修士学位論文

題目

コードスメルの深刻度がリファクタリングの実施に与える影響の実証的研究

指導教員
井上 克郎 教授

報告者
雜賀 翼

平成 28 年 2 月 9 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

平成 27 年度 修士学位論文

コードスメルの深刻度がリファクタリングに与える影響の調査

雑賀 翼

内容梗概

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, 内部構造を整理する作業である. リファクタリングはソフトウェアの保守性を向上させることを目的として実施され, 機能の追加やバグの修正などが容易になる事が期待できる.

リファクタリングの実施には明確な基準がないため, 開発者がリファクタリングの実施を判断することが難しい. この問題点を解決するため, コードスメルという, ソースコードの設計上の問題を示す指標を検出し, リファクタリング対象の候補として開発者に提示する, コードスメル検出ツールが開発されている.

しかし, 大規模なソフトウェアなどでコードスメルが大量に検出される場合, 開発者はどのコードスメルかを優先的に除去するかの判断が困難である. そのため, コードスメルに優先順位を付けた提示が必要である. 既存のコードスメル検出ツールの 1 つである inFusion では, コードスメルの検出に用いるメトリクス値の大きさからコードスメルの深刻度を定義し, 優先順位を決めている. しかし, コードスメルの深刻度とリファクタリングとの関係が明らかになっていないため, 優先的に提示されるコードスメルが開発者の認識とは合わない恐れがある.

このような問題に対して, 実際に開発者が実施するリファクタリングとその対象になるコードスメルの深刻度を調査し, その結果に基づいて, 開発者にとって有用なコードスメルを優先的に提示するツールの開発が必要である. そのため, 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し, コードスメルの深刻度とリファクタリングとの関係について分析した. その調査の結果, クラスレベルのコードスメルの Blob Class と God Class, Schizopherenic Class, Tradition Breaker と, メソッドレベルのコードスメル Blob Operation 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有意に高いことが分かった.

主な用語

リファクタリング
コードスメル
オブジェクト指向プログラミング
ソフトウェア保守

目 次

1　まえがき

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 [7]. 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 [1].

しかし, 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コードスメルを提案した [7]. コードスメルとは, ソースコードの設計上の問題を示す指標であり, コードスメル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コードスメルであり, 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, ソースコードからコードスメル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コードスメル検出ツールが数多く開発されている (例: inFusion, PMD)[5]. しかし, 大規模なソフトウェアなどでコードスメルが大量に提示される場合, 開発者はどのコードスメルを優先して除外するかを選択する必要があるが, ツールはコードスメルの存在する箇所と問題の種類が示すだけのものが多く, その選択に有用な情報は与えられない.

inFusion, inCode などのツールは, 関連するソフトウェア品質メトリクスの値の大きさからコードスメルの深刻度を定義し, コードスメルの優先順位を決めている. ソフトウェア品質メトリクスは保守性を定量的に評価 [9, 14] や, リファクタリング対象の候補の特定のために用いられる [17]. しかし, コードスメルの深刻度とリファクタリングとの関係は明らかになっていないため, ツールが決める優先順位が開発者が実施するリファクタリングと合わない恐れがある.

このような問題に対して, 実際に開発者が実施するリファクタリングとその対象になるコードスメルの深刻度を調査し, その結果に基づいて開発者にとって有用なコードスメルを優先的に提示するツールの開発が必要である. 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, コードスメル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.

もし開発者が深刻度の高いコードスメルに対して頻繁にリファクタリングを実施するならば, 深刻度の高いコードスメルを優先的にリファクタリング候補として提示することが必要である. さらに, コードスメルの深刻度がリファクタリングによって減少するならば, 深刻度の高いコードスメルに対して, 対応するリファクタリングの実施を優先的に推薦することが考えられる.

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 深刻度の高いコードスメル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コードスメル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:

- 一部のクラスレベルのコードスメルについて, 深刻度の高いクラスほど頻繁にリファクタリングされることが分かった. このことから, 深刻度の高いコードスメルを優先的に提示する支援は, 特定のリファクタリングの候補を探すのに有用だと考えられる.
- 一部のメソッドレベルのコードスメルについては, 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.

以降, 2 節では背景として, 本研究に関連する用語と既存研究について述べる. 5.2 節では調査対象のデータセットと調査手順を説明する. そして, 4 節ではその調査結果を説明し, 5 節では考察を行なう. 最後に 6 節では本研究のまとめと今後の課題を述べる.

2　背景

この節では, 本研究の技術的な背景と, 関連する既存研究について説明する. まず, 2.1 節ではリファクタリングについて説明し, 2.2 節でソフトウェア開発履歴からリファクタリングの実施事例を検出する手法を紹介する. 次に, 2.3 節で, コードスメルについて説明し, 2.4 節でソースコードからコードスメルを検出する手法を紹介する. 最後に, 2.5 節では, コードスメルとリファクタリングの関係を調査した既存研究について述べる.

2.1　リファクタリング

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 [7]. つまり, リファクタリングはソフトウェアの機能そのものは変更せず, ソースコードの可読性の向上のためにソフトウェアの構造を変化させる事である.

オブジェクト指向言語で開発されたソフトウェアならばリファクタリングを実施することによって, オブジェクト指向の利点であるコードの再利用性を高めることが出来る [21].

ソースコードは時間につれて, 機能の追加やバグの修正などの保守作業によって, 本来の設計から離れたものになる恐れがある. リファクタリングの実施によりソースコードを読みやすくし, 機能の追加やバグの修正などの保守作業の効率を向上することができる. また, プログラムの劣化を防ぐことができる [7].

また, リファクタリングの実施によってソフトウェアの設計を後から改善できる. 設計の良し悪しは開発の効率にも関係するため, リファクタリングの実施によって設計を改善することで, ソフトウェア開発の効率を向上させることができる [7].

以下, 代表的なリファクタリングの種類をいくつか紹介する [6].

Move Method/Field (メソッド/フィールドの移動)　対象のメソッドまたはフィールドを定義しているクラスとは別のクラスへと移動する.

Extract Class/Method/Field(クラス/メソッド/フィールドの抽出)　コードの一部を抽出して, 新しいクラスやメソッド, フィールドとする.

Encapsulate Collection (コレクションのカプセル化)　単純な getter や setter を削除し, 目的毎に分離したメソッドに置き換える.

Replase Inheritence With Delegation (委託による継承の置き換え)　クラスの継承関係を委託関係に置き換える.

Introduce Parameter Object (引数オブジェクトの導入)　複数の引数を一つのオブジェクトにまとめる.

Hide Delegate (委託の隠蔽)　あるオブジェクトの委託クラスを呼び出しているときに, 委託を継承に置き換えることで連鎖的なオブジェクトの呼び出しを解消する.

2.2　リファクタリング検出ツール

リファクタリング検出ツールとは, 版管理システム等に記録されたソフトウェアの開発履歴から, リファクタリングの実施事例を自動検出するツールのことである. 現在までに多くのリファクタリング検出ツールが提案されている [15, 3]. ここでは代表的なリファクタリング検出ツール Ref-Finder [12] について説明する.

Ref-Finder は, 版管理システムに記録された 2 つのバージョン間におけるソースコードの変化から, リファクタリングの実施された事例を自動的に検出するツールである. 多くのソフトウェア開発では版管理システムを用いてソースコードの変更履歴を記録しているため, Ref-Finder のように変更履歴解析に基づいたリファクタリング検出ツールは, 幅広いプロジェクトに適用できる. Ref-Finder は 2 つのバージョンのソースコードから各々の構造情報を抽出し, それらの差分を計算する. その差分がリファクタリングの検出規則 [16] と一致する場合に, リファクタリングの実施事例と見なして検出し, リファクタリングの種類とソースコードの位置情報を出力する. Ref-Finder は 63 種類のリファクタリングを検出することができる.

2.3　コードスメル

コードスメルとは, コードの設計について問題点を示す指標である. コードスメルはソフトウェアの開発や保守を困難にする要因の一つであり, Fowler によってリファクタリングの実施が推奨される状況として提案された [7]. コードスメルにはクラスレベルのものやメソッドレベルものがある. 効果的なリファクタリングを実施するには開発者の経験が必要なために, コードスメルによってリファクタリングを実施する対象の特定を支援することが提案されている. しかし, コードスメルは定量的な定義はされてない. また, コードスメルを含むソースコード全てにリファクタリングを実施する必要はない. そのため, 開発者はコード中のコードスメルの存在する箇所について, コードスメルの示す設計上の問題を取り除くべきかどうかをまず検討し, そしてコードスメルを除去するのに効果的なリファクタリングを実施する必要がある.

最初に Fowler が 22 種類のコードスメルを提案した [7]. その後, Emden らや Lanza らなどもコードスメルの種類を提案した [4, 13]. 今後も新しい種類のコードスメルが提案される

可能性もある．また, コードスメルの種類の分類と, コードスメルの種類間の関係についても研究されている [13, 22].

本研究で扱うコードスメルを以下に示す．Fowler の本または Lanza の本で紹介されているコードスメルについては, 対応するリファクタリングの例を紹介する [7, 13]．また, コードスメルの検出される単位, クラスレベルとメソッドレベルに分けて紹介する．

まず, クラスドレベルのコードスメルの代表例を以下に載せる．

Blob Class: クラスレベル　コードの量が多く複雑なクラス．対応するリファクタリングとしては, Move や Extract などで一部の機能を他のクラスを移すことが提案されている．

Data Class: クラスレベル　フィールドとそれに対する getter と setter だけを持ち, それ以外の機能を持たないクラス．他のクラスがカプセル化されていないデータにアクセスするため, 脆弱な設計である．また, 複数のクラスが getter と setter を用いるために保守が難しい．対応するリファクタリングとしては, Encapsulate Collection で単純な getter や setter を目的毎に分離して, データに対する処理をクラス内で行なうように変えることが提案されている．

Distorted Hierarchy: クラスレベル　狭く深い継承階層にあるクラス．深い入れ子構造が理解を難しくし, バグを発生させやすい．

God Class: クラスレベル　他のクラスと比べて責務の多過ぎるクラス．機能の分割が不適切だと考えられる．対応するリファクタリングとしては, Move や Extract などで一部の機能を他のクラスを移すことが提案されている．

Refused Parent Bequest: クラスレベル　親クラスから継承したものの一部しか利用しないサブクラス．メソッドのオーバーライドや protected メンバへのアクセスがなく, 継承が不適切だと考えられる．対応するリファクタリングとしては, Replase Inheritence With Delegation で継承を委託に置き換えることが提案されている．

Schizophrenic Class: クラスレベル　互いに関係のない機能を持ち, 凝集度が低いクラス．

Tradition Breaker: クラスレベル　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．空のメソッドによるオーバーライドなどを行なっていて, 継承が適切でないと考えられる．

次に, メソッドレベルのコードスメルの代表例を以下に載せる．

Blob Operation: メソッドレベル　コードの量が多く複雑なメソッド. このメソッドにクラスの機能が集中し過ぎている場合が多い. 対応するリファクタリングとしては, Extract でメソッドの一部を新たなメソッドとして抽出することが提案されている.

Data Clumps: メソッドレベル　複数個所で一緒に現れるデータ群をパラメータに持つメソッド. データを上手く表現できていないと考えられる. 対応するリファクタリングとしては, Introduce Parameter Object でパラメータをまとめることが提案されている.

External Duplication: メソッドレベル　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. 重複するコードに対する変更は管理が難しく, バグを発生させやすい. 対応するリファクタリングとしては, Extract で重複するコードを抽出してまとめることが提案されている.

Feature Envy: メソッドレベル　定義されたクラスのデータよりも他のクラスのデータを多く用いるメソッド. このメソッドは他のクラスで定義されるべきだと考えられる. 対応するリファクタリングとしては, Move でメソッドを適切なクラスに移動することが提案されている.

Intensive Coupling: メソッドレベル　他のクラスのメソッドの呼び出しが多過ぎるメソッド. 結合度が高く, コードが複雑になる. 他のクラスに新しくメソッドを作成し, このメソッドに対応した機能をそこにまとめて, 1 つのクラスから呼び出すメソッドの数を減らすことが提案されている.

Internal Duplication: メソッドレベル　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. 重複するコードに対する変更は管理が難しく, バグを発生させやすい. 対応するリファクタリングとしては, Extract で重複するコードを抽出してまとめることが提案されている.

Message Chains: メソッドレベル　他のクラスからオブジェクトを取得し, そのオブジェクトから更に他のクラスのオブジェクトを所得する, オブジェクトの連鎖的な呼び出しを行うメソッド. 関係するクラス全てと結合度が高く, 変更が複数クラスに影響する. 対応するリファクタリングとしては, Hide Delegate で中間のクラスを継承関係に置き換えることで, 連鎖を解消することが提案されている.

Shotgun Surgery: メソッドレベル　変更を行なうと複数のクラスに変更が必要になるメソッド. 多くの他のクラスのメソッドを呼び出すまたは呼び出される. 対応するリファクタリングとしては, Move で関係するメソッドを一つのクラスに移動してまとめることが提案されている.

Sibiling Duplication: メソッドレベル　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. 重複するコードに対する変更は管理が難しく, バグを発生させやすい. 対応するリファクタリングとしては, Extract で重複するコードを抽出してまとめることが提案されている.

2.4　コードスメル検出ツール

現在, 多くのソースコード品質分析ツールが, コードスメルを検出することにより, 開発者へリファクタリング実施対象の特定を支援している [5]. しかし, コードスメルは明確に定義されていないため, それぞれのコードスメル検出ツールは異なる種類のコードスメルを検出するだけでなく, 同じ種類のコードスメルに対しても, 異なった解釈に基づいてコードスメルを検出する. 多くのコードスメル検出ツールは, コードスメルに関連するソフトウェア品質メトリクスを用いた検出規則に基づいてコードスメルを検出する. 各種類のコードスメルを複数のメトリクスを組み合わせて表現するが, ツールによって用いるメトリクスの種類や閾値の設定は異なる.

ツールの出力結果には共通してコードスメルの種類と検出元のクラスの情報が含まれる. また, それに加えてコードスメルの優先順位付けをツール (例: inFusion) や, リファクタリングの候補の推薦を行うツール (例: JDeodorant) もある. 以下, いくつかのコードスメル検出ツールを例として紹介する.

inFusion　inFusion[1] は商用のソースコード品質分析ツールで, Java または C, C++で書かれたソースコードから 24 種類のコードスメル [2] を検出できる. inFusion は, コードスメルに関連するソフトウェア品質メトリクスの値について閾値を段階的に設定し, メトリクス値の大きさからコードスメルの深刻度という数値を定義している. そのため, 開発者はコードスメルの深刻度を コードスメルの優先順位付けのためにに利用することが出来る.

JDeodorant　JDeodorant[3] はオープンソースのコードスメル検出ツールで, Java で書かれたソースコードから 4 種類のコードスメルを検出できる. このツールは統合開発環境

[1] http://www.intooitus.com/products/infusion
[2] https://www.intooitus.com/products/infusion/detected-flaws
[3] http://users.encs.concordia.ca/~nikolaos/jdeodorant

Eclipse のプラグインとして開発されているため, Eclipse と連携する機能を持つ. そのため, JDeodorant は検出したコードスメルに対応するリファクタリング候補の推薦を行うことが出来る.

PMD　PMD[4] はオープンソースのソースコード品質分析ツールで, Java で書かれたソースコードから 5 種類のコードスメルを検出可能である. このツールは, Eclipse や JDeveloper, JEdit など様々な開発環境で利用できる.

2.5　コードスメルとリファクタリングの関係に関する研究

本節ではコードスメルとリファクタリングの関係を調査した既存研究を説明する.

2.5.1　コードスメルを用いたリファクタリング候補の推薦

コードスメルは定量的な定義はされてないため, コードスメルの検出ツールによってコードスメルの解釈やメトリクス値の閾値が異なる. また, 開発者によってリファクタリングを実施する基準が異なる. ツールがコードスメル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. この問題に関して, ツールが検出するコードスメルと実際の保守作業の関係を明らかにするため, コードスメルがソフトウェア保守に与える影響について研究されている.

Sjoberg らはコードスメルが保守作業にかかる時間に影響するかを, 実験的なソフトウェアの保守作業を通して調べた [18]. しかし, その結果からはコードスメルが作業時間に与える明確な影響は見られなかった.

また, Bavota らは各種類のコードスメル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コードスメル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コードスメルが完全に取り除かれる割合を調査した [2]. その結果, 42%のリファクタリングはコードスメル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コードスメル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コードスメルの間に明確な関係は見られなかったとしているが, 彼らの研究ではコードスメルの深刻度については考慮されていなかった.

リファクタリングの実施にはコストがかかるため, コードスメル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, 同種類のコードスメル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

[4] https://pmd.github.io

の傾向が異なると考えられる．そのため, コードスメルの深刻度について考慮して調査を行う必要がある．

2.5.2 コードスメルを用いたリファクタリングの効果測定

一般的に, リファクタリングはソフトウェアの保守性を向上できると言われている．しかし, コードスメルやメトリクスで表されるようなソフトウェアの品質へ与える影響に関しては明らかにされていない．リファクタリングの実施にはコストがかかり, リファクタリングの実施がソフトウェアの品質を向上するをする根拠を示すことができれば, リファクタリングをする明確な動機付けになる．そのため, リファクタリングの効果をメトリクスやコードスメルで測定する実証的研究が数多く行われている [11, 20, 10, 2]．そのうち多くの研究では, 大半のメトリクスやコードスメルについてはリファクタリングとの間に明確な関係が見られず, 一部のメトリクスやコードスメルについてのみリファクタリングによる改善が見られた．また, それぞれ異なるメトリクスやコードスメルを用いて別々のソフトウェアを調査しているが, 一貫した結果は得られておらず, いまだリファクタリングのソフトウェア品質への効果は明らかになっていない．

この原因として, 既存研究ではコードスメルについては取り除かれることを条件にしているが, リファクタリングと同時にバグ修正や機能の追加等の修正も行われる場合が多いので, リファクタリングの効果はコードスメルを完全に取り除くまで至らず, 深刻度を減少するだけに留まることが考えられる．

3　調査手法

コードスメルに優先順位を付けて提示するツールの開発を目的として, コードスメルの深刻度とリファクタリングとの関係を明らかにするため, 本研究では以下の 2 つのリサーチクエスチョン (RQ) を設定している.

RQ1 深刻度の高いコードスメル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コードスメルの深刻度を減少させるか？

リファクタリングの実施にはコストがかかるため, コードスメル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, 同種類のコードスメル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, 本研究ではコードスメルの深刻度について調査を行う.

RQ1 の目的は, コードスメル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査することである. 2.5.1 節で述べたように, ツールが検出するコードスメルは開発者の考えるコードスメルに合ったものでないと有用ではない. 本研究では, コードスメルの深刻度がリファクタリングの実施に影響するかを調査する.

RQ2 の目的は, 実施されたリファクタリングがコードスメルに関係あるかを確かめることと, リファクタリングの効果を測定することである. 2.5.2 節で述べたように, リファクタリングがコードスメルを取り除くかどうかは既存研究でも調査されているが, コードスメルの深刻度を考慮した研究はまだ行なわれていない. 本研究ではリファクタリング前後のコードスメルの増減を計測することで, リファクタリングの効果を調べる.

本研究の調査対象は **Xerces-J (Xerces)** [5] と **ArgoUML (Argo)** [6], **Apache Ant (Ant)** [7] という, Java で書かれた 3 つ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である. これらは Bavota らの研究で調査対象となったプロジェクトである [2]. 彼らが検出した信頼性の高いリファクタリングの一覧が利用可能であるため, 本研究はこれらのプロジェクトを調査対象として選択した. 各プロジェクトの概要を表 1 に示す. 表の最後の行は 3 つのプロジェクトの合計を示している.

本研究では RQ に回答するために, リファクタリングされるコードスメルとリファクタリングされないコードスメルの間で, 深刻度について統計的な有意差があるかを調べる. RQ1 に対してはコードスメルの深刻度の高さに有意差があるかを調べ, RQ2 に対しては連続する

[5] http://xerces.apache.org/xerces-j/
[6] http://argouml.tigris.org/
[7] http://ant.apache.org/

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
Xerces-J	2003 年 10 月-2006 年 11 月	1.0.0-2.9.0	34	19,567
ArgoUML	2002 年 10 月-2011 年 4 月	0.10.1-0.32.2	12	43,686
Apache Ant	2003 年 8 月-2010 年 12 月	1.1-1.8.2	18	22,768
全体	-	-	64	86,021

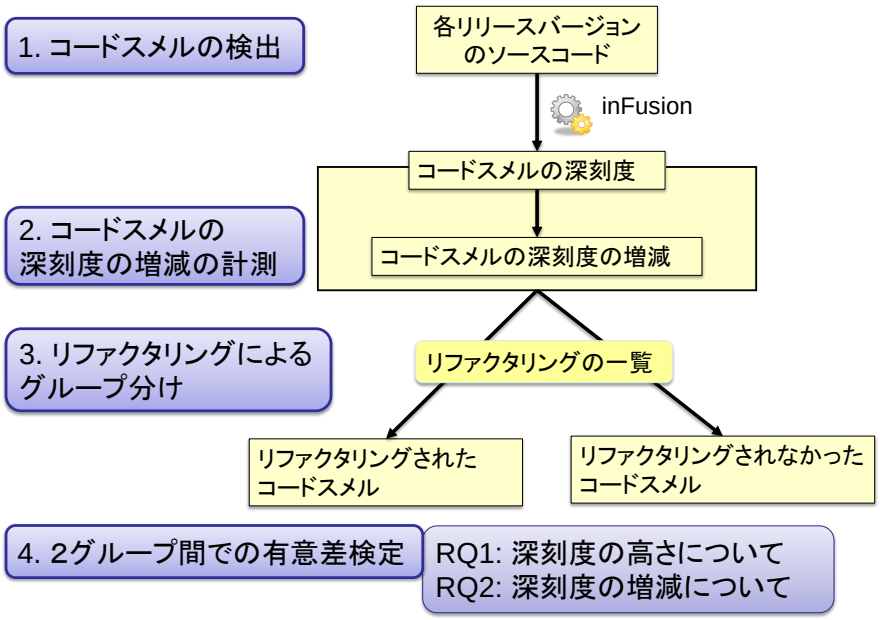

図 1: 手順全体図

リリースバージョン間でのコードスメルの深刻度の増減に有意差があるかを調べる. 有意差検定を行なう手順は図 1 に示す通り, 4 つの手順から構成される. まず, 手順 1 では, 調査対象の各リリースバージョンについて, ソースコードからコードスメルの検出を行なう. 手順 2 では, リリース間でコードスメルを比較し, コードスメルの深刻度の増減を計測する. 手順 3 では, コードスメル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.

以下の各節で手順 1 から手順 4 について説明する.

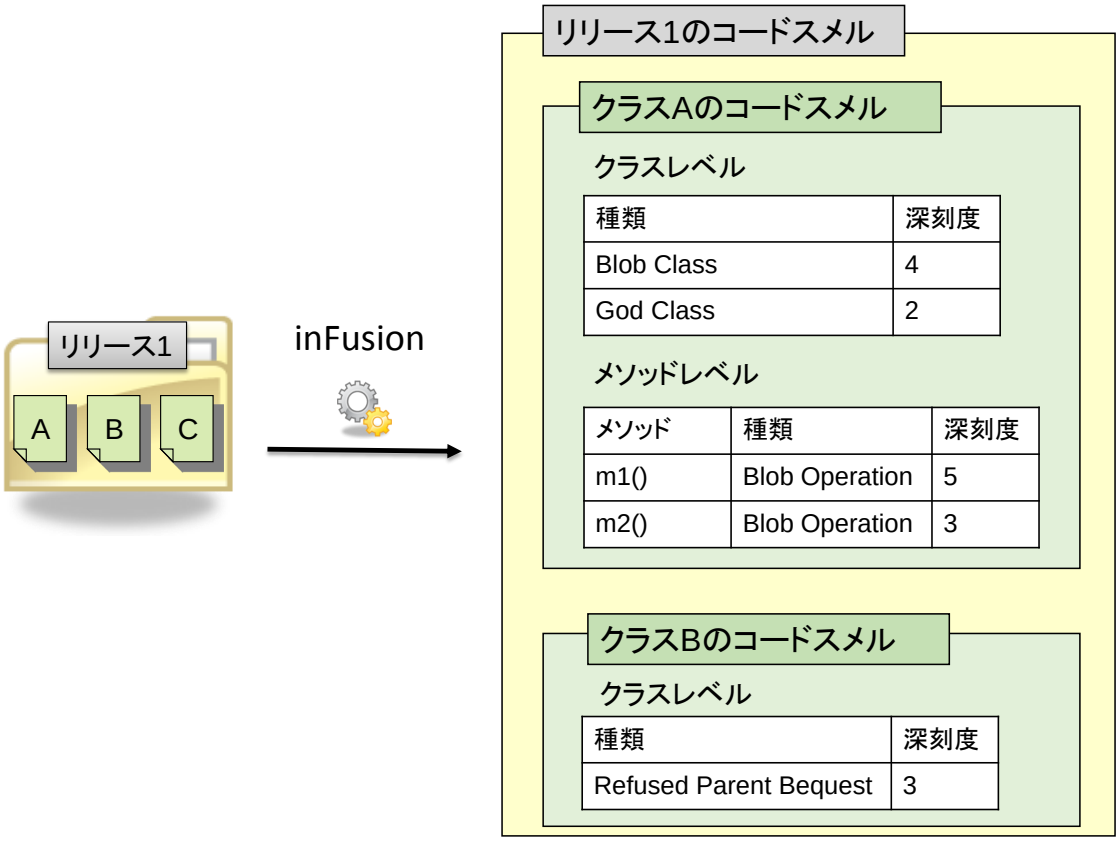

図 2: コードスメルの検出例

3.1 手順 1: コードスメルの検出

調査対象のプロジェクトの各リリースバージョンについて, ソースコードからコードスメルの検出を行う. 検出したコードスメルは図 2 の例に示すように, クラスレベルのコードスメルは検出元のクラス毎に, メソッドレベルのコードスメルはクラスのメソッド毎にまとめる. これにより, 各クラスおよび各メソッドについて各種類のコードスメルとその深刻度の一覧が得られる.

本研究ではソースコードからコードスメルを検出するために, inFusion[8] という商用のコードスメル検出ツールを利用した. inFusion はソースコードのメトリクス値の特徴に基づいて, 24 種類のコードスメルを検出することが可能である. また, 検出されたコードスメルには深刻度という, 問題の大きさを表す 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, 検出可能なコードスメルの種類の豊富なことと, 厳密な検出規則が定められて

[8] http://www.intooitus.com/products/infusion

いて検出の適合率の高いこと, そして深刻度によってコードスメル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 [22]. 本研究では inFusion の検出できるコードスメルのうち, Java のクラス単位またはメソッド単位で検出される 16 種類のコードスメルコードスメルを調査対象とする. 各種類のコードスメル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 2 に調査対象の各プロジェクトについてコードスメル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コードスメルが検出されることがあるため, コードスメルを含むクラスの数とコードスメルを含まないクラスの数をそれぞれ示す.

3.2 手順 2: リリースバージョン間でのコードスメルの比較

RQ2 について, 連続するリリースバージョン間でのコードスメルの深刻度の増減を得るため, クラス毎とメソッド毎に連続するリリースバージョン間で各種類のコードスメルの深刻度を比較する.

コードスメルの増減の例を図 3 に示す. 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, 同じ種類のコードスメルを比較して, コードスメルの深刻度の増減を得る. コードスメル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する. この例では, クラス A のクラスレベルの *God Class* の深刻度が増加している. また, 新たにクラス A のメソッド m3() に *Blob Operation* が検出されている. このように, 前のリリースバージョンでは検出されなかったコードスメルが, 新規に検出された場合も深刻度の増加に分類する. コードスメルがない状態の深刻度は 0 とする. 一方, クラス A のメソッド m1() の *Blob Operation* は深刻度が減少している. また, クラス A のメソッド m2() の *Blob Operation* とクラス B の *Refused Parent Bequest* はリリース 2 では検出されていない. このように, 除去されたコードスメルについても深刻度の減少に分類する. そして, クラス A の *Blob Class* は深刻度の変化なしである.

3.3 手順 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, コードスメル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る.

本研究では Bavota らの研究で検出されたリファクタリングをもとにグループ分けを行なう [2]. 彼らは, Ref-Finder[12] 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. しかし, Ref-Finder の検出結果には, 誤検出が大量に含まれるという問題点が指摘されている [19]. これに対して,

表 2: 検出されたコードスメルの数

コードスメルの種類	Xerces	Argo	Ant	All
Blob Class	665	83	87	835
Data Class	71	3	28	102
Distorted Hierarchy	9	0	0	9
God Class	952	160	143	1,255
Refused Parent Bequest	114	14	81	209
Schizophrenic Class	39	48	4	91
Tradition Breaker	99	3	0	102
Blob Operation	2,428	545	413	3,386
Data Clumps	834	434	54	1,322
External Duplication	713	269	46	1,028
Feature Envy	723	110	220	1,053
Intensive Coupling	476	463	132	1,071
Internal Duplication	701	160	85	946
Message Chains	19	9	1	29
Shotgun Surgery	39	10	34	83
Sibling Duplication	927	545	131	1,603
コードスメルを含むクラス数	1,949	311	343	2,603
コードスメルを含まないクラス数	17,618	43,375	22,425	83,418
全てのクラス数	19,567	43,686	22,768	86,021

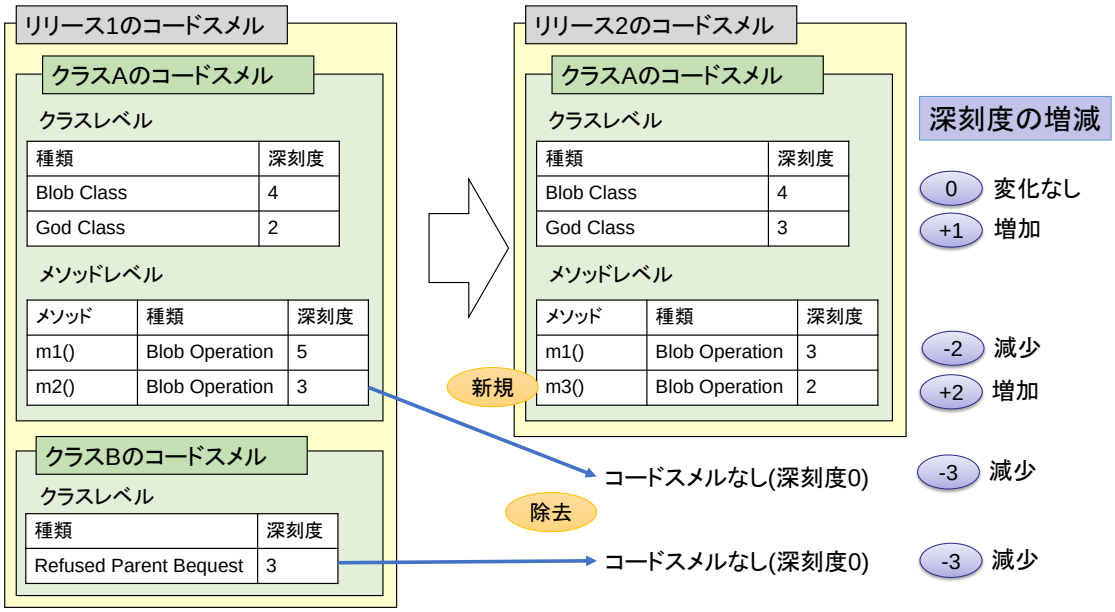

図 3: コードスメルの増減の例

Bavotara は Ref-Finder で検出されたリファクタリング 1 つ 1 つについて, それが本当にリファクタリングであるかを目視で確認しているため, 信頼できる検出結果だと考えられる. 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 3 に示す. 検出されたリファクタリング 1 つ 1 つについての情報は, リファクタリングの種類と, どのリリースバージョンのどのクラスから検出されたかの情報のみである. Ref-Finder の検出結果にはリファクタリングについて, リファクタリングの対象となったメソッドなどの詳細な情報が含まれるはずであるが, Bavota らの検出したリファクタリングについては公開されていない. そのため, 本研究では Bavota らの検出したリファクタリングの対象となったメソッドを特定するために, Ref-Finder を用いて Bavota らと同じデータセットからリファクタリングを検出し, その検出結果と Bavota らの検出したリファクタリングの対応付けを行った.

そして, 全てのコードスメルを含むクラスおよびメソッドについて, そのクラスおよびメソッドを対象として実施されたリファクタリングの有無から, 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. グループ分けを行なった結果の両群のコードスメルの数を 4 に示す.

表 3: 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

表 4: リファクタリングによるグループ分け後のコードスメルの数

コードスメルの種類	リファクタリングされたクラス・メソッド数	リファクタリングされなかったクラス・メソッド数
Blob Class	59	633
Data Class	8	1671
Distorted Hierarchy	0	8
God Class	25	1269
Refused Parent Bequest	0	538
Schizophrenic Class	0	311
Tradition Breaker	2	247
Blob Operation	107	4503
Data Clumps	0	1402
External Duplication	3	1653
Feature Envy	9	1355
Intensive Coupling	0	1290
Internal Duplication	8	2210
Message Chains	0	31
Shotgun Surgery	0	65
Sibling Duplication	11	1945

3.4 手順 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群と, リファクタリングされなかった群との間で, コードスメルの深刻度について統計的な有意差があるかどうかを, マン・ホイットニーの U 検定という, ノンパラメトリックな有意差検定の 1 つを用いて調べた. 検定のためのツールとしては, 統計分析ソフト R[9] を用いた. 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コードスメル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コードスメル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.

[9] https://cran.r-project.org

4　調査結果

本節では本研究の 2 つの RQ に回答するために行なった調査の結果について述べる.

RQ1：深刻度の高いコードスメルがよりリファクタリングされるか？　RQ1 について, リファクタリングが実施されたクラスまたはメソッドは, リファクタリングされなかったクラスまたはメソッドと比べて, 各種類のコードスメルの深刻度が有意に高いかどうかを, マン・ホイットニーの U 検定を用いて調べた結果を表 5 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, その種類のコードスメルが検出されなかった場合も含めて, その種類のコードスメル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, クラスレベルのコードスメルの *Blob Class* と *God Class*, *Schizopherenic Class*, *Tradition Breaker* と, メソッドレベルのコードスメル *Blob Operation* 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.

RQ2：リファクタリングはコードスメルの深刻度を減少させるか？　RQ2 について, リファクタリングが実施されたクラスは, リファクタリングされなかったクラスと比べて, 各種類のコードスメル深刻度が有意に減少するかどうかを, マン・ホイットニーの U 検定を用いて調べた結果を表 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, その種類のコードスメル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, メソッドレベルのコードスメルの *Blob Operation* と *Feature Envy*, *Intensive Coupling*, *Internal Duplication* について, リファクタリングされたコードスメル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.

表 5: RQ1 に対するマン・ホイットニーの U 検定結果

コードスメルの種類	有意差	効果量 r
Blob Class	✓	0.290
Data Class		0.042
Distorted Hierarchy	n/a	
God Class	✓	0.179
Refused Parent Bequest		0.033
Schizophrenic Class	✓	0.163
Tradition Breaker	✓	0.195
Blob Operation	✓	0.126
Data Clumps		0.035
External Duplication		0.028
Feature Envy		0.047
Intensive Coupling		0.024
Internal Duplication		0.025
Message Chains		0.154
Shotgun Surgery		0.010
Sibling Duplication		0.005

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 6: RQ2 に対するマン・ホイットニーの U 検定結果

コードスメルの種類	有意差	効果量 r
Blob Class		0.026
Data Class		0.043
Distorted Hierarchy		0.577
God Class		0.071
Refused Parent Bequest		0.016
Schizophrenic Class		0.080
Tradition Breaker		0.007
Blob Operation	✓	0.195
Data Clumps		0.001
External Duplication		0.011
Feature Envy	✓	0.234
Intensive Coupling	✓	0.212
Internal Duplication	✓	0.202
Message Chains		0.317
Shotgun Surgery	n/a	
Sibling Duplication		0.005

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)

5　考察

本節では調査結果についての考察を述べる. 5.1 節では, コードスメル検出ツールが開発者に優先的に提示するべきコードスメルについての考察を述べる. 5.3 節では, 調査結果について既存研究との関連を述べる.

5.1　優先して提示するべきコードスメル

RQ1 に対する調査結果から, クラスレベルのコードスメルの *Blob Class* と *God Class*, *Schizopherenic Class*, *Tradition Breaker* と, メソッドレベルのコードスメル *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いことが分かった. 一方, RQ2 に対する調査結果から, メソッドレベルのコードスメルの *Blob Operation* と *Feature Envy*, *Intensive Coupling*, *Internal Duplication* について, リファクタリングされたコードスメル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. RQ1 で深刻度が高いほどリファクタリングされやすいと分かったコードスメルの種類の中で, RQ2 でリファクタリングによって深刻度が減少する傾向が見られたコードスメルの種類は, *Blob Operation* のみである. このことから, 開発者は特に *Blob Operation* の深刻度の高い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.

以上より, 深刻度の高いクラスの中で, *Blob Operation* の深刻度の高いメソッドを優先して開発者に提示するべきだと考えられる.

RQ1 の結果にはクラスレベルのコードスメルが

5.2　コードスメルに対応するリファクタリング

Fowler と Lanza はそれぞれ著書の中でコードスメルに対応したリファクタリングを紹介している [7, 13]. しかし, それはあくまで定性的な対応付けであり, 実際にコードスメルに対して有効であるか調査が必要である. そのため, 本研究では RQ についての調査に関連付けて, コードスメルに対応するリファクタリングが行なわれているかどうかと, 対応するリファクタリングがコードスメルの深刻度を減少するかどうかを調べた.

まず, Fowler または Lanza が各種コードスメルに対応するとしているリファクタリングの種類を表 7 に示す. ただし, *Intensive Coupling* について Lanza が対応するとしているリファクタリングは, コードスメルのあるクラスとは別のクラスに実施されるリファクタリングであり, コードスメルとの関連を特定することが困難なために除外している. また, *Distorted Hierarchy* と *Schizophrenic Class* は Fowler や Lanza の定義したコードスメルではない.

調査手順はリファクタリングをコードスメルに対応する種類に限定する以外は, 節で述べた調査手法と同じである. 3.3 節で説明した調査手順 3 のリファクタリングのグループ分けについて, Fowler[7] または Lanza[13] がコードスメルに対応するとしているリファクタリングの種類に限定する場合でグループ分けを行なう.

リファクタリングをコードスメルに対応する種類に限定した場合の調査結果を表 8 と表 9 に示す. チェックマークは有意差 ($p < .05$) を示している. RQ1 については, n/a と表記している部分は, その種類のコードスメルが検出されなかった場合も含めて, その種類のコードスメル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. RQ2 については, n/a と表記している部分は, その種類のコードスメル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, リファクタリングの種類を限定しない場合とは異なり, 全ての有意差の出ている種類について効果量も小以上とはならなかった.

RQ1 と RQ2 の両方で有意差が出たコードスメルの種類は, クラスレベルのコードスメルの *Blob Class* とメソッドレベルのコードスメルの *Blob Operation* であった.

5.3 関連研究との比較

後藤らの研究でメソッド抽出リファクタリングの推薦を目的として, 実際にメソッド抽出リファクタリングが実施されたメソッドの特徴を調査した結果, メソッドの行数と凝集度が大きく関与していた [8]. 一方, 本研究で扱うコードスメルのうち, 検出規則に行数用いるコードスメルは *Blob Class* と *Blob Operation* であり, 検出規則に凝集度を用いるコードスメルは *Blob Class* と *God Class*, *Schizopherenic Class* である. 調査の結果, これらのコードスメルについては, 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高い傾向にあった.

Sjoberg らの研究ではファイル毎の作業時間に影響するコードスメルを調べるための実験が行なわれた [18]. 実験では複数の開発者にあるシステムの保守作業のタスクを共通して与え, ファイル毎の作業時間とファイルの変更されたコミット数を記録した. また, 変更前のシステムについて, 各ファイルの行数が計測され, 12 種類のコードスメルが検出された. この 12 種類のコードスメルの中には本研究でも扱う *Data Class* と *God Class*, *Refused Parent Bequest*, *Data Clump*, *Feature Envy*, *Shotgun Surgery* が含まれている. この実験の結果としては, どの種類のコードスメルも作業時間への影響は見られなかった一方で, ファイルの行数が大きさが作業時間の長期化に影響していた. そのため, 彼らはファイルの行数を削減することが保守作業にかかる労力の減少に効果的であるとしている. 本研究では行数に関係

表 7: Fowler または Lanza によりコードスメルに対応付けられたリファクタリングの種類

コードスメルの種類	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
Distorted Hierarchy	
God Class	Extract Interface
	Extract Subclass
	Extract Class
	Replace Data Value with Object
Refused Parent Bequest	Replase Inheritence With Delegation
	Extaract Class
Schizophrenic Class	
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
Data Clumps	Extract Class
	Introduce Prameter Object
	Preserve Whole Object
External Duplication	Extract Method
	Extract Class
	Pull Up Method
	Form Template Method
Feature Envy	Extract Method
	Move Method
	Pull Up Method
	Move Field
Intensive Coupling	
Internal Duplication	Extract Method
	Extract Class
	Pull Up Method
	Form Template Method
Message Chains	Hide Delegate
Shotgun Surgery	Move Method
	Move Field
	Inline Class
Sibling Duplication	Extract Method
	Extract Class
	Pull Up Method
	Form Template Method

表 8: リファクタリングをコードスメル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

コードスメルの種類	有意差	効果量 r
Blob Class	✓	0.158
Data Class		0.020
Distorted Hierarchy	n/a	
God Class		0.021
Refused Parent Bequest	n/a	
Schizophrenic Class	n/a	
Tradition Breaker		0.025
Blob Operation	✓	0.049
Data Clumps	n/a	
External Duplication		0.075
Feature Envy		0.004
Intensive Coupling	n/a	
Internal Duplication		0.053
Message Chains	n/a	
Shotgun Surgery	n/a	
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 9: リファクタリングをコードスメル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

コードスメルの種類	有意差	効果量 r
Blob Class	✓	0.162
Data Class	n/a	
Distorted Hierarchy	n/a	
God Class		0.059
Refused Parent Bequest	n/a	
Schizophrenic Class	n/a	
Tradition Breaker	n/a	
Blob Operation	✓	0.189
Data Clumps		0.042
External Duplication		0.052
Feature Envy		0.110
Intensive Coupling	n/a	
Internal Duplication	✓	0.204
Message Chains	n/a	
Shotgun Surgery	n/a	
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)

するコードスメルである *Blob Operation* を特に優先的に提示するべきという調査結果になったが, *Blob Operation* を含むメソッドに対してリファクタリングが実施されることによって保守作業にかかる労力が減少することが期待できる.

6　まとめ

本研究では, 優先順位付きで開発者にコードスメルを提示するツールの開発のために, コードスメル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, クラスレベルのコードスメルの *Blob Class* と *God Class*, *Schizopherenic Class*, *Tradition Breaker* と, メソッドレベルのコードスメル *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いことが分かった. また, *Blob Operation* についてはリファクタリングによってコードスメルの深刻度が減少する傾向が見られた. よって, 深刻度の高いクラスの中で, *Blob Operation* の深刻度の高い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コードスメルを提示するツールを開発する必要がある.

謝辞

本研究において，常に適切な御指導及び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 井上 克郎 教授に心より深く感謝いたします．いつも非常に鋭い御指摘を頂くことが出来るので, 御指導頂けたことをとても幸運に思います．また, 非常に恵まれた環境で研究室生活を送れたことに感謝します．先生が海外から教員や留学生を多数招いてくださっているおかげで, リファクタリングやソフトウェア品質の専門家である, Marouane 先生や Ali 先生に御指導いただく機会を得ることも出来ました．研究について以外にも, 学生生活や就職活動などの相談に乗って頂き, 3 年間大変お世話になりました．

本研究において，随時適切な御指導及び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 春名 修介 特任教授に心より深く感謝いたします．企業での経験から貴重な御助言を度々頂きありがとうございました．また, 就職活動においても様々な助言を頂き大変お世話になりました．

本研究において，随時適切な御指導及び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 准教授に心より深く感謝いたします．また, 研究室の計算機管理業務についても大変お世話になりました．普段利用しているソフトウェアについての歴史的な経緯など, 興味深いお話を多くして頂いて大変勉強になりました．

本研究において，終始適切な御指導及び御助言を頂きました名古屋大学大学院情報科学研究科附属組込みシステム研究センター 吉田 則裕 准教授に心より深く感謝いたします．学部の卒業研究から 3 年間, 非常に熱心に御指導頂き誠にありがとうございました．他大学院の学生の研究も見ておられて大変お忙しいにもかかわらず, 研究室に訪れる機会を頻繁に設けて頂いた上に, 土日や休暇中などいつでも研究の相談に乗って頂きました．私が国内外の学会や研究会への論文を投稿することが出来たのは, 先生が常に研究を引っ張って頂いたからだと思います．

本研究において，終始適切な御指導及び御助言を頂きました大阪大学大学院国際公共政策研究科国際公益システム講座 崔 恩瀞 助教に心より深く感謝いたします．学部の卒業研究から 3 年間, 優しく御指導頂き誠にありがとうございました．昨年度までは博士課程の学生として研究室に在籍されていたため, 他の教員の方よりも気軽に接してしまうことが多く申し訳ありませんが, 相談しやすい位置に居てくれたことはとても助かりました．

本研究において，逐次適切な御指導及び御助言を頂き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 石尾 隆 助教に心より深く感謝いたします．また, 本研究について以外にも, 様々な研究や専門的知識について教えて頂きました．

本研究において，様々な御指導及び御助言を頂き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 Raula Gaikovina Kula 特任助教に心より深く感謝いたします．

また, 研究室生活においても大変お世話になりました.

本研究において，様々な御指導及び御助言を頂き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 Ali Ouni 特任助教に心より深く感謝いたします．本研究の方向性を決めるにあたって何度も相談に乗って頂き, 重要な御助言を頂きました.

また, 本研究の基礎において, 随時適切な御指導及び御助言を賜りましたミシガン大学コンピュータ工学部 Marouane Kessentini 助教授に心より深く感謝いたします．昨年度に共同研究者として研究室に滞在していらした際に, 本研究の基礎についてリファクタリングとソフトウェア品質の専門家として様々な御助言を頂きました.

最後に，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井上研究室の皆様に深く感謝いたします.

参考文献

[1] 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. In *Proc. of SCAM*, pp. 104–113, 2012.

[2] 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.

[3] 崔恩瀞, 藤原賢治, 吉田則裕, 林晋平. 変更履歴解析に基づくリファクタリング検出技術の調査", コンピュータソフトウェア. コンピュータソフトウェア, Vol. 32, No. 1, pp. 47–59, 2015.

[4] E. V. Emden and L. Moonen. Java quality assurance by detecting code smell. In *Proc. of WCRE*, pp. 97–106, 2002.

[5] Francesca Arcelli Fontana, Pietro Braione, and Marco Zanoni. Automatic detection of bad smells in code: An experimental assessment. *Journal of Object Technology*, Vol. 11, No. 2, pp. 1–38, 2012.

[6] M. Fowler. Refacoring. http://www.refactoring.com/.

[7] M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison Wesley, 1999.

[8] 後藤祥. ソースコードの特徴量を用いた機械学習によるメソッド抽出リファクタリング推薦手法. Master's thesis, 大阪大学大学院情報科学研究科, 2014.

[9] Brian Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity.* Prentice Hall, 1996.

[10] 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.

[11] Y. Kataoka, I. Takeo, A. Hiroki, and F. Tetsuji. A quantitative evaluation of maintainability enhancement by refactoring. In *Proc. of ICSM*, 2002.

[12] M. Kim, M. Gee, A. Loh, and N. Rachatasumrit. Ref-Finder: a refactoring reconstruction tool based on logic query templates. In *Proc. of FSE*, pp. 371–372, 2010.

[13] M. Lanza and R. Marinescu. *Object-Oriented Metrics in Practice*. Springer-Verlag, 2006.

[14] R. Marinescu and D. Ratiu. Quantifying the quality of object-oriented design: the factor-strategy model. In *Proc. of WCRE*, pp. 192–201, Nov 2004.

[15] E. Murphy-Hill, A. P. Black, D. Dig, and C. Parnin. Gathering refactoring data: a comparison of four methods. In *Proc. of the WRT*, 2008.

[16] K. Perete, N. Rachatasumrit, and M. Kim. Catalog of templete refactoring rules. In *Technical Report UTAUSTINECE-TR-041610*, 2010.

[17] F. Simon, F. Steinbruckner, and C. Lewerentz. Metrics based refactoring. In *Proc. of CSMR*, pp. 30–38, 2001.

[18] D. I. K. Sjoberg, A. Yamashita, B. C. D Anda, A. Mocks, and T. Dyba. Quantifying the effect of code smells on maintenance effort. *IEEE Transactions on Software Engineering*, Vol. 39, No. 8, pp. 1144–1156, 2013.

[19] G. Soares, R. Gheyi, E. Murphy-Hill, and B. Johnson. Comparing approaches to analyze refactoring activity on software repositories. *The Journal of Systems and Software*, Vol. 86, No. 4, pp. 1006–1022, 2013.

[20] 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? In *Proc. of SCAM*, pp. 95–104, 2014.

[21] L. Tokuda and D. Batory. Evolving object-oriented designs with refactorings. *Automated Software Engineering*, Vol. 8, No. 1, pp. 89–120, 2001.

[22] Aiko Yamashita, M. Zanoni, F. A. Fontana, and B. Walter. Inter-smell relations in industrial and open source systems: A replication and comparative analysis. In *Proc. of ICSME*, pp. 121–130, 2015.