コードクローンへの欠陥混入防止に向けた欠陥混入クローンの特徴分析

野口 耕二朗
和歌山大学 システム工学部
noguchi.kojiro@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

大規模ソフトウェアの品質および保守効率の改善を目的として，欠陥が混入したコードクローンを検出する手法や欠陥混入クローンの特徴分析などの研究が行われている．しかしながら，コードクローンに欠陥が混入する原因を明らかにした研究は存在しておらず，クローンに対して欠陥が混入するのを未然に防ぐ方法については十分な検討がなされていない．本論文では，*RxJava* プロジェクトを対象としてコードクローンに欠陥が混入する原因を分析した結果について報告する．欠陥混入クローンのソースコードメトリクスの特徴を分析した結果，欠陥が混入していないコードクローンと比較して，*(1)* コードクローンに対する入力の数が大きい，*(2)* 規模に関するメトリクスの値が全体的に小さい，という結果が得られた．また，欠陥混入クローンの欠陥を分析した結果，コードクローンを持たない部分に混入した欠陥と比較して，*(1)* コードの不整備による欠陥が多く，*(2)* 欠陥の修正範囲が大きい，という結果が得られた．

1. はじめに

短期納品が求められるソフトウェア開発では生産性の向上を図る必要がある．生産性向上の手段の 1 つに，コピー&ペーストによる既存ソースコードの再利用がある．コピー&ペーストが行われる中で作成される互いに「一致」または「類似」したソースコードの断片をコードクローンと呼ぶ．

生産性の向上を目的としてコピー&ペーストを多用しすぎると，ソフトウェア中のコードクローンが増加するため，コードクローンに対して変更を加える必要が生じた場合，対象コードクローンの探索や変更のためのコストが必要となるため，コピー&ペーストの多用は結果的に生産性を低下させる場合がある．また，一部のコードクローンに変更漏れがあった場合，新たな不具合を混入することがあるため，品質を低下させる要因ととしてコードクローンは除去の対象とされることが多い．コードクローンによるソフトウェア品質の低下に対処するため，これまでにコードクローンの検出手法や [1–6] やコードクローンの追跡・除去手法 [7–12] に関する研究が盛んに行われてきた．また，コードクローンの性質に関する研究 [13, 14] も盛んに行われている．

一方，Kasper らの研究 [15] は，コードクローンが保守性に悪影響を与えることが少ないことが報告されている．コードクローンすべてを注意して管理する対象とするのではなく，品質に悪影響を与えるものに限定して管理することが重要であることが明らかになったことから，近年ではソフトウェアの品質を低下させる要因となりうる欠陥混入クローンに関する研究 [16–18] がいくつか行われている．特に，Mondal らの研究 [19] では，欠陥混入クローンの 33%から欠陥が混入したままさらにコードクローンが生成されることを明らかにしている．

欠陥混入クローンから欠陥が混入したままコードクローンが生成されると，さらなる品質低下につながるだけではなく欠陥を修正する手間が増えることになる．そのため，コードクローンへの欠陥混入を防止することが重要となるが，欠陥混入の防止手段を考えるためには，コードクローンに欠陥が混入する原因をまず明らかにする必要がある．しかし，コードクローンに欠陥が混入する原因について調査した研究はほとんど存在しない．本研究では，コードクローンに欠陥が混入する原因を明らかにすることを目的として，欠陥混入クローンのソースコードおよび欠陥の特徴を分析する．

2. 欠陥混入クローン

本章では，欠陥混入クローンに関する先行研究を紹介し，本研究の位置付けを明確にする．

2.1. 先行研究

コードクローンは欠陥が混入した際の修正のコストを上げる要因と考えられる場合が多く，コードクローンに関する研究は現在までに多数行われている．特に，コードクローンの検出手法に関する研究 [1–6] やコードクローンの追跡・除去手法に関する研究 [7–12] はこれまで盛んに行われてきた．

コードクローンの検出手法や追跡・除去手法以外には，コードクローンそのものの性質を分析した研究 [13–18] がある．これまでの研究により，ソフトウェアの保守性に悪影響を与えるコードクローンは少ないこと [15] や，コードクローンの欠陥密度がコードクローンでないコード片よりも小さいこと [16] などが明らかになっており，コードクローンがソフトウェアの品質に必ずしも悪影響を与える訳ではないことが示唆されている．したがって，コードクローンすべてを注意して管理する対象とするのではなく，品質に悪影響を与えるものに限定して管理することが重要である．

欠陥が混入したコードクローン，すなわち，欠陥混入クローンに関する研究もいくつか行われている．Mondal らの研究 [17] では，より直近に変更されたコードクローンに欠陥が混入しやすいことを明らかにしている．Li らの研究 [18] では，ソースコードをプログラム依存グラフに変換し重複した欠陥が混入しているコードを探索するツール CBCD を開発している．Mondal らの研究 [19] では，コードクローンに混入した欠陥の拡散の様子を調査し，欠陥混入クローンの 33%から欠陥が混入したままさらにコードクローンが生成されることを明らかにしている．

2.2. 本研究の位置付け

先行研究では主に欠陥混入クローンそのものの性質や保守に与える影響について分析を行っている．一方本研究は，コードクローンに欠陥が混入する原因を明らかにすることを目的としている．コードクローンに欠陥が混入する原因を明らかにすることで，コードクローンに欠陥が混入するのを防止する手段を考えることができるようになる．

Mondal らの研究 [17] は，より直近に変更されたコードクローンに欠陥が混入しやすいという欠陥混入クローンの性質を明らかにしており，直近の変更を監視することで欠陥の混入を早期に除去することができるという知見を提供しているという点においては本研究と類似している．しかしながら，本研究ではすべてのリビジョンを対象に欠陥混入クローンのおよび欠陥の特徴を広範に分析することで，コードクローンに欠陥が混入するのを未然に防ぐための知見を提供しようとする点で Mondal らの研究 [17] とは目的が異なる．

3. アプローチ

本章では，欠陥混入クローンの調査を行うにあたって設定したリサーチクエスチョンについて説明し，それぞれの具体的な分析手順について述べる．

3.1. リサーチクエスチョン

本研究では，以下のリサーチクエスチョンを設定し欠陥混入クローンについての分析を行う．

RQ1 欠陥混入クローンと非欠陥混入クローンとの間でソースコードメトリクスにどのような違いがあるか

- **RQ1-1** 複雑度に関するメトリクスにどのような違いがあるか
- **RQ1-2** 規模に関するメトリクスにどのような違いがあるか

RQ2 コードクローンに混入した欠陥とコードクローンを持たない部分に混入した欠陥の特徴にどのような違いがあるか

- **RQ2-1** 欠陥の種類にどのような違いがあるか
- **RQ2-2** 修正範囲はコードクローンに混入した欠陥の方が大きいか

RQ1 を設定した理由は，欠陥混入クローンのメトリクスの特徴的な値を調査し，**コードクローンに欠陥が混入する原因となったソースコードの特徴**を明らかにするためである．そのため，RQ1-1 では，ソースコードの複雑さと欠陥の混入しやすさとの関係を調査する．調査

表 1. RQ1 で用いる複雑度および規模メトリクス

カテゴリ	メトリクス	説明
複雑度	CountInput	メソッドに対する入力（パラメータの数，使用しているグローバル変数 測定対象のメソッドが呼び出されている回数）の合計
	CountOutput	メソッドに対する出力（メソッド内で呼び出しているメソッドの数など）の合計
	Cyclomatic	メソッドの制御フローグラフにおいて，(制御グラフ内のエッジ数)− (制御グラフ内のノード数) + (連結されたコンポーネントの数) で計算される複雑度
	CyclomaticModified	Cyclomatic 複雑度を測る際に複数の分岐制御構造（java や C 言語における switch 文など） のそれぞれの分岐をカウントせず全体を 1 としてカウントする
	CyclomaticStrict	Cyclomatic 複雑度を測る際に論理演算などをそれぞれ 1 としてカウントする
	Essential	Cyclomatic 複雑度を測る際に単純な条件構造（if-else，while，do-while など）を 単一のステートメントで置き換えて計測した複雑度
	Knots	break, goto などにより制御フローの経路が交差する数
	CountPath	制御文の実行可能な経路の数
	CountPathLog	ユニークな経路を常用対数で示した値
	MaxNesting	制御構造（if，while，for，switch など）の最大ネストレベルの値
規模	CountLine	行数
	CountLineBlank	空白行数
	CountLineCode	空白行やコンパイルされない行を除いたコードの行数
	CountLineCodeDecl	宣言文の行数
	CountLineCodeExe	実行可能なコードの行数
	CountLineComment	コメントの行数
	CountSemicolon	セミコロンの数
	CountStmt	宣言文や実行可能なステートメントの数
	CountStmtDecl	宣言のステートメント数
	CountStmtExe	実行可能なステートメント数
	RatioCommentToCode	コード行に占めるコメント行の割合

するソースコードの複雑さは条件分岐構造の経路数とネストの深さ，サイクロマティック複雑度の大きさである．RQ1-2 では，ソースコードの規模と欠陥の混入しやすさとの関係を調査する．調査するソースコードの規模は行数とステートメント数，コメント数の大きさである．

RQ2 を設定した理由は，コードクローンに混入した欠陥を調査し，**コードクローンに混入しやすい欠陥の特徴**を明らかにするためである．そのため，RQ2-1 では，コードクローンにどのような種類の欠陥が混入しやすいかを調査する．RQ2-2 では，コードクローンに混入した欠陥を修正した範囲の大きさを調査する．

3.2. 分析手順

3.2.1. 分析準備

本研究ではコードクローン追跡ツール CCT (Code Clone Tracer) [12] を用いて欠陥混入クローンの抽出を行う．紙面の都合上，CCT の詳細については割愛するが，CCT は粗粒度なコードクローン検出手法 [6] をベースとして，分散型版管理システムと不具合管理システムとを関連付ける機能を付加したツールである．ブロック単位でのコードクローンの検出・追跡が行うのと同時に，不具合混入クローンの特定・追跡が行える．CCT で用いられているコードクローン検出手法で検出したコードクローンの最小トークン数は 30 である．

3.2.2. RQ1 の分析手順

RQ1 では，欠陥が混入したコードクローン（欠陥混入クローン）と欠陥が混入していないコードクローン（非欠陥混入クローン）の複雑度および規模メトリクスを計測し比較する．メトリクスについては，ソースコード解析ツール Understand[1]を用いて取得する．表 1 に，Understand で得られるメトリクスの内，本分析で用いるメトリクスとその概要を示す．

分析ではまず，対象プロジェクトの対象期間の全リビジョンから CCT を用いて欠陥混入クローンと非欠陥混入クローンを特定し，Understand によりメトリクスを取得する．以降では，メトリクスの集計や比較の手順について説明する．

コードクローンの親子関係の特定：コードクローンの親子関係とは，あるリビジョンの前後で前のリビジョンのコードクローンが後のリビジョンの同じ位置にある場合に結ばれる関係である．図 1 は，分散型版管理システム Git でのコードクローンの親子関係を表したものである．図中の Rev5 に含まれるコードクローン集合 A の親をたどることで Rev1 に含まれるコードクローン集合 A'

[1]Understand,
https://www.techmatrix.co.jp/product/understand/

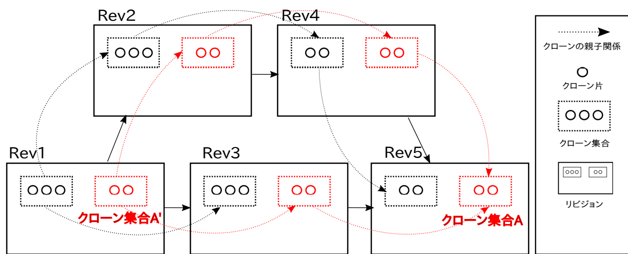

図 1. コードクローンの親子関係の例

が親であることを特定することができる．また，コードクローン集合 A からコードクローン集合 A' までのコードクローン集合の親子関係がわかるようになる．

本分析ではコードクローンに欠陥が混入してから修正されるまで欠陥混入クローンがどのリビジョンに含まれているかを知る必要がある．図中のコードクローン集合 A に欠陥が混入していて Rev5 が欠陥を修正する直前のリビジョンとする．コードクローン集合 A に混入している欠陥を欠陥 D とし，Rev1 でコードクローン集合 A' に混入したことが分かっているとする．コードクローン集合 A の親をコードクローン集合 A' までたどることで，どのリビジョンのどのコードクローン集合に欠陥 D が混入しているかがわかるようになる．

メトリクスの集計：どのリビジョンにどの欠陥混入クローンが含まれているかが分かれば，それぞれのリビジョンで欠陥混入クローンと非欠陥混入クローンとを区別することができる．Understand で計測したメトリクスが記録されている CSV ファイルにはどのコード片がどのようなメトリクスを持っているかがわかるようになっている．欠陥混入クローンであるコード片と非欠陥混入クローンであるコード片とで別々にメトリクスを集計する．

メトリクスの比較：比較対象とする非欠陥混入クローンは，欠陥混入クローンを含むリビジョンの中から欠陥混入クローン以外のコードクローンを用いる．欠陥混入クローンと非欠陥混入クローンのメトリクスの比較は，それぞれのメトリクスで欠陥混入クローンのメトリクスの値と非欠陥混入クローンのメトリクスの値に対してマンホイットニーの U 検定の両側検定を有意水準 $\alpha = 0.05$ で行い，有意差があるかどうかを確かめる．また，それぞれのメトリクスで欠陥混入クローンのメトリクスと非欠陥混入クローンの平均値にどれだけ差があるかを確かめるために Cohen's d による効果量を算出する．

3.2.3. RQ2 の分析手順

RQ2 ではまず，コードクローンに混入した欠陥の不具合票を CCT を用いて取得する．次に，コードクローンを持たない部分に混入した欠陥の不具合票をコードクローンに混入した欠陥の不具合票と同数になるようにを無作為に取得する．コードクローンと非コードクローンそれぞれに混入した欠陥に対する不具合票を調査し，欠陥の種類と修正箇所の範囲を比較する．以降では，欠陥の種類の分析と修正範囲の分析の手順について説明する．

欠陥の分析：コードクローンに混入した欠陥とコードクローンを持たない部分に混入した欠陥の不具合票を目視し，欠陥を分類する．欠陥の種類については分析対象固有の分類は行わず，ドメインに依存しない粒度で欠陥の種類を分類する．例えば，無限ループやメモリリーク，不正な値の出力などは「想定外の振る舞い」，新たに必要となる機能の追加は「不足している機能の追加」といった一般性の高い分類を行う．コードクローンに混入した欠陥とコードクローンを持たない部分に混入したコードクローンの欠陥を，種類ごとに集計した結果をそれぞれ棒グラフで示し，どのような違いがあるかを比較する．

修正範囲の分析：欠陥の修正範囲を行数とファイル数の 2 種類を用いて集計する．修正された行数やファイル数は Git の各リビジョンで記録されている．欠陥修正リビジョンから修正された行数やファイル数を抽出することで，その欠陥を修正するのに必要となった範囲（欠陥の影響範囲）がわかるようになる．

4. 分析結果

4.1. データセット

本研究では，リアクティブプログラミングを Java で使用するためのライブラリである RxJava プロジェクトを対象に分析を行う．表 2 にデータセットの概要を示す．

分析対象として RxJava を選んだ理由の 1 つは，本研究で用いるコードクローン検出ツール CCT は Java にしか対応しておらず，Java を使用しているプロジェクト

表 2. 対象プロジェクトの対象期間とリビジョン数

プロジェクト	対象期間	リビジョン数
RxJava	2014 年 8 月 30 日～2017 年 3 月 29 日	1,703

表 3. 欠陥混入クローンなどの個数

欠陥混入クローン集合	非欠陥混入クローン集合	リビジョン数
2,478	62,654	691

である必要があるためである．2つ目の理由は，RxJavaのリビジョン数が多いからである．リビジョン数が多いことで検出することが出来るコードクローンの数も多くなるため，リビジョン数が多いRxJavaは分析対象として適していると言える．3つ目の理由は，不具合の報告が他のプロジェクトに比べて活発に行われているからである．多種多様な不具合のデータを多く集めることが出来るためRxJavaは分析対象として適していると言える．

4.2. RQ1：欠陥混入クローンと非欠陥混入クローンとの間でソースコードメトリクスにどのような違いがあるか

表3に，特定した欠陥混入クローン集合と非欠陥混入クローン集合の個数および欠陥混入クローン集合を含むリビジョンの数を示す．非欠陥混入クローンとは，欠陥混入クローンを含むリビジョンに存在する欠陥が混入していないコードクローンを指す．表2，3より，分析対象期間の全リビジョン数は1,703件あるのに対して，欠陥混入クローン集合を含む691件あり，約41% (691/1703)のリビジョンのコードクローンに欠陥が混入している状態であることがわかる．なお，RxJavaはGitHub[2]上のプロジェクトであるため分散型版管理システムGitを用いた開発を行っている．3.2.2項の図2に示したように，分析対象はメインブランチだけではなく分岐し将来的にマージされるサブブランチも含まれている点には留意されたい．複雑度と規模メトリクスを比較した結果については以降で詳細に説明する．

RQ1-1：複雑度に関するメトリクスにどのような違いがあるか

複雑度に関するメトリクスが欠陥混入クローンと非欠陥混入クローンとの間でどのように違いがあったかについて報告する．表4に，欠陥混入クローンと非欠陥混入クローンとの間の複雑度メトリクスの有意差検定の結果と効果量を示す．表5に，欠陥混入クローンと非欠陥混入クローンのそれぞれの複雑度メトリクスの中央値と平均値を示す．

[2]GitHub, https://github.com/

表 4. RQ1-1 の検定結果

メトリクス	p 値	Cohen's d
CountInput	0.000	**0.312**
CountOutput	0.000	**-0.399**
Cyclomatic	0.000	-0.070
CyclomaticModified	0.000	-0.068
CyclomaticStrict	0.000	0.090
Essential	0.000	**-0.245**
Knots	0.000	**-0.255**
CountPath	0.000	-0.165
CountPathLog	0.000	**-0.298**
MaxNesting	0.000	**0.269**

表 5. 複雑度に関するメトリクスの中央値と平均値

	中央値		平均値	
メトリクス	欠陥混入	非欠陥混入	欠陥混入	非欠陥混入
CountInput	4	3	4.334	3.733
CountOutput	3	4	3.421	4.084
Cyclomatic	3	2	2.823	2.936
CyclomaticModified	3	2	2.823	2.933
CyclomaticStrict	4	2	3.206	3.047
Essential	1	1	1.409	1.769
Knots	0	1	0.953	1.514
CountPath	3	2	2.973	3.576
CountPathLog	0	0	0.187	0.327
MaxNesting	1	1	1.555	1.322

表4より，欠陥混入クローンと非欠陥混入クローンとの間ですべての複雑度メトリクスに有意差があることがわかる．また，CountInput, CountOutput, Essential, Knots, CountPathLog, MaxNestingの効果量が，小から中程度の値（0.2〜0.4程度）であることがわかる．効果量の値が正である場合は欠陥混入クローンのメトリクスの平均値の方が大きく，効果量の値が負である場合は欠陥混入クローンのメトリクスの平均値の方が小さいことを示している．

CountInput, MaxNestingの2種類の効果量は正の値であり，欠陥混入クローンの方が平均値が大きいことがわかる．表5からも欠陥混入クローンの方がCountInputが大きいが読み取れる．したがって，値が大きいコードクローンには欠陥が混入しやすいと言える．Cyclomaticの効果量 (-0.070) は小さかったもののMaxNestingは小程度の正の効果量が示されていることから，ネストが深いコードクローンには欠陥が混入しやすいと言える．ただし，表5からはネストの数の中央値がどちらも1であり平均値も大きな差としては観察されていないという点には注意が必要である．

一方，CountOutput, Essential, Knots, CountPathLog の 4 種類の効果量は負の値であり，欠陥混入クローンの方が平均値が小さいことがわかる．表 5 からも欠陥混入クローンの方が CountOutput が小さいことが読み取れる．したがって，内部で呼び出しているメソッドの数が少ないコードクローンに欠陥が混入しやすいと言える．同様に，Essential から，単純な条件構造を 1 つのステートメントとした場合の複雑度が小さいコードクローンに欠陥が混入しやすいと言える．また，Knots から，break 文や goto 文が少ないコードクローンに欠陥が混入しやすいことが分かった．CountPathLog から，実行可能な経路数が小さいコードクローンに欠陥が混入しやすいことが分かった．なお，表 5 より，これら 5 種類のメトリクスについても中央値と平均値の差は僅差であるため注意が必要である．

RQ1-2：規模に関するメトリクスにどのような違いがあるか

規模に関するメトリクスが欠陥混入クローンと非欠陥混入クローンとの間でどのように違いがあったかについて報告する．表 6 に，欠陥混入クローンと非欠陥混入クローンとの間の規模メトリクスの有意差検定の結果と効果量を示す．表 7 に，欠陥混入クローンと非欠陥混入クローンのそれぞれの規模メトリクスの中央値と平均値を示す．

表 6 より，欠陥混入クローンと非欠陥混入クローンとの間ですべての規模メトリクスに有意差があることがわかる．また，CountLine, CountLineCode, CountLineCodeExe, CountSemicolon, CountStmt, CountStmtExe の効果量が，小程度（0.2 程度）の負の値を示していることがわかる．欠陥混入クローンにおける上記メトリクスの平均値の方が非欠陥混入クローンに比べて小さいことを示している．したがって，行数やステートメント数などが小さいコードクローンに欠陥が混入しやすいと言える．ただし，表 7 からわかるように，RQ1-1 の複雑度メトリクスと同様に，規模メトリクスも中央値と平均値の差は僅差であるため注意が必要である．

4.3. RQ2：コードクローンに混入した欠陥とコードクローンを持たない部分に混入した欠陥の特徴にどのような違いがあるか

CCT により，コードクローンに混入した欠陥に紐づけられた不具合票は合計 86 個検出された．コードクローンに混入した欠陥と同数のコードクローンを持たない部分に混入した欠陥を無作為に抽出し，欠陥の種類と修正範囲の比較を行う．

表 6. RQ1-2 の検定結果

メトリクス	p 値	Cohen's d
CountLine	0.000	**-0.251**
CountLineBlank	0.000	0.033
CountLineCode	0.000	**-0.220**
CountLineCodeDecl	0.000	-0.065
CountLineCodeExe	0.000	**-0.235**
CountLineComment	0.000	-0.164
CountSemicolon	0.000	**-0.263**
CountStmt	0.000	**-0.223**
CountStmtDecl	0.000	-0.127
CountStmtExe	0.000	**-0.226**
RatioCommentToCode	0.000	-0.191

表 7. 規模に関するメトリクスの中央値と平均値

	中央値		平均値	
メトリクス	欠陥混入	非欠陥混入	欠陥混入	非欠陥混入
CountLine	9	11	11.849	14.168
CountLineBlank	0	0	0.688	0.645
CountLineCode	9	9	10.144	11.652
CountLineCodeDecl	2	1	1.836	1.915
CountLineCodeExe	5	6	6.277	7.469
CountLineComment	0	0	1.059	1.909
CountSemicolon	3	5	4.352	5.300
CountStmt	6	7	7.119	8.212
CountStmtDecl	2	2	1.800	1.961
CountStmtExe	4	5	5.319	6.251

RQ2-1：欠陥の種類にどのような違いがあるか

図 2 に，コードクローンに混入した欠陥とコードクローンを持たない部分に混入した欠陥の不具合票から目視し，欠陥を種類ごとに集計した結果について示す．また，表 8 に分類した欠陥の種類と概要を示す．

図 2 からいずれのコードクローンにおいても,「想定外の振る舞い」,「既存の機能の改良」,「不足している機能の追加」が欠陥の種類として多く観察された．一方で，コードクローンに混入した欠陥には,「コードの不整備」に関するものが比較的多いことがわかる．また，コードクローンを持たない部分に混入した欠陥には,「JavaDoc の部分の修正」が比較的多いことが見て取れる．しかし，「JavaDoc の部分の修正」はメトリクスに影響を与えないことに注意する必要がある．また,「不要な機能の削除」については，コードクローンを持たない部分に比べ（7 件），コードクローンに混入することが少ない（3 件）

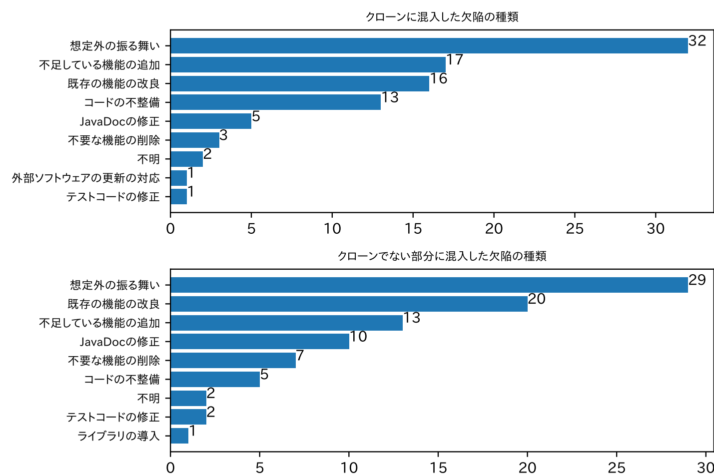

図 2. 欠陥の種類 (上:コードクローン 下:非コードクローン)

表 8. 欠陥の種類の説明

欠陥の種類	説明
想定外の振る舞い	意図していない値の出力や動作を引き起こす部分を是正する際に必要となる修正
既存の機能の改良	パフォーマンスの向上や新たに追加された機能の対応の際に必要となる修正
不足している機能の追加	他の部分の変更に対応して必要となる機能の追加や要件を満たすための機能の追加の際に行われる修正
不要な機能の削除	一致または類似した機能の存在により一方の機能を削除する際に必要となる修正
コードの不整備	修正を行った時点では必ずしも修正が必要ではなかったが，ソースコードの品質を上げるために行われた修正
JavaDoc の修正	java で記述するソフトウェアの API 仕様書を書くために必要となる JavaDoc を記述している部分で行われる修正
テストコードの修正	テストコードの部分だけ行われた修正
外部ソフトウェアの更新の対応	プロジェクトが用いているライブラリなどのバージョンアップに対応するために必要となる修正
ライブラリの導入	ライブラリの導入の対応のために必要となる修正

ことがわかる．

RQ2-2：修正範囲はコードクローンに混入した欠陥の方が大きいか

欠陥の修正範囲に関するメトリクスについて，コードクローンに混入した欠陥とコードクローンを持たない部分に混入した欠陥の間にどのような違いがあったかを報告する．表 9 に，コードクローンに混入した欠陥とコードクローンを持たない部分に混入した欠陥との間の修正範囲に関するメトリクスの有意差検定の結果と効果量を示す．表 10 に，コードクローンに混入した欠陥とコードクローンを持たない部分に混入した欠陥のそれぞれの修正範囲に関するメトリクスの中央値，平均値を示す．

表 9 より，コードクローンに混入した欠陥とコードクローンを持たない部分に混入した欠陥の修正範囲との間で，行数およびファイル数のどちらのメトリクスにも有意差があることがわかる．また，行数およびファイル数のどちらのメトリクスも効果量が正の値かつ 0.2 を超えていることが見て取れる．コードクローンに混入した欠陥の修正範囲がコードクローンを持たない部分に混入し

表 9. RQ2-2 の検定結果

メトリクス	p 値	Cohen's d
行数	0.000	0.321
ファイル数	0.000	0.418

表 10. 欠陥の修正範囲に関するメトリクスの中央値と平均値

	中央値		平均値	
メトリクス	欠陥混入	非欠陥混入	欠陥混入	非欠陥混入
行数	562	150	8038.1	631.2
ファイル数	7	3	54.1	10.5

た欠陥の修正範囲に比べて大きいことを示唆している．表10から，コードクローンを持たない部分に混入した欠陥を修正する場合と比較して，コードクローン混入した欠陥を修正する場合には中央値で約3.8倍，平均値で約12.7倍の範囲の修正が必要であることがわかる．同様に，ファイル数についてもコードクローン混入した欠陥を修正する場合には中央値で約2,3倍，平均値で約5.2倍のファイルを対象に修正を行う必要があり，修正漏れ（≒品質低下）の原因の１つとなり得ることが示唆される．

5. 考察

本章では4章でケーススタディにより得られた結果に対して考察を行う．

5.1. 欠陥混入クローンのソースコードメトリクス

5.1.1. 複雑度に関するメトリクスについて

欠陥混入クローンと非欠陥混入クローンとの間で複雑度に関するメトリクスの値を比較した結果，すべてのメトリクスで有意差があることが確認できた．非欠陥混入クローンと比較して欠陥混入クローンの方が平均値が大きかったメトリクスは，CountInput，MaxNestingであり，平均値が小さかったメトリクスはCountOutput，Knots，Essential，Knots，CountPathLogである．以下ではそれぞれのメトリクスについて考察する．

CountInputとは，メソッドで使用されているパラメータ，グローバル変数，測定対象のメソッドが呼び出されている回数の合計である．CountInputの値が大きいということは，メソッドが外部と結ばれている関係が多いことを意味する．関係が結ばれている部分で修正が行われた場合にメソッドも修正する必要がある可能性が発生するため，コードクローンに欠陥が混入しやすくなると考えられる．

MaxNestingとは，メソッドの最大ネスト数である．ネスト数が大きくなることでコードが複雑になることがあり，可読性が低下する場合がある．非欠陥混入クローンと比較して欠陥混入クローンの方がMaxNestingが大きいため，欠陥混入クローンの可読性がより低下したことを示唆しており，コードクローンに欠陥が混入しやすくなった原因の一つと考えられる．

Essential，Knots，CountPathLogという条件分岐構造の経路の多さや複雑さを意味するメトリクスが小さいクローンに欠陥が混入しやすくなった．一方，RQ2-1では,「不足している機能の追加」と「既存の機能の改良」が36.6%という多くの割合を占めてることがわかった．経路の多さや複雑さに関係なく混入する「不足している機能の追加」と「既存の機能の改良」により，欠陥混入クローンの条件分岐構造の経路数や複雑度自体は小さくなった可能性があり，今後詳細に分析する必要がある．

5.1.2. 規模に関するメトリクスについて

複雑度に関するメトリクスと同様に，欠陥混入クローンと非欠陥混入クローンとの間でメトリクスの値に有意差があることが確認された．効果量に関してはほとんどのメトリクスにおいて負の値で絶対値0.1以上となっており，それらのメトリクスの平均値は欠陥混入クローンより非欠陥混入クローンの方が大きいことが示されている．つまり，非欠陥混入クローンの方が欠陥混入クローンに比べてコードの行数やステートメントの数などが大きいことを意味している．

規模が小さいコード片からクローンを生成する際は，その時点では規模が小さく欠陥が混入していないことがわかっている中でクローンを生成する．しかし，コードが変更されていく中でそれらクローンに対しても変更が加えられ，その中で欠陥が混入するものと考えられる．一方で，規模が大きいコードからクローンを生成する際は，以降に機能の追加や改善が必要ないものである可能性がある．これらの仮説を検証するためには，今後さらに詳細に分析する必要がある．

5.2. 欠陥混入クローンに混入した欠陥

5.2.1. 欠陥の種類

クローンに混入した欠陥とクローンでない部分に混入した欠陥との間で欠陥の種類を比較した結果，クローンに混入した欠陥の種類として多く見られたのはコードの不整備である．実際にコードが不整備であることによって修正された例を図3に示す．行頭に-がついた行は，修正の際に削除された行で，+がついた行は追加された行である．削除された部分は図3のクローン以外にも存在しており，同じような修正が行われている．削除された部分を修正したり機能追加する場合に同じような修正がこのコードが存在する部分で必要となるため，保守性が低下する．保守性の低下を避けるために図3のようにク

```
@Override
public void onNext(U t) {
        frc.dispose();
        if (tus.compareAndSet(false, true)) {
-               serial.onSubscribe(EmptySubscription.INSTANCE);
-               serial.onComplete();
+               EmptySubscription.complete(serial);
        }
}
...
@Override
public void onComplete() {
        frc.dispose();
        if (tus.compareAndSet(false, true)) {
-               serial.onSubscribe(EmptySubscription.INSTANCE);
-               serial.onComplete();
+               EmptySubscription.complete(serial);
        }
}
```

図 3. コードの不整備によって修正された例

ローンを 1 つのメソッドに集約し，修正の手間を減らすような処置を行うことがある．クローンが存在する場合は図 3 のような修正が必要になる場面がクローンでない部分と比較して多いことから，クローンに混入した欠陥としてコードの不整備が多く見られたと考えられる．

5.2.2. 欠陥の修正範囲

コードクローンに混入した欠陥とコードクローンを持たない部分に混入した欠陥との間で欠陥の修正範囲の大きさを比較した結果，全てのメトリクスで有意差が見られた．また，効果量がすべて正で 0.2 を超える値となっている．つまり，コードクローンに混入した欠陥の方が修正範囲が広いことを意味している．

欠陥を修正する際にその欠陥と関係があるコードも修正する場合がある．例えば，欠陥が発生した箇所がメソッドであり，そのメソッドの戻り値の型を変更する必要があるとする．そのメソッドを呼び出している部分でその値を格納している変数があった場合，その変数の型をその戻り値の型と同じものにする必要がある．このようにして欠陥が発生した箇所が他のコードと多くの関係があるほど修正範囲も大きくなる．

クローンに混入した欠陥の修正範囲はメソッドの呼び出しの関係などに加えて，クローン集合の一部または全体を修正する可能性がある．クローンが存在することにより欠陥の修正範囲が大きくなる場合があり，クローンに混入した欠陥の修正範囲が広くなったと考えられる．

6. まとめと今後の課題

本研究ではコードコードクローンに欠陥が混入する原因を明らかにするために欠陥混入クローンの特徴の分析を行った．複雑度に関する分析から，欠陥混入クローンに対する入力数が大きく，出力数が小さいという結果が得られた．また，条件分岐構造の最大ネスト数は大きいが，経路数と経路の複雑度が小さいという結果が得られた．規模に関する分析では，ほとんどのメトリクスにおいて欠陥混入クローンの方が非欠陥混入クローンと比べて小さいという結果が得られた．クローンに混入した欠陥の種類に関する分析では，欠陥混入クローンに特有の欠陥の種類が「コードの不整備」であるという結果が得られた．欠陥の修正範囲に関する分析では，クローンに混入した欠陥の修正範囲はクローンでない部分に混入した欠陥の修正範囲に比べて大きいという結果が得られた．修正範囲が大きいことは欠陥が混入した箇所の影響範囲が大きいことを示している．クローンに欠陥を混入させないことが保守の生産性・品質向上に重要であることが改めて示された．

本研究の今後の課題は，分析対象プロジェクトを増やして知見の一般性を高めること，得られた分析結果からコードクローンに欠陥が混入することを未然に防ぐためのツールを開発することなどがある．

謝辞

本研究を実施するにあたり有益なアドバイスを頂きました久木田雄亮氏に心より感謝します．本研究の一部は，文部科学省科学研究補助金（基盤 (A): 17H00731，基盤 (C): 18K11243）による助成を受けた．

参考文献

[1] 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.

[2] I.D. Baxter, A. Yahin, L. Moura, M. Sant’Anna, and L. Bier, “Clone detection using abstract syntax trees,” Proceedings of the International Conference on Software Maintenance(ICSM1998), pp.368–377, 1998.

[3] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, “Deckard: Scalable and accurate tree-based detection of code clones,” ICSE ’07, pp.96–105, 2007.

[4] J. Krinke, “Identifying similar code with program dependence graphs,” WCRE ’01, pp.301–309, 2001.

[5] C.K. Roy and J.R. Cordy, “Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization,” ICPC ’08, pp.172–181, 2008.

[6] 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.

[7] 肥後芳樹，植田泰士，神谷年洋，楠本真二，井上克郎，“コードクローン解析に基づくリファクタリングの試み,” 情報処理学会論文誌，vol.45，no.5，pp.1357–1366，2004.

[8] N. Yoshida, Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue, “On refactoring support based on code clone dependency relation,” METRICS ’05, pp.10pp.–16, 2005.

[9] 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.

[10] E. Duala-Ekoko and M.P. Robillard, “Clone region descriptors: Representing and tracking duplication in source code,” ACM Transactions on Software Engineering and Methodology (TOSEM), vol.20, no.1, pp.3:1–3:31, 2010.

[11] Y. Higo, K. Hotta, and S. Kusumoto, “Enhancement of crd-based clone tracking,” IWPSE ’13, pp.28–37, 2013.

[12] 久木田雄亮，大平雅雄，“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,” ソフトウェア・シンポジウム 2017 in 宮崎，pp.120–129，2017.

[13] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, “An empirical study of code clone genealogies,” ESEC/FSE-13, pp.187–196, 2005.

[14] V. Saini, H. Sajnani, and C. Lopes, “Comparing quality metrics for cloned and non cloned java methods: A large scale empirical study,” ICSME ’16, pp.256–266, 2016.

[15] C.J. Kapser and M.W. Godfrey, “”cloning considered harmful” considered harmful: Patterns of cloning in software,” Empirical Software Enggineering, vol.13, no.6, pp.645–692, 2008.

[16] H. Sajnani, V. Saini, and C.V. Lopes, “A comparative study of bug patterns in java cloned and non-cloned code,” SCAM ’14, pp.21–30, 2014.

[17] M. Mondal, C.K. Roy, and K.A. Schneider, “Identifying code clones having high possibilities of containing bugs,” ICPC ’17, pp.99–109, 2017.

[18] J. Li and M.D. Ernst, “Cbcd: Cloned buggy code detector,” ICSE ’12, pp.310–320, 2012.

[19] M. Mondal, C.K. Roy, and K.A. Schneider, “Bug propagation through code cloning: An empirical study,” ICSME ’17, pp.227–237, 2017.