはメソッド本体の変更が同時に検出される場合がある．編集の種別のリストには，検出されたすべての編集の種別が格納される．

RefactoringMiner は，メソッドのリネームやメソッドの移動など，複雑なリファクタリングも検出できる [26]．CodeTracker は，RefactoringMiner が検出したリファクタリング情報を用いて，リファクタリングによって分断されたメソッドの編集履歴をつなぎ合わせてメソッドの編集履歴を構築する．本研究では CodeTracker を，集約されたコードクローンの編集履歴を追跡するために使用する．

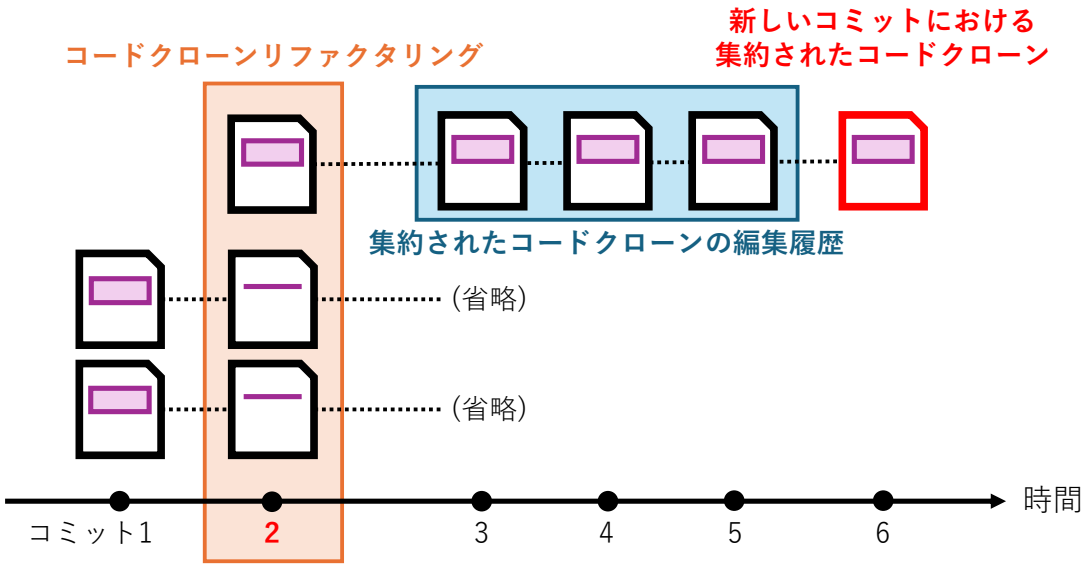

図 2: コードクローンと編集履歴の概要図

集約されたコードクローンと，その編集履歴，最新のコミットにおける集約されたコードクローンの概要を図 2 に示す．図中の左から右に向かって時間が経過しており，黒丸が各コミットを表す．黒い囲みはメソッドを表し，桃色の小さな囲みはコードクローン，桃色の横線はコードクローンを集約したメソッドの呼び出しを表す．この図ではコミット 1 までにコードクローンが既に発生しており，コミット 2 でコードクローンを集約するリファクタリングが行われている．さらに，コミット 3 からコミット 5 にかけて，集約されたコードクローンに何らかの編集がされている．コミット 6 はコミット 5 より新しい，集約されたコードクローンを含むコミットを表す．

この図において，SmartSHARK データセットに含まれるリファクタリングの情報は橙色の囲みで示されたコードクローンリファクタリングにあたる．しかし，CodeTracker に入力するのは，図中で赤く示した，新しいコミットにおける集約されたコードクローンおよびコミット 2 である．CodeTracker はより新しいコミットから過去方向に編集履歴を調べるツールであるが，新しいコミットにおける調査対象のメソッド位置，即ち集約されたコードクローンの情報は SmartSHARK データセットには含まれない．

CodeTracker に入力するメソッド位置の情報を取得するため，集約直後のメソッド位置の情報から，可能な限り新しいコミットにおけるメソッド位置を推定する．この推定が正しいか確認するには，メソッド位置の情報を CodeTracker に入力して，メソッドの編集履歴に集約直後のメソッドが含まれるか確認すればよい．見落としなく目的の編集履歴を得るには，全メソッドの編集履歴を得るのが最も単純な方法である．進化の過程でメソッドが削除され

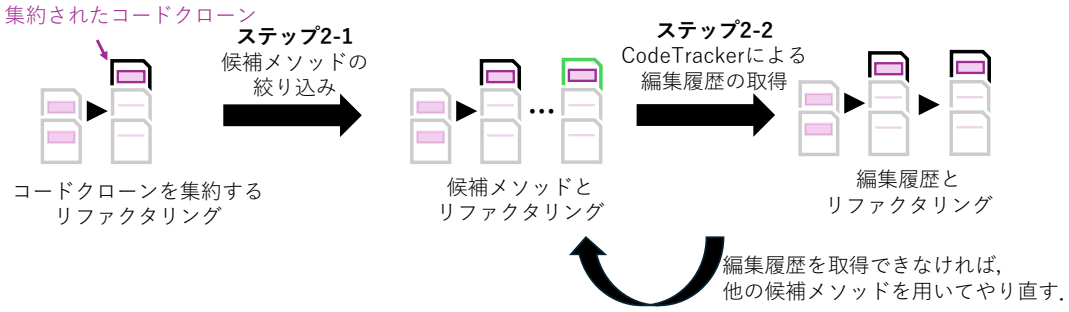

図 3: メソッドの絞り込み手順

ている可能性もあるため，全コミットの全メソッドで編集履歴を取得すれば，見落としなく編集履歴が得られる．しかし，CodeTracker は 200 個のメソッドの編集履歴を得るために約 20 分を要する [15]．全メソッドの編集履歴の取得を，コードクローンの集約以降の全コミットを対象に実行するのは現実的ではない．

編集履歴の取得に要するコストを抑えるため，次節に述べる条件に当てはまるメソッドに絞り込んで編集履歴を得た．また，調査対象のコミットは 1 年ごとに 1 つのコミットを取り出して調査した．次節以降，絞り込んで得たメソッドを指して候補メソッドと呼ぶ．

4.5 メソッドの絞り込み

図 3 にメソッドの絞り込みの手順を示す．ステップ 2 はさらに次の 2 つのステップに分けられる．

ステップ 2-1. 候補メソッドの絞り込み

ステップ 2-2. CodeTracker による編集履歴の取得

編集履歴の取得を試みる候補メソッドを絞り込むにあたり，CodeMapper を使用した．CodeMapper [14] は，コミット間でメソッド等のコード領域を追跡する言語非依存のツールである．このツールは，メソッド位置の情報と追跡するコミットを入力にとり，追跡するコミット内のメソッドの位置を出力する．追跡するコミット内でメソッドが見つからなかった場合は，見つからなかったと出力する．このツールは追跡するコミット内で対象のメソッドを含むファイルを特定するモジュールと，そのファイル内で対応するメソッドを特定するモジュールからなる．本研究では，前者のファイルを特定するモジュールを使用した．

続いて，メソッドの絞り込みの手順をステップごとに説明する．

4.5.1 候補メソッドの絞り込み

まず，CodeTracker の入力に与えるメソッドの候補を，コミットごとにヒューリスティックを用いて絞り込む．メソッドのパラメータが変更されるケースと，メソッドを含むファイルが移動するケースを考慮してヒューリスティックを次の通り定めた．

1. 集約されたコードクローンと同じファイル内にある，シグネチャが集約直後のメソッドと完全に一致するメソッド
2. 集約されたコードクローンと同じファイル内にある，パラメータを除くシグネチャが集約直後のメソッドと一致するメソッド
3. CodeMapper が提案するファイル内にある，シグネチャが集約直後のメソッドと完全に一致するメソッド
4. CodeMapper が提案するファイル内にある，パラメータを除くシグネチャが集約直後のメソッドと一致するメソッド

これらの条件のいずれかを満たすメソッドを CodeTracker の入力に与えるメソッドの候補とした．複数の候補が存在する場合，より先に示した条件を満たすメソッドをより有力な候補とした．得られる編集履歴の長さを可能な限り長くするため，異なるコミットのメソッドの場合，より新しいコミットから取り出したメソッドをより有力な候補とした．

4.5.2 CodeTracker による編集履歴の取得

このステップでは，先のステップで得られたリスト内のメソッド候補を順に CodeTracker に与え，編集履歴を取得する．得られた編集履歴に，コードクローンを集約するリファクタリングが含まれていれば，目的の編集履歴が得られたとした．目的の編集履歴が得られたらその時点で調査を終了する．目的の編集履歴が得られなければ次に有力な候補でこのステップを繰り返す．全ての候補を試しても目的の編集履歴が得られなかった場合は，編集履歴の取得に失敗したとして処理を終了する．

4.6 編集履歴に含まれる編集の分類

RQ2 で言及したように，得られた編集履歴には，集約によってはじめて必要になる編集が含まれる．編集履歴に含まれる編集を，集約によらず必要であった編集，即ち先述の一貫性が必要だった編集と集約によってはじめて必要になる編集に分類する．本研究では，編集が持つ編集の種別のリストにメソッド本体の変更を表す “Body Change” か，コメントやド

キュメントの変更を表す “Documentation Change” のいずれかが含まれる編集を，一貫性が必要だった編集と定義した．

集約によってはじめて必要になる編集の種別の例には，メソッド名の変更を表す “Rename” がある．コードクローンを集約して作成されたメソッドの名前を変える編集は，編集の種別のリストに “Rename” を含む．コードクローンを集約しなければ，このメソッドはそもそも作成されていないため，メソッド名の変更しか行わない編集は集約によってはじめて必要になる編集といえる．

次の章では，上述の手順で得られた編集履歴の分析結果について述べる．

5　調査結果と考察

本章では，調査手法の章で示した手順に沿って得られた編集履歴について述べ，設定した2つのRQに対する回答を与える．また，具体的な事例を通してコードクローン集約の有効性を考察する．

5.1　編集履歴の追跡結果

SmartSHARKデータセットに含まれるリファクタリング703,260件のうち，コードクローンを集約するリファクタリングは345件であった．これらの345件で編集履歴の追跡を行った結果を表1に示す．345件のコードクローン集約リファクタリングのうち，54.2%にあたる187件で，その後の編集履歴を追跡できた．追跡に失敗した158件は，集約から1年経たない間に削除されたか，メソッド名が変わったか，あるいはCodeTrackerで取得できない複雑な編集が行われていた．

表2に，検出された編集の種別とその内訳を示す．集約によってはじめて必要になる編集で最も多く見られた編集の種別は，ソースファイルの移動を表す“Container Change”であった．

図4に，コードクローンの集約からの経過年数ごとに追跡できた編集履歴の数を折れ線グラフで，その経過年数において発生した編集の数を棒グラフで示す．それぞれの具体的な値は表3に示している．編集履歴の数はその年数以上，集約されたコードクローンの編集を追跡している編集履歴の数を表す．また，一貫性が必要だった編集および集約によってはじめて必要になる編集は，その年に施された編集の件数を表す．全編集423件のうち，一貫性が必要だった編集は360件であり，全体の約85%を占める．一方，集約によってはじめて必要になる編集は全体の約15%にあたる63件に留まった．10年以上にわたって追跡できた編集履歴は全体の約27%にあたる51件存在していた．

編集回数の推移を見ると，集約直後の1年目に最も多くの編集が発生しており，その後減少する傾向にある．

表1: 編集履歴の追跡結果.

	件数	割合
追跡成功	187	54.2%
候補なし	158	45.8%
合計	345	100.0%

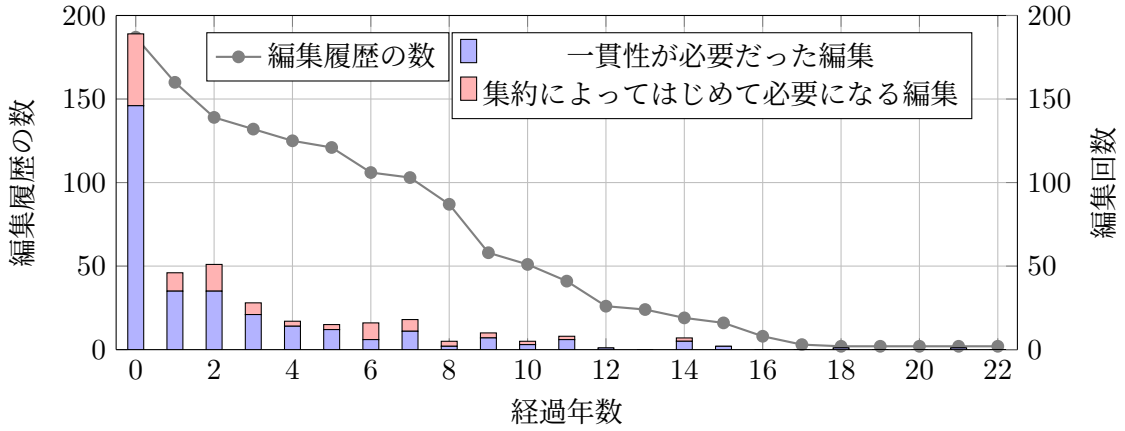

図 4: 経過年数ごとの編集履歴の数と編集の数の推移.

5.2 RQ1：集約されたコードクローンの編集頻度

図 4 の棒グラフを見ると，0 年目の値が最も大きい．これは，集約から 1 年以内の編集が多く検出されたと分かる．しかし，集約からの経過年数が大きくなるにつれ，追跡できた編集履歴の数も減少している．RQ1 に回答するには，編集の頻度，即ちコードクローンあたりの編集の数を調べる必要がある．

追跡できた編集履歴の数の影響を取り除くため，以降では集約されたコードクローンあたりの平均編集回数を用いて議論する．集約されたコードクローンあたりの平均編集回数は，その年における編集回数をその年まで追跡できた編集履歴の数で割って算出する．さらに，集約されたコードクローンに施された編集の累計を取る．図 5 に，集約されたコードクローンあたりの累積平均編集回数を示す．このグラフによると，コードクローンは集約されてから 1 年以内に 0.78 回の一貫性が必要だった編集を経験する．また，10 年経過時点では 1.93 回の編集を経験する．

Saha ら [21] の研究によれば，タイプ 1 およびタイプ 2 のコードクローンが経験する平均編集数は 0.48 回である．本研究の調査結果は，1 年以内の編集のみを集計した時点で既にこの回数を上回っている．10 年経過時点の編集の数は約 4.0 倍であり，明らかな差が見られた．この結果から，開発者が集約したコードクローンは，残存しているコードクローンよりも頻繁に編集される傾向にあるといえる．

したがって，RQ1 への回答は以下のようになる．

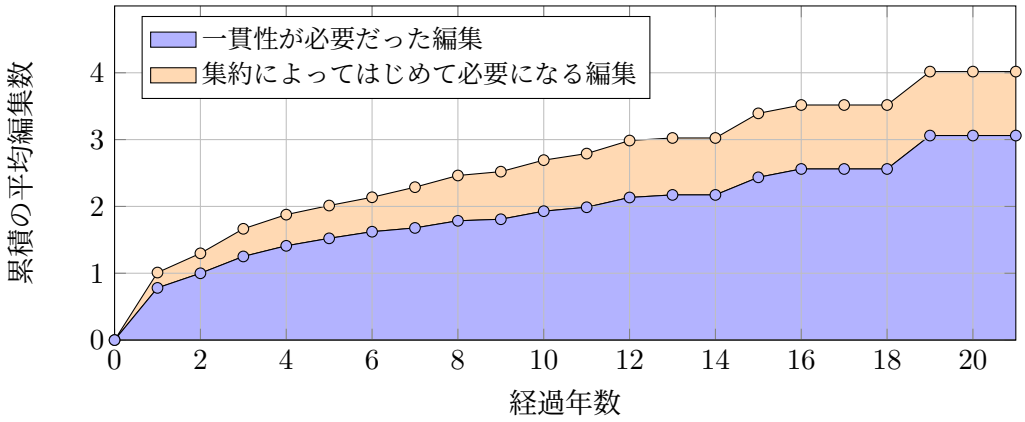

図 5: 経過年数ごとの累積の平均編集数.

RQ1 への回答

集約されたコードクローンは，平均して集約から 1 年以内に 0.78 回，10 年以内に 1.93 回の編集を経験する．編集は，特に集約から 1 年以内に集中しているが，その後も一貫性が必要だった編集は発生し続ける．

5.3 RQ2：集約による追加の保守コスト

RQ2 では，集約によってはじめて必要になる編集の頻度について調査する．前節で示した通り，この編集は全体の約 15%と少数であった．

この編集の内訳をみると，メソッドを含むファイルの位置が変わる Container Change と，メソッドのパラメータが変わる Parameter Change が大半を占める．これらの編集に要するコストをより抑えるためには，適切なリファクタリング機能を備えたツールの活用が有効である．AlOmar ら [2] が指摘するように，リファクタリング自体のコストが低い場合，開発者はより積極的にコード構造の改善に取り組むことができる．高機能なリファクタリングツールを用いれば，リファクタリングの有効性はより高まるといえる．

RQ2 への回答は以下のようになる．

RQ2 への回答

集約に伴って増加する編集は集約されたコードクローンに対する編集の約 15%を占める．また，適切なリファクタリングツールを活用することで，このコストはさらに低減できる．

表 2: 検出されたリファクタリングの種別と件数.

分類	編集の種別	件数
一貫性が必要だった編集	Body Change	294
	Documentation Change	66
集約によってはじめて必要になる編集	Container Change	86
	Parameter Change	83
	Modifier Change	12
	Return Type Change	7
	Exception Change	6
	Annotation Change	4
	Moved	1
	Rename	1

表 3: 経過年数ごとの編集履歴の数と編集の数.

経過年数	編集履歴の数	一貫性が必要だった編集	集約によってはじめて必要になる編集
0	187	146	43
1	160	35	11
2	139	35	16
3	132	21	7
4	125	14	3
5	121	12	3
6	106	6	10
7	103	11	7
8	87	2	3
9	58	7	3
10	51	3	2
11	41	6	2
12	26	1	0
13	24	0	0
14	19	5	2
15	16	2	0
16	8	0	0
17	3	0	0
18	2	1	0
19	2	0	0
20	2	0	0
21	2	1	0
22	2	0	0
合計	187	360	63

5.4 具体例を用いたリファクタリングの有効性の考察

本節では，調査で得られた編集履歴の具体例を用いて，リファクタリングの有効性を考察する．表 4 に，取り上げる例の一覧を示す．それぞれの例を実際のソースコードとともに説明する．

5.4.1 集約後にバグが混入し，その後修正された例

コードクローンの集約が，実際に一貫性のない編集が発生するリスクを回避した例を示す．コードクローンとこれを含むプロジェクトの情報は次の通りである．

- リポジトリ：`apache/activemq`
- クラス：`org.apache.activemq.broker.region.cursors.AbstractStoreCursor`
- メソッド：`setLastCachedId`

集約されたコードクローンの編集履歴を表 5 に示す．集約から約 2.7 年後にバグが混入する編集が行われ，バグは混入から約 1 年後に修正された．

この例で用いる図中では簡単のため，変数名とメソッド名を簡略化し，キャスト演算子を用いたキャストを取り除いている．図 6 は集約直後のメソッドであり，図 7 はバグが混入する直前のメソッドである．集約から 1 年以内にメソッド本体が大きく変更されている．

図 8 はバグが混入したメソッドである．13 行目の `if` は，本来 `else if` とすべきであるが書き損じたものである．この問題を修正した後のメソッドを図 9 に示す．`else` が書き加えられ，8 行目から始まる，`else` 節をもつ 1 文の `if` 文に修正されている．最後のコミットである c1e7dbd コミットは，コミットメッセージにバグ報告のリンクを含んでいた．このバグ報告は上に示した通りの症状を報告していたため，これらの編集をそれぞれバグ混入およびバグ修正の編集とこの実験では判断した．

このバグは，混入してから修正されるまでに約 1 年の期間を要している．また，コードクローンの集約からバグが修正されるまでに 3 年以上の期間が空いている．しかし，集約に

表 4: 本節で紹介する具体例.

プロジェクト名	追跡年数	概要
activemq	11.4 年	集約後にバグが混入し，その後修正された例.
activemq	11.9 年	アクセス修飾子が編集された例.
calcite	12.0 年	ドキュメントのみ編集された例.

よって修正する必要のある行は 1 行のみになっていたため，修正漏れのリスクは排除できていた．コードクローンの集約は，前の編集から次の編集までの期間が空くほど有効だと考えられる．

表 5: 集約されたコードクローンの編集履歴

コミット ID	経過日数 (経過年数)	図番号	概要
54e2e3b	0 日 (0.0 年)	図 6	コードクローンを集約したコミット.
140ce1b	38 日 (0.1 年)	なし	ロジックの編集.
97c127d	40 日 (0.1 年)	なし	ロジックの編集.
b40dc4c	292 日 (0.8 年)	図 7	バグが混入する直前のコミット.
a0ba0bf	977 日 (2.7 年)	図 8	`if` を誤って追加したコミット.
c1e7dbd	1348 日 (3.7 年)	図 9	`if` から `else if` に修正したコミット.

```
1   private void setLastCachedId(MessageId candidate) {
2       if (lastCachedId == null) {
3           lastCachedId = candidate;
4       } else if (candidate.getSequence() > lastCachedId.getSequence()) {
5           lastCachedId = candidate;
6       }
7   }
```

図 6: 集約直後のメソッド（54e2e3b）.

```
1  - private void setLastCachedId(MessageId candidate) {
2  + private void setLastCachedId(final int index, MessageId candidate) {
3  +     MessageId lastCacheId = lastCachedIds[index];
4           if (lastCacheId == null) {
5  -     } else if (candidate.getSequence() > lastCachedId.getSequence()) {
6  -         lastCachedId = candidate;
7  +         lastCachedIds[index] = candidate;
8  +     } else {
9  +         Object lastCacheSequence = lastCacheId.getSequence();
10 +         Object candidateSequence = candidate.getSequence();
11 +         if (lastCacheSequence == null) {
12 +             lastCachedIds[index] = candidate;
13 +         } else if (candidateSequence != null &&
14 +                 candidateSequence > lastCacheSequence) {
15 +             lastCachedIds[index] = candidate;
16 +         }
17      }
18  }
```

図 7: バグ混入前のメソッド（b40dc4c）.

```
1   private void setLastCachedId(final int index, MessageId candidate) {
2       MessageId lastCacheId = lastCachedIds[index];
3       if (lastCacheId == null) {
4           lastCachedIds[index] = candidate;
5       } else {
6           Object lastCacheSequence = lastCacheId.getFutureOrSequenceLong();
7           Object candidateSequence = candidate.getFutureOrSequenceLong();
8           if (lastCacheSequence == null) {
9               lastCachedIds[index] = candidate;
10          } else if (candidateSequence != null &&
11                  candidateSequence > lastCacheSequence) {
12              lastCachedIds[index] = candidate;
13  +       } if (LOG.isTraceEnabled()) {
14  +           LOG.trace("no set last cached...");
15          }
16      }
17  }
```

図 8: バグが混入した直後のメソッド（a0ba0bf）.

```
1   private void setLastCachedId(final int index, MessageId candidate) {
2       MessageId lastCacheId = lastCachedIds[index];
3       if (lastCacheId == null) {
4           lastCachedIds[index] = candidate;
5       } else {
6           Object lastCacheSequence = lastCacheId.getFutureOrSequenceLong();
7           Object candidateSequence = candidate.getFutureOrSequenceLong();
8           if (lastCacheSequence == null) {
9               lastCachedIds[index] = candidate;
10          } else if (candidateSequence != null &&
11                  candidateSequence > lastCacheSequence) {
12              lastCachedIds[index] = candidate;
13  -       } if (LOG.isTraceEnabled()) {
14  +       } else if (LOG.isTraceEnabled()) {
15              LOG.trace("no set last cached...");
16          }
17      }
18  }
```

図 9: バグが修正された直後のメソッド（c1e7dbd）.

5.4.2 アクセス修飾子が編集された例

コードクローンの集約によって追加の保守コストがかかってしまった例を示す．この例では，アクセス修飾子が編集されている．コードクローンとこれを含むプロジェクトの情報は次の通りである．

- リポジトリ：`apache/activemq`
- クラス：`org.apache.activemq.store.kahadb.KahaDBStore`
- メソッド：`recoverRolledBackAcks`

表 6 に，このメソッドの編集履歴を示す．

表 6: 集約されたコードクローンの編集履歴 (recoverRolledBackAcks)

コミット ID	経過日数 (経過年数)	図番号	概要
cfe099d	0 日 (0.0 年)	図 10	コードクローンを集約したコミット．
ed5edb0	2031 日 (5.6 年)	なし	パラメータ追加とロジックの編集．
e6cec27	3988 日 (10.9 年)	図 11	リファクタリング．
b7184b4	4326 日 (11.9 年)	図 12	アクセス修飾子の編集．

集約後にアクセス修飾子が編集された例を示す．アクセス修飾子の編集は，表 2 における編集の種別では “Modifier Change” にあたる．この編集は，集約によってはじめて必要になる編集に分類される．

この例で用いる図中では簡単のため，メソッド本体を省略する．図 10 に示すように，集約直後ではこのメソッドのアクセス修飾子は `protected` であった．しかし，10 年以上が経過した後の図 12 の編集で，アクセス修飾子が削除され，外部のパッケージには公開されない `package private` に編集された．さらに，プロジェクト内に限定して調べた限りでは，このメソッドが子クラスで呼び出された形跡は見られなかった．

この例では，コードクローンを集約して追加したメソッドのアクセス修飾子を，集約時に少し広めに設定してしまったものと考えられる．コードクローンを集約して追加されたメソッドのアクセス修飾子は，可能な限り狭く設定しておくべきだといえる．

```
1   protected int recoverRolledBackAcks(StoredDestination sd, Transaction tx, int
      maxReturned, MessageRecoveryListener listener) throws Exception {
2       ...
3   }
```

図 10: 集約直後のメソッド.

```
1 - protected int recoverRolledBackAcks(StoredDestination sd, Transaction tx,
            int maxReturned, MessageRecoveryListener listener) throws Exception {
2 + protected int recoverRolledBackAcks(String recoveredTxStateMapKey,
            StoredDestination sd, Transaction tx, int maxReturned,
            MessageRecoveryListener listener) throws Exception {
3       ...
4   }
```

図 11: アクセス修飾子変更前のメソッド.

```
1 - protected int recoverRolledBackAcks(String recoveredTxStateMapKey,
            StoredDestination sd, Transaction tx, int maxReturned,
            MessageRecoveryListener listener) throws Exception {
2 + int recoverRolledBackAcks(String recoveredTxStateMapKey,
            StoredDestination sd, Transaction tx, int maxReturned,
            MessageRecoveryListener listener) throws Exception {
3       ...
4   }
```

図 12: アクセス修飾子変更後のメソッド.

5.4.3 ドキュメントのみ編集された例

Documentation Change のみの編集があった例を示す．コードクローンとこれを含むプロジェクトの情報は次の通りである．

- リポジトリ：`apache/calcite`
- クラス：`org.apache.calcite.util.Util`
- メソッド：`firstDuplicate`

表 7 に，このメソッドの編集履歴を示す．

表 7: 集約されたコードクローンの編集履歴 (firstDuplicate)

コミット ID	経過日数 (経過年数)	図番号	概要
6bd6e8e	0 日 (0.0 年)	図 13	コードクローンを集約したコミット．
a611d64	302 日 (0.8 年)	なし	ソースファイルの移動．
a0ba73c	302 日 (0.8 年)	なし	ソースファイルの移動およびロジックの編集．
4ae0298	725 日 (2.0 年)	なし	リファクタリング．
ecf4d6d	748 日 (2.1 年)	なし	ロジックの編集．
ce2ae64	2386 日 (6.5 年)	図 14	ドキュメントのみの編集．
850f0f4	2424 日 (6.6 年)	なし	リファクタリング．

この例で用いる図中では簡単のため，メソッド本体の一部を省略する．図 14 の編集では，メソッド本体には変更がなく，既存の実装の意図を説明するコメントが追加されている．この編集は，表 2 における編集の種別ではコメントやドキュメントの変更を表す “Documentation Change” に分類される．また，このコミットでは同メソッドに他の編集を施しておらず，他の編集の種別には該当しない．

仮にコードクローンを集約していなかった場合，どのコードクローンを見ても同じコメントを確認できるようにするには，全コードクローンで同様のコメントを付加する必要がある．更に，すべてのコードクローンについて，一貫してコメントを更新し続ける必要がある．実際のソースコードの更新に伴ってコメントを更新するのはプログラム理解を助けるために重要である [28]．実際のソースコードとコメントの間の不整合はプログラム理解を妨げるといわれている [28]．タイプ 1 およびタイプ 2 のコードクローンは型名や識別子名等を除いたプログラムの構造が一致しているため，処理の実装自体はほとんどの場合同一であろうと考え

られる．同一の実装に付加するコメントは同様に同一であるべきと考えられるため，本研究では “Documentation Change” も一貫性が必要だった編集として扱った．

```
1   public static <E> int firstDuplicate(List<E> list) {
2       final int size = list.size();
3       if (size < 2) {
4           // Lists of size 0 and 1 are always distinct.
5           ...
6           return -1;
7       }
8       final Map<E, Object> set = new HashMap<E, Object>(size);
9       for (E e : list) {
10          if (set.put(e, "") != null) {
11              return set.size();
12          }
13      }
14      return -1;
15  }
```

図 13: 集約直後のメソッド.

```
1   public static <E> int firstDuplicate(List<E> list) {
2       final int size = list.size();
3       if (size < 2) {
4           // Lists of size 0 and 1 are always distinct.
5           ...
6           return -1;
7       }
8   +   // we use HashMap here, because it is more efficient than HashSet.
9       final Map<E, Object> set = new HashMap<E, Object>(size);
10      for (E e : list) {
11          if (set.put(e, "") != null) {
12              return set.size();
13          }
14      }
15      return -1;
16  }
```

図 14: コメントが追加されたメソッド.

6　妥当性への脅威

本研究の妥当性への脅威について，内的妥当性と外的妥当性の観点から述べる．

6.1　内的妥当性への脅威

6.1.1　編集履歴追跡ツールの精度

本研究では，リファクタリングの検出に RefactoringMiner [1, 25, 26] を，メソッドの編集履歴の追跡に CodeTracker [15] を使用した．RefactoringMiner の適合率は 99.6%，再現率は 94%と高精度であるものの，検出結果が常に正しいとは限らない．本研究で使用したツールの精度は，調査結果の妥当性に影響を与える可能性がある．

6.1.2　見落とされた編集履歴による調査結果の偏り

本研究では，調査対象とした 345 件の集約されたコードクローンのうち，45.8%にあたる 158 件の集約されたコードクローンで，メソッドの編集履歴を追跡できなかった．追跡に失敗する原因のうち，研究の妥当性に影響を及ぼす原因には，4 章で述べたメソッドの絞り込みのステップにおける，絞り込みの条件や CodeMapper の精度が挙げられる．これらの原因による追跡失敗が，分析結果に偏りを生じさせている可能性がある．

6.2　外的妥当性への脅威

6.2.1　データセットの偏り

本研究では，SmartSHARK データセットに含まれる 77 件のオープンソースプロジェクトを調査対象とした．別のオープンソースプロジェクトを対象に同様の調査をすると，本研究の報告とは異なる傾向が得られる可能性がある．

7　おわりに

本研究では，リファクタリングによって集約されたコードクローンの編集履歴を用いて，リファクタリングの有効性を調査した．集約されたコードクローンの編集履歴を用いるため，CodeTracker [15] を用いた．調査の結果，集約されたコードクローンは集約から 1 年以内に平均して 0.78 回の編集を経験していると判明した．この数値は既存研究で報告されている残存したコードクローンの編集頻度である 0.48 回よりも高い．

コードクローンを集約してはじめて必要になる編集は全体の約 15%であった．具体的な事例から，集約されたコードクローンが長期間にわたって保守され，バグ修正や機能追加などの一貫性が必要だった編集が適切に管理されている様子が観察された．また，コードクローンを集約して追加されたメソッドのアクセス修飾子は可能な限り小さく設定しておくべきとの考察を得た．

以上の結果から，コードクローンの集約は，特に将来の編集が予想されるコードクローンに対して実施されており，実際に集約が行われた例の多くにおいては，このリファクタリングがソフトウェアの保守性向上に寄与していると結論付けられる．

今後の課題は次の 2 つが挙げられる．

- 追跡に失敗したケースについての分析と手法の改善
- 本研究の結果を基にしたコードクローンの自動リファクタリング手法の開発

本研究の調査では，345 件のコードクローンリファクタリングのうち，約 46%にあたる 158 件で編集履歴の追跡に失敗した．追跡に失敗したケースには，集約直後にメソッドが削除されたケースや，メソッド名やクラス構造が大幅に変化するなどの複雑な編集があったケースが考えられる．より多くのケースについて調査するため，追跡に失敗したケースを分析し，追跡手法を改善したい．

本研究の結果と既存研究を比較すると，コードクローンは，将来活発に編集されるだろうコードクローンに限ってリファクタリングするべきだと考えられる．本研究で得られた知見を基に，コードクローンの特徴から将来の編集頻度の高さを予測し，集約すべき適切なコードクローンを提示する自動リファクタリング推奨手法の開発に取り組みたい．

謝辞

本研究を行うにあたり，研究の方向性や論文執筆，発表など，日頃より多岐にわたって御指導，御助言を賜りました，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 肥後 芳樹 教授に深謝申し上げます．本研究の全課程を通して，論文執筆や発表などについて，適切な御助言を賜りました，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 クラ ラウラ ガイコビナ 教授に深く感謝申し上げます．本研究の全課程を通して，研究の方向性の検討や論文執筆や発表について，様々な御指導，御助言を賜りました，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 准教授に篤く感謝申し上げます．本研究の全課程を通して，研究の方向性の検討や研究生活について，的確な御助言を賜りました，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 ヌリ オリビエ 助教に心より感謝申し上げます．また，日頃より助言や激励をいただき，研究活動を始め多くのサポートを賜りました，研究室事務職員の 軽部 瑞穂 氏，宮崎 貴羅 氏に心より感謝申し上げます．研究室生活の様々な場面で支えてくださ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肥後研究室の皆さまに感謝申し上げます．最後に，日頃の生活を支え，応援してくれた家族に感謝申し上げます．ありがとうございます．

参考文献

[1] P. Alikhanifard and N. Tsantalis. A novel refactoring and semantic aware abstract syntax tree differencing tool and a benchmark for evaluating the accuracy of diff tools. *ACM Transactions on Software Engineering and Methodology*, Vol. 34, No. 2, January 2025.

[2] E. A. AlOmar. An empirical study on the impact of code duplication-aware refactoring practices on quality metrics. *Information and Software Technology*, Vol. 182, p. 107687, 2025.

[3] E. A. AlOmar and M. W. Mkaouer. Anticopypaster: An open-source ecosystem for just-in-time code duplicates extraction. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 2*, SIGCSE 2024, p. 1923, New York, NY, USA, 2024. Association for Computing Machinery.

[4] A. Bagheri and P. Hegedüs. Is refactoring always a good egg? exploring the interconnection between bugs and refactorings. In *Proceedings of the 19th International Conference on Mining Software Repositories*, MSR '22, pp. 117–121, New York, NY, USA, 2022. Association for Computing Machinery.

[5] G. Bavota, B. De Carluccio, A. De Lucia, M. Di Penta, R. Oliveto, and O. Strollo. When does a refactoring induce bugs? an empirical study. In *IEEE 12th International Working Conference on Source Code Analysis and Manipulation*, pp. 104–113, 2012.

[6] M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, Boston, MA, USA, 1999.

[7] M. W. Godfrey and L. Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Transactions on Software Engineering*, Vol. 31, No. 2, pp. 166–181, 2005.

[8] F. Grund, S. A. Chowdhury, N. C. Bradley, B. Hall, and R. Holmes. Codeshovel: Constructing method-level source code histories. In *IEEE/ACM 43rd International Conference on Software Engineering*, pp. 1510–1522, 2021.

[9] H. Hata, O. Mizuno, and T. Kikuno. Historage: fine-grained version control system for java. In *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th Annual ERCIM Workshop on Software Evolution*,

IWPSE-EVOL '11, pp. 96–100, New York, NY, USA, 2011. Association for Computing Machinery.

[10] Y. Higo, S. Hayashi, and S. Kusumoto. On tracking java methods with git mechanisms. *Journal of Systems and Software*, Vol. 165, p. 110571, 2020.

[11] Y. Higo, S. Kusumoto, and K. Inoue. A metric-based approach to identifying refactoring opportunities for merging code clones in a java software system. *J. Softw. Maint. Evol.*, Vol. 20, No. 6, pp. 435–461, November 2008.

[12] A. Hora, D. Silva, M. T. Valente, and R. Robbes. Assessing the threat of untracked changes in software evolution. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pp. 1102–1113, New York, NY, USA, 2018. Association for Computing Machinery.

[13] K. Hotta, Y. Sano, Y. Higo, and S. Kusumoto. Is duplicate code more frequently modified than non-duplicate code in software evolution?: An empirical study on open source software. In *Proceedings of the Joint ERCIM Workshop on Software Evolution and International Workshop on Principles of Software Evolution*, pp. 73–82, 09 2010.

[14] H. Hu and M. Pradel. CodeMapper: A language-agnostic approach to mapping code regions across commits. In *Proceedings of the 48th International Conference on Software Engineering*, 2026.

[15] M. Jodavi and N. Tsantalis. Accurate method and variable tracking in commit history. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, pp. 183–195, New York, NY, USA, 2022. Association for Computing Machinery.

[16] S. Kim, K. Pan, and E. J. Whitehead. When functions change their names: automatic detection of origin relationships. In *12th Working Conference on Reverse Engineering*, pp. 10 pp.–152, 2005.

[17] J. Krinke. A study of consistent and inconsistent changes to code clones. In *14th Working Conference on Reverse Engineering*, pp. 170–178, 2007.

[18] J. Krinke. Is cloned code more stable than non-cloned code? In *Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*, pp. 57–66, 2008.

[19] A. Lozano and M. Wermelinger. Assessing the effect of clones on changeability. In *IEEE International Conference on Software Maintenance*, pp. 227–236, Beijing, China, 2008. IEEE.

[20] C. Roy and J. Cordy. A survey on software clone detection research. *School of Computing TR 2007-541*, Vol. 541, No. 115, pp. 3–7, 01 2007.

[21] R. K. Saha, C. K. Roy, K. A. Schneider, and D. E. Perry. Understanding the evolution of type-3 clones: an exploratory study. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pp. 139–148. IEEE Press, 2013.

[22] D. Silva and M. T. Valente. Refdiff: Detecting refactorings in version histories. In *IEEE/ACM 14th International Conference on Mining Software Repositories*, pp. 269–279, 2017.

[23] A. Trautsch, F. Trautsch, and S. Herbold. The smartshark repository mining data, 2021.

[24] N. Tsantalis and A. Chatzigeorgiou. Identification of extract method refactoring opportunities. In *Proceedings of the 2009 European Conference on Software Maintenance and Reengineering*, CSMR '09, pp. 119–128, USA, 2009. IEEE Computer Society.

[25] N. Tsantalis, A. Ketkar, and D. Dig. Refactoringminer 2.0. *IEEE Transactions on Software Engineering*, Vol. 48, No. 3, pp. 930–950, 2022.

[26] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, and D. Dig. Accurate and efficient refactoring detection in commit history. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pp. 483–494, New York, NY, USA, 2018. ACM.

[27] N. Tsantalis, D. Mazinanian, and S. Rostami. Clone refactoring with lambda expressions. In *IEEE/ACM 39th International Conference on Software Engineering*, pp. 60–70, 2017.

[28] F. Wen, C. Nagy, G. Bavota, and M. Lanza. A large-scale empirical study on code-comment inconsistencies. In *IEEE/ACM 27th International Conference on Program Comprehension*, pp. 53–64, 2019.

[29] N. Yoshida, S. Numata, E. Choiz, and K. Inoue. Proactive clone recommendation system for extract method refactoring. In *IEEE/ACM 3rd International Workshop on Refactoring*, pp. 67–70, 2019.

[30] 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. 529–536, 2001.

[31] 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. 4_43–4_56, 2011.

[32] 堀田圭佑, 肥後芳樹, 楠本真二. プログラム依存グラフを用いたコードクローンに対するテンプレートメソッドパターン適用支援手法. 電子情報通信学会論文誌 D, Vol. J95-D, No. 7, pp. 1439–1453, 7 2012.