社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

リファクタリング実施履歴を用いたCode Smellの深刻度に関する調査

雜賀　翼† 崔　恩瀞†† 吉田　則裕††† 春名　修介† 井上　克郎†

† 大阪大学 大学院情報科学研究科 〒 565-0871 大阪府吹田市山田丘 1 番 5 号
†† 大阪大学 大学院国際公共政策研究科 〒 560-0043　大阪府豊中市待兼山町 1-31
††† 名古屋大学 大学院情報科学研究科 〒 464-8601 名古屋市千種区不老町
E-mail: †{t-saika,haruna,inoue}@ist.osaka-u.ac.jp, ††ejchoi@osipp.osaka-u.ac.jp, †††yoshida@ertl.jp

あらまし　Code Smell とは，ソースコードの設計上の問題を示す指標であり，リファクタリングが推奨されている．しかし，　実際に開発者がリファクタリングするのはどのような Code Smell かは明らかにされていない．本研究では，深刻度という問題の大きさの指標に着目して，Code Smell のメトリクスベースの深刻度とリファクタリングされる頻度の関係を調査した．結果から，長く複雑なソースコードを表す Code Smell である，Blob Class と Blob Operation について，深刻度の高いクラス・メソッドほど頻繁にリファクタリングされることが分かった．

キーワード　リファクタリング，不吉な匂い，ソフトウェア開発履歴，オブジェクト指向プログラミング，ソフトウェア保守

An Empirical Study of the Severity of Code Smell Using a Refactoring Dataset

Tsubasa SAIKA†, Eunjong CHOI††, Norihiro YOSHIDA†††, Shusuke HARUNA†, and Katsuro INOUE†

† Graduate School of Information Science and Technology, Osaka University, Yamadaoka 1-5, Suita, Osaka 565-0871 Japan
†† Osaka School of International Public Policy, Osaka University, Toyonaka, Osaka, 560-0043 Japan
††† Graduate School of Information Science, Nagoya University, Furo-cho, Chikusa-ku, Nagoya, 464-8601, Japan
E-mail: †{t-saika,haruna,inoue}@ist.osaka-u.ac.jp, ††ejchoi@osipp.osaka-u.ac.jp, †††yoshida@ertl.jp

Abstract　Code smells are structures in the code that suggest the possibility of refactoring. To prioritize code smells in large-scale source code, several tools for refactoring calculate the severity based on software metrics. Although several metrics are known as maintainability indicators, it is still unclear whether these severity indicators are in line with developer's perception. In this paper, we investigate whether developers focus on severe code smells. The result shows that developers focus on only particular types of severe smells.

Key words　Refactoring, Code Smell, Software development history, Object-oriented programming, Software maintenance

1. はじめに

リファクタリングとは，ソフトウェアの外部から見た振る舞いを変えずに，内部の構造を整理する作業のことである [1]．リファクタリングの主な目的は，ソースコードの可読性や保守性の向上である．また，リファクタリングを実施することで，機能の追加や，バグの発見が容易になり，ソフトウェア開発の効率を向上させることができる [2]．

リファクタリングの実施には明確な基準は無く，開発者はソースコードのどの部分にどの種類のリファクタリングを実施するかを判断する必要がある．しかし，リファクタリングの実施にかかるコストや，保守性への効果を事前に予測することは難しく，リファクタリングの実施の判断は開発者の経験によるところが大きい．

この問題を解決するために，Fowler は，リファクタリングの実施の判断材料として Code Smell を提案した [1]．Code Smell

とは, ソースコードの設計上の問題を示す指標であり, Code Smell が存在するソースコードにはリファクタリングの実施が推奨されている. 例えば, *God Class* は責務の多すぎる巨大なクラスを表す Code Smell であり, 保守性を悪化させる要因として知られている. この問題を解決するためには, 複数のクラスへ機能を分割するリファクタリングの実施が必要とされる.

現在, ソースコードから Code Smell を検出して開発者に提示することで, リファクタリングを実施するべきソースコードの特定を支援する Code Smell 検出ツールが数多く開発されている (例: inFusion, PMD) [3]. しかし, 開発者がリファクタリングを実施する Code Smell の基準は明確にされていないため, ツールが Code Smell を検出する基準と開発者がリファクタリングを実施する基準が大きく異なると, ツールは開発者にとって有益なリファクタリング実施対象の候補を提示できない. この問題に対して, ツールが検出する Code Smell が, 開発者のリファクタリング実施対象になっているかを調べる必要がある.

この問題に関して, Bavota らは各種類の Code Smell がリファクタリングの実施に与える影響を明らかにするため, リファクタリングが Code Smell を含むクラスに実施される割合と, リファクタリングによって Code Smell が完全に取り除かれる割合を調査した [4]. その結果, 42%のリファクタリングは Code Smell を含むクラスに対して実施されていた. しかし, リファクタリングの実施によって Code Smell が取り除かれたのはそのうちわずか 7%であった. そのため彼らはリファクタリングの実施と Code Smell の間に明確な関係は見られなかったとしている.

しかし, 彼らの研究では同種類の Code Smell の中で深刻度の違いについて考慮されていなかった. Code Smell の深刻度とは, Code Smell の示す問題の大きさを数値化したもので, Code Smell の検出に用いるメトリクス値を組み合わせて測定される. リファクタリングの実施にはコストがかかるため, Code Smell を修正するコストと効果は, リファクタリングを実施するかの判断に影響すると推測できる. この推測が正しければ, 同種類の Code Smell でも軽度なものと重度なものでは, リファクタリングが実施されるかの傾向が異なると考えられる. そのため, Code Smell の深刻度について考慮して調査を行う必要がある.

このような問題に対して, 本研究では, Java で書かれた 3 つのオープンソースソフトウェアのリファクタリング実施履歴を調査対象として, Code Smell 検出ツールが提示する深刻度という指標と, 開発中に実施されたリファクタリングの関係について調査を分析した. Code Smell の深刻度がリファクタリングの推薦に有用であるかを調べるため, 本研究では以下のリサーチクエスチョン (RQ) を設定している:

RQ1　深刻度の高い Code Smell がよりリファクタリングされるか？

RQ2　リファクタリングは Code Smell の深刻度を減少させるか？

調査の結果得られた, RQ への回答は以下の通りである:

- クラスレベルの Code Smell の *Blob Class* と, メソッドレベルの Code Smell*Blob Operation* と *Sibling Duplication* について, 深刻度の高いクラス・メソッドほど頻繁にリファクタリングされることが分かった.
- クラスレベルの Code Smell の *Blob Class* と, メソッドレベルの Code Smell の *Blob Operation* と *Internal Duplication* について, リファクタリングによって深刻度が減少する傾向にあった.

このことから, 開発者は特に *Blob Class* と *Blob Operation* の深刻度の高いクラス・メソッドに対して深刻度を減少させるようにリファクタリングを実施していると考えられる.

以降, 2. では調査対象のデータセットと調査手順を説明する. 3. では調査結果と考察を述べる. そして, 4. では本研究に関連する研究について述べる. 最後に, 5. では本研究のまとめと今後の課題を述べる.

2. 調 査 手 法

2.1 調査対象のデータセット

本研究では Java で書かれた 3 つのオープンソースソフトウェア, Xerces-J, ArgoUML, Apache Ant, の合計 64 リリースバージョンを調査対象とした. 本研究では, 各リリースバージョンのソースコードと, Bavota らの研究で検出されたリファクタリングの一覧を調査対象とする [4]. データセットの概要を表 1 に示す.

Bavota らは, Ref-Finder [5] というリファクタリング検出ツールを用いて, リリースバージョン間のソースコードの変化からリファクタリングを検出した. そして, Ref-Finder で検出されたリファクタリングのうち, それが本当にリファクタリングであるか確認出来たものを公開している. 本研究では彼らが正しく検出されたと判断したリファクタリングのみを調査対象とする. 検出されたリファクタリングの種類とそれぞれの検出数を表 2 に示す.

2.2 調 査 手 順

本研究では, Code Smell の深刻度がリファクタリングに与える影響を明らかにするため, リファクタリングされる Code Smell とリファクタリングされない Code Smell の間で, 深刻度について統計的な有意差があるかを調べる. 調査手順は図 1 に示す通り, 4 つの手順から構成される. 手順 1 では, 調査対象の各リリースバージョンについて, ソースコードから Code Smell の検出を行なう. 手順 2 では, リリース間で Code Smell を比較し, Code Smell の深刻度の増減を計測する. 手順 3 では, Code Smell の検出されたクラスやメソッドに対してリファクタリングが実施されたかどうかでグループ分けを行なう. 手順 4 では, 2 つのグループ間で RQ に対応した有意差検定を行なう. 以降, 手順 1 から手順 4 について順番に説明する.

（1） Code Smell の検出

調査対象の各リリースバージョンのソースコードから inFusion を利用して Code Smell の検出した. そして, 検出した Code Smell のうちクラス単位またはメソッド単位の Code Smell を, 検出元のクラス毎やメソッド毎に各種類の Code Smell の深刻度をまとめた.

表 1　調査対象のデータセットの概要

プロジェクト	対象期間	対象リリース	リリース数	クラス数	リファクタリング数	リファクタリングの種類数
Xerces-J	2003 年 10 月-2006 年 11 月	1.0.0-2.9.0	34	19,567	6,052	43
ArgoUML	2002 年 10 月-2011 年 4 月	0.10.1-0.32.2	12	43,686	3,423	43
Apache Ant	2003 年 8 月-2010 年 12 月	1.1-1.8.2	18	22,768	1,493	31
全体	-	-	64	86,021	10,968	52

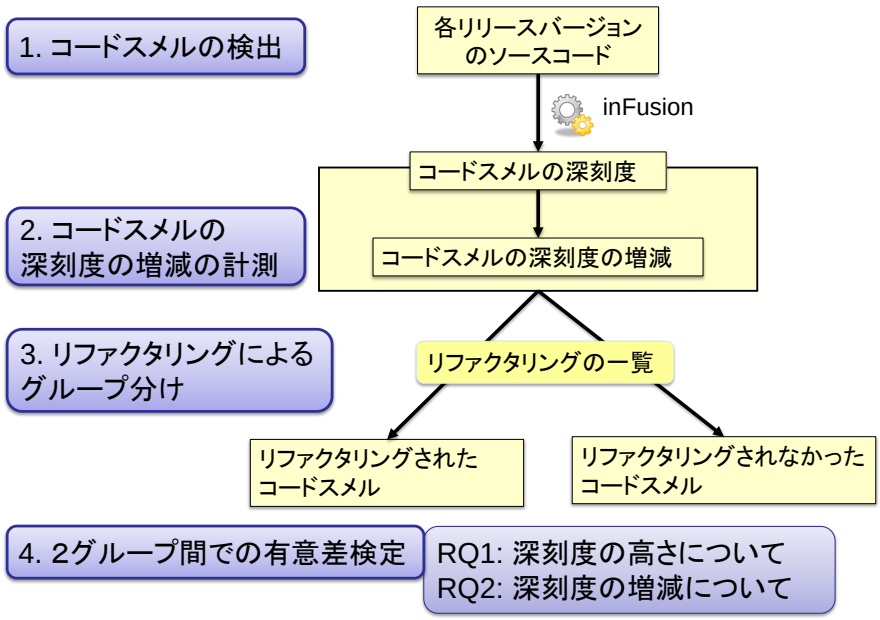

図 1　手順全体図

2.3　Code Smell の検出方法

本研究では inFusion という商用の Code Smell の検出ツールを利用し, 調査対象の各リリースバージョンのソースコードから Code Smell を検出した(注1). inFusion は Code Smell をソースコードのメトリクス値の特徴に基づいて抽出し, 24 種類の Code Smell を検出することが可能である. また, 検出された Code Smell には深刻度という, 1 から 10 までの整数値が付けられる.

inFusion を選択した理由は, 検出可能な Code Smell の種類の豊富なことと, 厳密な検出規則が定められていて検出の適合率の高いこと, そして深刻度によって Code Smell の重要さが表現されることである. また, inFusion は実際に企業等のソフトウェア開発現場で利用された実績があり, Yamashita らの研究でも利用されている [6].

本研究では inFusion の検出できる Code Smell のうち, 表 3 に示す 9 種類の Code Smell を調査対象とする. 各種類の Code Smell の詳細な説明と検出規則は, inFusion のマニュアルに掲載されている.

表 4 に調査対象の各システムについて Code Smell の検出数を種類毎に示す. 1 つのクラスまたはメソッドからは複数種類の Code Smell が検出されることがあるため, Code Smell を含むクラスの数と Code Smell を含まないクラスの数をそれぞれ示す.

（2） Code Smell の深刻度の増減の計測

RQ2 について, 連続するリリースバージョン間での Code Smell の深刻度の増減を計測した. 連続するリリースバージョン間で, 同じクラスの同じメソッド, 同じ種類の Code Smell を比較し, Code Smell の深刻度の増減を得た. Code Smell の深刻度の増減には, 増加または減少, 変化なしの 3 パターンがあり, 増加と減少については深刻度が変化した量も記録した. また, Code Smell がない状態の深刻度は 0 とし, Code Smell が除去された場合は深刻度の減少として扱う.

（3） リファクタリングによるグループ分け

有意差検定を行なうための準備として, Code Smell 検出されたクラスおよびメソッドを, リファクタリングが実施された群とリファクタリングが実施されなかった群に分けた. 本研究では Code Smell とは無関係に実施されたリファクタリングを除外するために, リファクタリングを Code Smell に対応するものに限定した. 本研究で扱う Code Smell を定義した Fowler [1] と Lanza [7] が, 各種類の Code Smell に対応するとしているリファクタリングの種類を表 5 に示す.

（4） 2 グループ間での有意差検定

リファクタリングが実施されたクラス群と, リファクタリングされなかったクラス群との間で, Code Smell の Code Smell の深刻度に有意差があるかどうかをマン・ホイットニーの U 検定で調べた. 検定のためのツールとしては, 統計分析ソフト R(注2) を

(注1) : http://www.intooitus.com/products/infusion

(注2) : https://cran.r-project.org

表 3　調査対象の Code Smell の種類の説明

Code Smell の種類	検出単位	説明
Blob Class	クラス	コードの量が多く複雑なクラス
Data Class	クラス	フィールドと getter, setter 以外の機能を持たないクラス
God Class	クラス	他のクラスと比べて責務の多過ぎるクラス
Tradition Breaker	クラス	親クラスの提供する機能を打ち消してしまうサブクラス
Blob Operation	メソッド	コードの量が多く複雑なメソッド
External Duplication	メソッド	他のクラスの複数のメソッドと著しく重複するメソッド
Feature Envy	メソッド	所属するクラスとは違うクラスのデータを多く用いるメソッド
Internal Duplication	メソッド	同じクラスの複数の他のメソッドと著しく重複するメソッド
Sibling Duplication	メソッド	継承階層で兄弟関係にあるクラスに重複するコードがあるメソッド

表 4　検出された Code Smell の数

Code Smell の種類	Xerces-J	ArgoUML	Apache Ant	全体
Blob Class	665	83	87	835
Data Class	71	3	28	102
God Class	952	160	143	1,255
Tradition Breaker	99	3	0	102
Blob Operation	2,428	545	413	3,386
External Duplication	713	269	46	1,028
Feature Envy	723	110	220	1,053
Internal Duplication	701	160	85	946
Sibling Duplication	927	545	131	1,603
Code Smell があるクラス数	3,907	1,992	897	6,796
Code Smell がないクラス数	15,660	41,694	21871	79,225
総クラス数	19,567	43,686	22,768	86,021

用いた. RQ1 については, リファクタリングされるクラスまたはメソッドの方が各種類の Code Smell 深刻度が有意に高いかを知るために, マン・ホイットニーの U 検定を片側検定で行なった. RQ2 については, リファクタリングされるクラスまたはメソッドの方が各種類の Code Smell 深刻度が有意に減少するかを知るために, マン・ホイットニーの U 検定を片側検定で行なった.

3. 調 査 結 果

a) RQ1：深刻度の高い Code Smell がよりリファクタリングされるか？

RQ1 に対するマン・ホイットニーの U 検定の結果を表 6 に示す. チェックマークは有意差 ($p < .05$) を示している. n/a と表記している部分は, その種類の Code Smell が検出されなかった場合も含めて, その種類の Code Smell のあるクラスまたはメソッドに対してリファクタリングが実行されなかったものである. また, 有意差以外の指標として効果量 r を載せている. 効果量はサンプルサイズによらない 2 グループの実施的な差を示すもので, 効果量 r の基準は r=0.1 なら効果量小, r=0.3 なら効果量中, r=0.5 なら効果量大とされている. この検定結果では, 有意差の出ている種類については全て効果量も小以上である.

結果から, クラスレベルの Code Smell の *Blob Class* と, メソッドレベルの Code Smell*Blob Operation* と *Sibling Duplication* について, 深刻度の高いクラスほどリファクタリングされる頻度が高い傾向にあることが分かった.

b) RQ2：リファクタリングは Code Smell の深刻度を減少させるか？

RQ2 に対するマン・ホイットニーの U 検定の結果を表 7 に示す. チェックマークは有意差 ($p < .05$) を示している. n/a と表記している部分は, その種類の Code Smell についてはリファクタリングの実施前後で深刻度が変化しなかったことを示している. また, 有意差以外の指標として効果量 r を載せている. 効果量はサンプルサイズによらない 2 グループの実施的な差を示すもので, 効果量 r の基準は r=0.1 なら効果量小, r=0.3 なら効果量中, r=0.5 なら効果量大とされている. この検定結果では, 有意差の出ている種類については全て効果量も小以上である.

結果から, クラスレベルの Code Smell の *Blob Class* と, メソッドレベルの Code Smell の *Blob Operation* と *Internal Duplication* について, リファクタリングされた Code Smell の深刻度が減少する, またはリファクタリングされない場合と比べて増加の幅が小さい傾向にあることが分かった.

以降, 調査結果から, Code Smell 検出ツールが開発者に優先的に提示するべき Code Smell についての考察を述べる. RQ1 と RQ2 の両方で有意差が出た Code Smell の種類は, クラスレベルの Code Smell の *Blob Class* とメソッドレベルの Code Smell の *Blob Operation* である. このことから, 開発者は特に *Blob Class* と *Blob Operation* の深刻度の高いクラス・メソッドに対して深刻度を減少させるようにリファクタリングを実施していると考えられる. そのため, *Blob Class* と *Blob Operation* の深刻度の高いクラス・メソッドを優先して提示するべきだと

表 2　調査対象のリファクタリングの数

リファクタリングの種類	Xerces	Argo	Ant	全体
Add Parameter	665	511	133	1309
Change Bidirectional Association To Undirectional	3	2	0	5
Change Undirectional Association To Bidirectional	6	0	0	6
Collapse Hierarchy	3	1	0	4
Consolidate Conditional Expression	171	45	32	248
Consolidate Duplicate Conditional Fragments	422	103	73	598
Decompose Conditional	0	2	1	3
Encapsulate Field	0	1	0	1
Extract Hierarchy	3	4	0	7
Extract Interface	78	40	10	128
Extract Method	166	135	73	374
Extract Subclass	6	4	0	10
Extract Superclass	2	13	3	18
Form Template Method	0	10	0	10
Hide Delegate	1	0	0	1
Hide Method	0	9	0	9
Inline Class	1	0	1	2
Inline Method	74	22	25	121
Inline Temp	86	98	55	239
Introduce Assertion	0	14	23	37
Introduce Explaining Variable	165	104	115	384
Introduce Local Extension	25	18	3	46
Introduce Null Object	35	25	2	62
Introduce Parameter Object	16	0	0	16
Move Field	920	399	67	1386
Move Method	747	349	96	1192
Parameterize Method	2	1	1	4
Preserve Whole Object	3	0	0	3
Pull Up Constructor Body	0	5	1	6
Pull Up Field	6	4	2	12
Pull Up Method	11	0	4	15
Push Down Field	52	0	0	52
Push Down Method	44	1	0	45
Remove Assignment To Parameters	73	40	49	162
Remove Control Flag	136	147	26	309
Remove Middle Man	0	1	0	1
Remove Parameter	496	442	114	1052
Rename Method	714	262	182	1158
Replace Conditional With Polymorphism	6	4	0	10
Replace Constructor With Factory Method	5	5	1	11
Replace Data With Object	32	10	6	48
Replace Delegation With Inheritance	1	0	0	1
Replace Eerror Code With Exception	1	0	0	1
Replace Exception With Test	38	18	18	74
Replace Magic Number With Constant	516	158	327	1001
Replace Method With Method Object	170	374	36	580
Replace Nested Conditional With Guard Clauses	124	33	13	170
Replace Parameter with Explicit Methods	3	1	0	4
Replace Parameter With Method	0	3	0	3
Replace Temp With Query	0	1	0	1
Self Encapsulate Field	7	2	0	9
Separate Query From Modifier	17	2	1	20
リファクタリングの総検出数	6052	3423	1493	10968

考えられる.

4. 関 連 研 究

Code Smell がリファクタリングに与える影響についての既存研究の 1 つとして, Bavota らはリファクタリングが Code Smell のあるクラスに実施される割合と, リファクタリングによって Code Smell が完全に取り除かれる割合を調査した [4]. 彼らは Code Smell の種類毎にリファクタリングに与える影響を分析し, リファクタリングと Code Smell の間に明確な関係は見られなかったとしているが, Code Smell の深刻度については考慮されていなかった. Code Smell の修正にもコストがかかるので, 深刻でない Code Smell は修正されずに放置されるやすいと推測できる. また, 深刻度の低い Code Smell が問題視されにくいとすると, 開発者は Code Smell を完全に取り除くのではなく, 深刻度を和らげることを目標にして修正を行なうと考えられる. そのため, 本研究では Code Smell の深刻度について考慮して調査を行なった.

リファクタリングと Code Smell の関係についての研究と近い内容で, リファクタリングとソフトウェア品質メトリクスの関係について研究が行なわれている [8]〜[11]. 本研究で調査した Code Smell の検出規則にはメトリクス値の条件に含まれているため, Code Smell の深刻度がリファクタリングに影響するように, メトリクス値の大小がリファクタリングに影響することが予想できる.

Szoke らの研究によると, 単一のリファクタリングは Code Smell などの保守性の問題をあまり改善しないが, 複数のリファクタリングを連続して実施することで保守性が大きく向上することができる [10]. また, 雑賀らの研究によると一度に複数種類のリファクタリングが組み合わせて実施される [12]. そのため, Code Smell の存在するクラスに対してどのようなリファクタリングが実施されるかを調査する場合には, 単一のリファクタリングではなくリファクタリングの組み合わせで調査するべき

表 5　Fowler または Lanza により Code Smell に対応付けられたリファクタリングの種類

Code Smell の種類	対応するリファクタリングの種類
Blob Class	Extract Interface
	Extract Subclass
	Extract Class
	Replace Data Value with Object
Data Class	Move Method
	Encapsulate Field
	Encapsulate Collection
God Class	Extract Interface
	Extract Subclass
	Extract Class
	Replace Data Value with Object
Tradition Breaker	Extract Class
	Pull Up Method
	Pull Up Field
Blob Operation	Extract Method
	Replace Method with Method Object
	Replace Temp with Query
	Decompose Conditional
External Duplication	Extract Method
	Extract Class
	Pull Up Method
	Form Template Method
Feature Envy	Extract Method
	Move Method
	Pull Up Method
	Move Field
Internal Duplication	Extract Method
	Extract Class
	Pull Up Method
	Form Template Method
Sibling Duplication	Extract Method
	Extract Class
	Pull Up Method
	Form Template Method

表 6　リファクタリングを Code Smell に対応する種類に限定した場合における, RQ1 に対するマン・ホイットニーの U 検定結果

Code Smell の種類	有意差	効果量 r
Blob Class	✓	0.158
Data Class		0.020
God Class		0.021
Tradition Breaker		0.025
Blob Operation	✓	0.049
External Duplication		0.075
Feature Envy		0.004
Internal Duplication		0.053
Sibling Duplication	✓	0.050

効果量 r の基準：r=0.1(効果量小), r=0.3(効果量中), r=0.5(効果量大)

表 7　リファクタリングを Code Smell に対応する種類に限定した場合における, RQ2 に対するマン・ホイットニーの U 検定結果

Code Smell の種類	有意差	効果量 r
Blob Class	✓	0.162
Data Class	n/a	
God Class		0.059
Tradition Breaker	n/a	
Blob Operation	✓	0.189
External Duplication		0.052
Feature Envy		0.110
Internal Duplication	✓	0.204
Sibling Duplication		0.071

効果量 r の基準：r=0.1(効果量小), r=0.3(効果量中), r=0.5(効果量大)

だと考えられる.

Yamashita らの研究によると, 複数種類の Code Smell が集中することは, 保守性を低下させたりバグの要因となるとされている [13]. 本研究では各種類の Code Smell の深刻度とその中の最大値について個別に調査したが, 複数種類の Code Smell の集中がリファクタリングにどのように影響するかは, 今後の研究課題である.

Tufano らの研究によると, あるクラスに Code Smell の発生するタイミングは, そのクラスが作成されたときの場合がほとんどである [14]. また, Kim らの研究によると, リファクタリングはリリースの直前に集中して実施される [15]. したがって, リリースバージョンのソースコードから Code Smell を検出した場合には, リリース間で発生した Code Smell の多くを見落としている可能性がある. 本研究ではリファクタリング検出の制限からリリースバージョン間に限定した調査を行なったが, 任意のリビジョンで調査可能なデータセットやリファクタリング検出ツールを探して調査する必要がある.

5. まとめと今後の課題

本研究では, 優先順位付きで開発者に Code Smell を提示するツールの開発のために, Code Smell の深刻度と開発者の実施するリファクタリングの関係を明らかにすることを目的として, 3 つの Java のオープンソースソフトウェアの合計 64 リリースバージョンを対象に調査を行なった.

調査の結果, クラスレベルの Code Smell の *Blob Class* と, メソッドレベルの Code Smell の *Blob Operation* について, 深刻度の高いクラスやメソッドほどリファクタリングされる頻度が有意に高く, リファクタリングによって Code Smell の深刻度が減少する傾向が見られた. よって, *Blob Class* と *Blob Operation* の深刻度の高いクラス・メソッドを優先して提示するべきだと考えられる.

今後の課題としては, 調査対象のシステムを更にを増やして調査結果の信憑性を向上し, 調査結果に基づいた優先順位を付けて Code Smell を提示するツールを開発する必要がある.

謝辞　本研究は JSPS 科研費 25220003，26730036, 15H06344 の助成を受けたものです.

文　献

[1] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison Wesley, 1999.
[2] G. Bavota, B.D. Carluccio, A.D. Lucia, M.D. Penta, R. Oliveto, and O. Strollo, “When does a refactoring induce bugs? an empirical study,” Proc. of SCAM, pp.104–113, 2012.
[3] F.A. Fontana, P. Braione, and M. Zanoni, “Automatic detection of bad smells in code: An experimental assessment.,” Journal of Object Technology, vol.11, no.2, pp.1–38, 2012.
[4] G. Bavota, A.D. Lucia, M.D. Penta, and R. Oliveto, “An experimental investigation on the innate relationship between quality and refactoring,” Journal of Systems and Software, vol.107, pp.1–14, 2015.
[5] M. Kim, M. Gee, A. Loh, and N. Rachatasumrit, “Ref-Finder: a refactoring reconstruction tool based on logic query templates,” Proc. of FSE, pp.371–372, 2010.
[6] A. Yamashita, M. Zanoni, F.A. Fontana, and B. Walter, “Inter-smell relations in industrial and open source systems: A replication and comparative analysis,” Proc. of ICSME, pp.121–130, 2015.
[7] M. Lanza and R. Marinescu, Object-Oriented Metrics in Practice, Springer-Verlag, 2006.
[8] K. Stroggylos and D. Spinellis, “Refactoring–does it improve software quality?,” Proc. of WoSQ, pp.1–6, 2007.
[9] G. Szoke, G. Antal, C. Nagy, R. Ferenc, and T. Gyimothy, “An empirical assessment of refactoring impact on software quality using a hierarchical quality model,” International Journal of Software Engineering and Its Applications, vol.5, no.4, pp.127–150, 2011.
[10] G. Szoke, G. Antal, C. Nagy, R. Ferenc, and T. Gyimothy, “Bulk fixing coding issues and its effects on software quality: Is it worth refactoring?,” Proc. of SCAM, pp.95–104, 2014.
[11] S.H. Kannangara and W.M.J.I. Wijayanake, “An empirical evaluation of impact of refactoring on internal and external measures of code quality,” International Journal of Software Engineering and Its Applications, vol.6, no.1, pp.51–67, 2015.
[12] T. Saika, E. Choi, N. Yoshida, A. Goto, S. Haruna, and K. Inoue, “What kinds of refactorings are co-occurred? an analysis of eclipse usage datasets,” Proc. of IWESEP, pp.31–36, 2014.
[13] A. Yamashita and L. Moonenl, “Exploring the impact of inter-smell relations on software maintainability: An empirical study,” Proc. of ICSE, pp.682–691, 2013.
[14] M. Tufano, F. Palomba, G. Bavota, and R. Oliveto, “When and why your code smell starts to smell bad,” Proc. of ICSE, pp.403–414, 2015.
[15] M. Kim, D. Cai, and S. Kim, “An empirical investigation into the role of api-level refactorings during software evolution,” Proc. of ICSE, pp.151–160, 2011.