オープンソースソフトウェアにおけるCode Smellと対応するリファクタリングの特徴に関する調査

本田 澄[1]　西尾 達哉[1]　鷲崎 弘宜[2]　深澤 良彰[2]

概要：Code Smell はソフトウェアの品質を低下させる要因の 1 つであるが，そもそもソフトウェアに Code Smell はどれだけ存在しているのか．また，その Code Smell に対するリファクタリングは行われているのかわからない．また，Code Smell は開発者や開発環境などによっても異なっており，基準が明確化されていない．そのため，複数のソフトウェアに共通して，具体的にどのような Code Smell の種類が多く存在しているかわからず，開発者は，どの Code Smell の種類を優先して気を付けるべきであるかわからない．さらに，ソフトウェアの特徴によって，存在する Code Smell の種類は変化するのか明らかになっていない．そこで，本研究では，GitHub 上のオープンソースソフトウェア (OSS) を対象として，Code Smell とリファクタリングの現状を調査して分析を行った．分析の結果，複数のソフトウェアに共通して，多く存在する Code Smell の種類を特定することができた．また，存在する Code Smell 数に対するリファクタリングの割合が高い Code Smell の種類を特定することができた．

1. はじめに

Code Smell(スメル) とはソフトウェアに対して深刻な問題を引き起こす可能性があるソースコード片のことである．リファクタリングとはソフトウェアを外部から見た時の動作を変更することなく，内部構造を整理する作業である [1]．リファクタリングを行う上での指標の 1 つとしてスメルがある．

スメルはソフトウェアの品質を低下させる要因の 1 つであるが，そもそもソフトウェアにスメルはどれだけ存在しているのか．また，そのスメルに対するリファクタリングは行われているのかわからない．

また，スメルは開発者や開発環境などによっても異なっており，基準が明確化されていない．そのため，ソフトウェアに共通して，具体的にどのようなスメルの種類が多く存在しているかわからず，開発者は，どのスメルの種類を優先して気を付けるべきであるかわからない．さらに，ソフトウェアの特徴によって，存在するスメルの種類は変化するのか明らかになっていない．

開発者がリファクタリングをすべてのスメルに対して行うことは不可能である．そこで，リファクタリングを行うスメルを選ばなくてはならないが，ソフトウェアのスメルを減少させるためにどのスメルが簡単にリファクタリングできるか明らかになっていない．

本研究の研究課題を以下に定義する．

RQ1 オープンソースソフトウェア (OSS) に存在するスメル数とリファクタリング数の調査

RQ2 ソフトウェアに共通して，多く存在するスメルの種類を特定

RQ3 ソフトウェアの特徴と存在するスメルの種類に関係はあるのか

RQ4 ソフトウェアに共通して，リファクタリング数の多いスメルの種類を特定

RQ5 ソフトウェアに共通して，存在するスメル数に対するリファクタリングの割合が高いスメルの種類を特定

以上の 5 つの課題を解決するために GitHub 上の OSS を対象として，スメルとリファクタリングの現状を調査する．そして，調査結果をもとにして分析を行った．

本研究の貢献を以下に示す．

- OSS に存在するスメル数とリファクタリング数を確認した．
- ソフトウェアに共通して，多く存在するスメルの種類を示した．
- ソフトウェアの特徴によって，多く存在するスメルの種類を確認した．
- ソフトウェアに共通して，リファクタリング数の多いスメルの種類を示した．

[1] 大阪工業大学
Osaka institute of Technology
[2] 早稲田大学
Waseda University

© 2021 Information Processing Society of Japan

- ソフトウェアに共通して，存在するスメル数に対するリファクタリングの割合が高いスメルの種類を示した．

2. 背景

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を外部から見た時の動作を変更することなく，内部構造を整理する作業である [1]．つまり，リファクタリングはソフトウェアの機能は変更することなく，ソースコードの可読性を高めることで，機能の追加やバグの修正が行いやすいようにソフトウェアの構造を変化させることである．

リファクタリングを行うことでソースコードを読みやすくし，機能追加やバグ修正が行いやすくなることで，ソフトウェア開発の効率を向上させることにもつながる．しかし，適切なリファクタリングが行われないと，反対にソースコードの可読性を低下させ，ソフトウェア開発の効率を低下させることにもつながりかねない．

以下に代表的なリファクタリングの種類をいくつか示す [1]．本研究で用いるリファクタリングはこれらの一部である．

- 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 SonarQube

SonarQube[2] とはソフトウェアのソースコードの行数やファイル数といった基本的な情報の解析から，バグやスメルの発見，複雑度や重複するコードの割合など，ソフトウェアの品質を評価するための様々な機能を持っている統合品質管理ツールである．解析できるプログラミング言語も Java，JavaScript，C，C++，Python など様々な言語に対応している．

そして，SonarQube は実際のソフトウェア開発の途中で使用されることを想定して，ソフトウェアのある特定のバージョンのみを解析するのではなく，開発の途中で何度も解析を行うことで，開発全体を通してスメル数はどのように変化しているか．また，どのような種類のスメルが新たに発生したかを確認することができる．

2.3 具体的なスメルの内容とリファクタリング例

ソフトウェアの中に発生するスメルは，開発に用いているプログラミング言語や，開発方法，開発者などによって，そのソースコードがスメルであるか，それともスメルではないのか変化する．つまり，スメルかどうかを判断するのは主観的な部分が大きい．

本研究では Java 言語で開発されたソフトウェアを解析対象としており，SonarQube には Java 言語の規約として独自に決められている 386 種類の規約がある [3]．対象の規約にはステータスが割り当てられており，ステータスの一つである Beta の場合は，ユーザーのフィードバックも反映させて，規約として採用するかを検討中の規約である．Deprecated の場合は，同様の正確なルールが存在するため，使用されない規約である．Ready の場合は，本番環境で使用することができることを指している．解析に使用している 386 種類の規約はステータスがすべて Ready の規約である．

スメルを含むソースコードの例として，OSS の Apache Commons Lang のバージョン 3.6 の CharRange.java ファイルで発見されたスメルを図 1 に示す．バージョン 3.6 の 209 行目で false を使用しているが，ブールリテラルを用いると冗長になり，コードの可読性を低下させることによるスメルである．スメルがバージョン 3.7 でリファクタリングされたている．スメルに対して，false を削除し，論理否定演算子（!）を書き加えることで，if 文を否定の形にしてスメルを削除した [4]．

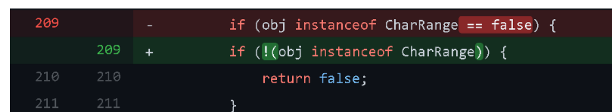

図 1 スメルを含むソースコードとその修正

3. 解析手法

解析手法の全体像を図 2 に示す．本研究では GitHub 上にある OSS を対象とし，リリースバージョンごとに解析を行う．

入力としては各リリースバージョンのソースコードであり，解析を行うツールは SonarQube を用いてスメル数を出力する．その解析結果をもとに，1 つはスメルの分類ごとに集計し，各ソフトウェアにおけるスメル割合の算出と対象ソフトウェアすべてのスメル割合を集計する．もう 1

© 2021 Information Processing Society of Japan

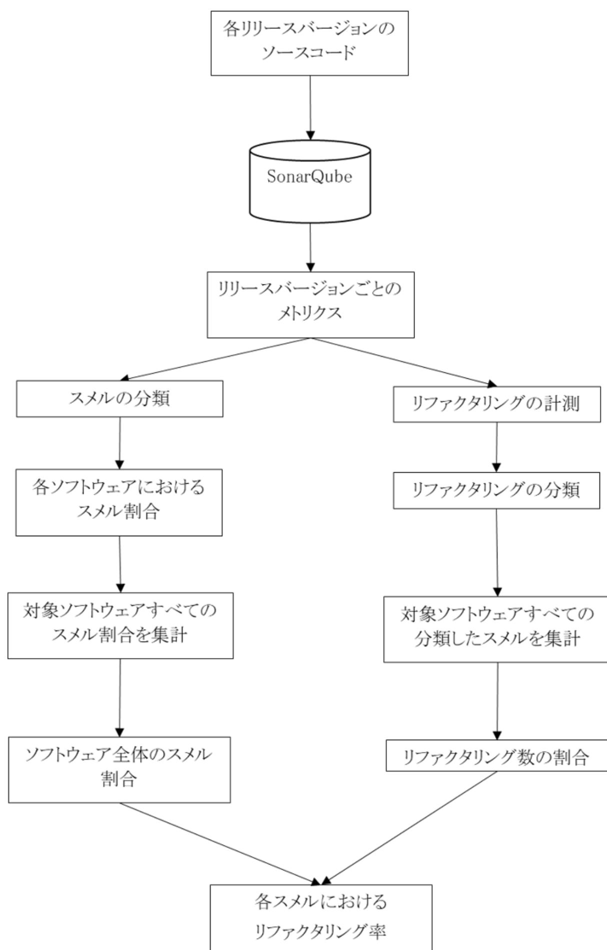

図 2 解析手法の全体像

つは，リファクタリングを計測し，リファクタリングされたスメルの分類を行い，対象ソフトすべてのスメルを集計する．その結果から，リファクタリング数の割合の算出を行う．そして，発見されたスメルの総数とリファクタリングされたスメル数からスメルの種類ごとのリファクタリング率を求める．詳しい内容は以下の節で説明する．

3.1 対象ソフトウェアの解析

本研究では GitHub 上の OSS を対象に SonarQube を用いて解析を行う．研究を進めるために，事前に 12 個の OSS の最新バージョンのみを対象として，スメル解析を実施した結果を表 1 に示す．

この結果から分析した 12 個の OSS において，スメルが無いものは無かった．次に，この中から過去のバージョンのソースコードが入手可能であり，SonarQube を利用してスメル解析を実施できるものを選んだ．SonarQube を利用したスメル解析においては，ソフトウェアの実行環境を過去の OS 等に合わせる必要があり，解析できないソフトウェアがあったため，過去のバージョンでも解析できるソフトウェアを対象とした．

先に述べた理由から，以下の 5 個のソフトウェアを対象にスメル数の変化を確認するために，リリースバージョンごとに解析を行う．解析した OSS はすべて Java 言語で開発されている．

表 1 12 個の OSS のスメル数

ソフトウェア名	スメル
Apache Commons IO [5]	303
Apache Commons Lang [6]	678
Apache Commons Logging [7]	247
Apache Commons Math [8]	5733
Apache Commons Net [9]	890
Apache Drill [10]	107
Apache Mahout [11]	537
Apache Ozone [12]	149
Apache Phoenix [13]	8612
Apache Tika [14]	881
Google Mug [15]	132
Yahoo Oak [16]	133

- Apache Commons IO
- Apache Commons Lang
- Apache Commons Logging
- Apache Commons Math
- Apache Commons Net

OSS にはテストファイルが含まれているが，本研究では分析結果にテストファイルのコード行数やスメル数は含まないものとする．

3.2 リファクタリングの計測方法

本研究におけるリファクタリングとはスメルが削除されることを意味する．SonarQube を使用するとバージョンの異なるソフトウェアの間で増加したスメルを特定することができる．しかし，減少したスメルを特定することはできないため，リファクタリングを計測することができない．

そこで，図 3 の図を用いて計測手順を説明する．バージョン k とバージョン $k+1$ で同じ名前のファイル A を探す．そして，バージョン k のファイル A にはスメルが存在しているのに，バージョン $k+1$ ではなくなっていることがわかる．そのため，GitHub のコード変更履歴を使用して，2 つのファイルを比較し，スメルの消えているコードを特定し，リファクタリングとして計測する．バージョンが異なるソフトウェアの間でスメルを含むファイル自体が削除された場合は，リファクタリング数として計測していない．

3.3 スメルの分類

対象のソフトウェアに存在するスメルを種類ごとに分類する．その結果をもとにソフトウェアに共通して多く発見されるスメルと各ソフトウェアの特徴によって多く発見されるスメルを確認するために 2 つの視点から分析を行う．

1 つ目は，5 つのソフトウェアの全体で多く発生してい

© 2021 Information Processing Society of Japan

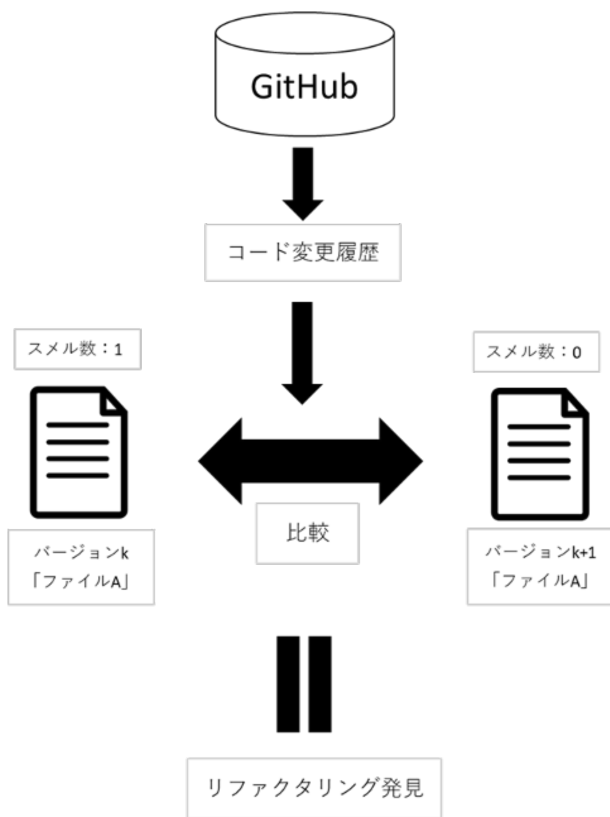

図 3　リファクタリング計測手法の例

るスメルの種類を分析する．SonarQube で解析したスメルを種類ごとに分け，バージョンごとのスメルの増加数を足し合わせて，ソフトウェアに発生したスメルの総数を算出する．これをソフトウェアごとに行う．しかし，ソフトウェアによってスメルの総数は異なるため，単純に各ソフトウェアで測定されたスメルの種類を分析するわけにはいかない．そこで，各ソフトウェアのスメルの総数を用いて，スメルの種類ごとにスメルの割合を算出する．

2 つ目は，1 つ目で算出したスメルの割合をソフトウェアごとに表すことで，そのソフトウェアに発生しているスメルの特徴を分析する．

3.4　リファクタリングの分類

リファクタリングされる数の多いスメルの種類を分析するためと，ソフトウェアで発見されたスメル数に対するリファクタリングの割合を分析するため，それぞれの視点から分析を行う．

1 つ目はリファクタリング数に対する分析である．各ソフトウェアで計測されたリファクタリングをスメルの種類ごとに分け，割合として算出する．ソフトウェアごとに算出した結果を 1 つにまとめて分析を行う．

2 つ目はリファクタリング率に対する分析である．ソフトウェアによって発生しているスメルの数は異なり，スメルの種類ごとに分類した場合でも同様である．そこで，発生したスメルの総数のうち，どの程度リファクタリングが行われているかをスメルの種類ごとに算出し，分析を行う．発生したスメルの総数にはファイル削除によって消えたスメルの数を含まないものとする．

4.　結果

結果に示す値は SonarQube のバージョン 8.5 を利用して分析した結果である．

4.1　スメルの分類

4.1.1　ソフトウェア全体のスメルの割合

5 つのソフトウェアで発見したスメルを合わせて，種類ごとの割合として表した結果を図 4 に示す．

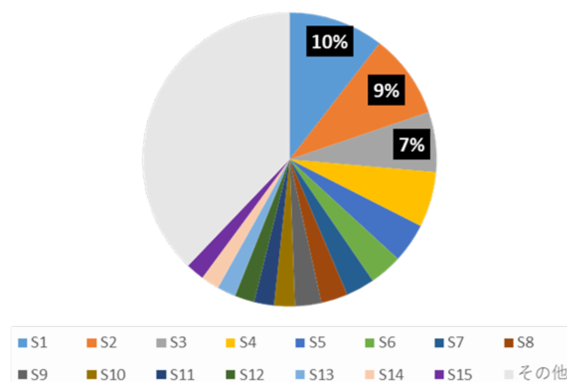

図 4　5 つのソフトウェアで発見したスメルの割合

解析結果では，141 種類のスメルが発見できたが，割合が 2%よりも小さいスメルはその他としている．図 4 で割合が 2%以上のスメルの種類を表 2 に示す．

表 2　5 つのソフトウェアに含まれるスメルの種類

スメル番号	スメルの種類
S1	不要な throw 文の使用
S2	ジェネリクス型を型パラメータなしで使用
S3	非推奨メソッドの定義
S4	非推奨メソッドを使用
S5	同期されたクラスの使用
S6	標準出力を使用してのログの記録
S7	複雑度が高い
S8	フィールド名が命名規則に反している
S9	オーバーライドが示されていない
S10	不必要なブールリテラルの使用
S11	配列指定子が変数名に書かれている
S12	コードをコメントアウトしている
S13	ファイル内に重複したコードのかたまりが書かれている
S14	1 行に複数の変数が宣言されている
S15	空のメソッドを使用

図 4 では 5 つのソフトウェアに共通して発見されたスメルの割合が確認でき，多く発見されたスメルの種類を特定

© 2021 Information Processing Society of Japan

することができる．5 つのソフトウェアで最も多く発見されたスメルは「不要な throw 文の使用」，2 番目は「ジェネリクス型を型パラメータなしで使用」，3 番目は「非推奨メソッドの定義」であった．

4.2　リファクタリングの分類

4.2.1　リファクタリング数の割合

リファクタリングの計測結果からスメルの種類に分けて，リファクタリングされた種類別に割合として表した結果を図 5–9 に示す．

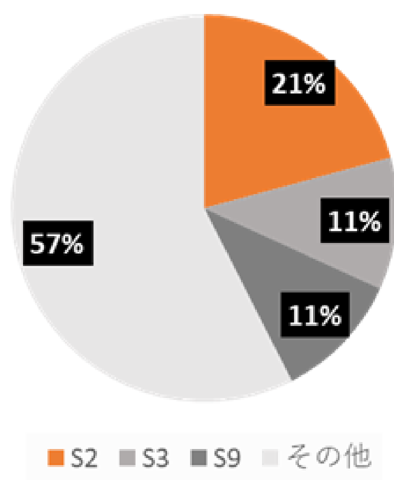

図 5　Apache Commons IO のスメルの割合

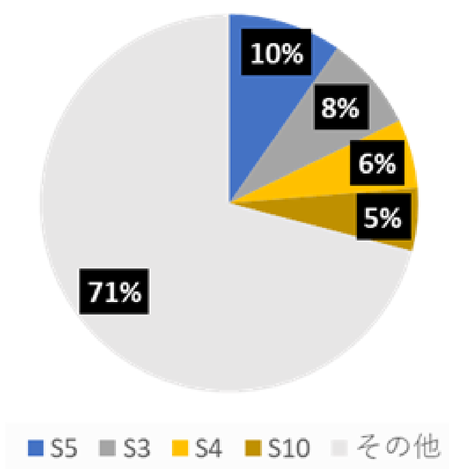

図 6　Apache Commons Lang のスメルの割合

リファクタリングされたスメルの種類は 80 種類であった．また，対象となるスメルは表 2 以外のスメルもあったため，表 3 に追加となったスメル番号とスメルの種類を示す．

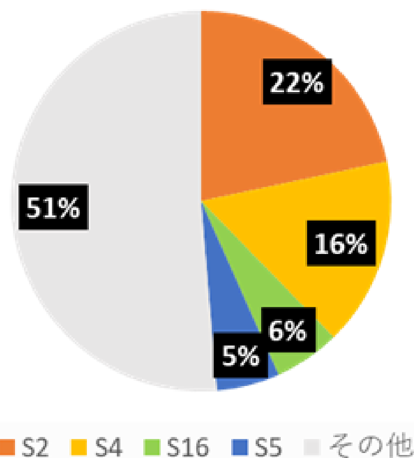

図 7　Apache Commons Logging のスメルの割合

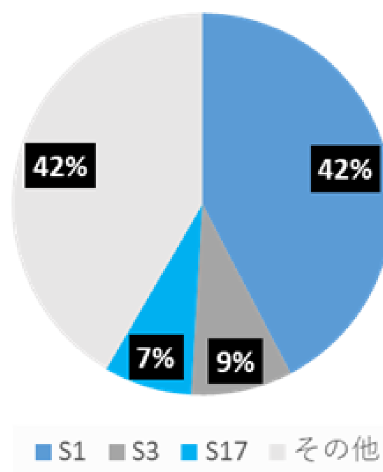

図 8　Apache Commons Math のスメルの割合

表 3　追加となったスメルの種類

スメル番号	スメルの種類
S16	定義分が正しい順序で書かれていない
S17	コンストラクター呼び出しの型指定を diamond 演算子（“<>”）で省略可能
S18	メソッド名が命名規則に反している

割合が 3%よりも小さいスメルはその他としている．図 5–9 で 3%以上のリファクタリングされたスメルの種類を表 4 に示す．

© 2021 Information Processing Society of Japan

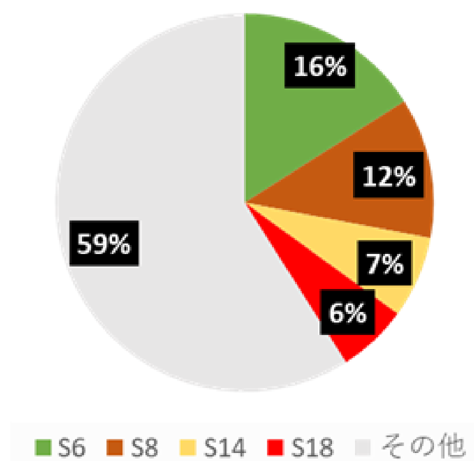

図 9　Apache Commons Net のスメルの割合

表 4　リファクタリングされたスメルの種類

スメル番号	スメルの種類
S2	ジェネリクス型を型パラメータなしで使用
S4	非推奨メソッドの使用
S5	同期されたクラスの使用
S9	オーバーライドが示されていない
S10	不必要なブールリテラルの使用
S14	1 行に複数の変数が宣言されている
S19	プリミティブラッパークラスをインスタンス化するためにコンストラクターを使用している
S20	定数名が命名規則に反している
S21	未使用のパラメータがある

図 10 は 5 つのソフトウェアに共通して，リファクタリングされたスメルの割合が確認でき，多くリファクタリングされたスメルの種類を特定できる．5 つのソフトウェアで最も多くリファクタリングされたスメルの種類は「非推奨メソッドの使用」，2 番目は「ジェネリクス型を型パラメータなしで使用」，3 番目は「オーバーライドが明記されていない」であった．

4.2.2　各スメルにおけるリファクタリング率

5 つのソフトウェアで発見できたスメル数に対するリファクタリング率を算出し，区分ごとにまとめた結果を表 5 に示す．表 2，表 3，表 4 のスメルも区分し，種類数にはそれらも含んだ数が示されている．表 5 からはリファクタリング数の割合とは関係なく，ソフトウェアに発生したスメル数に対して，どれだけの割合でリファクタリングされているか確認できる．

5.　考察

表 2 から表 5 の結果からバージョンごとにスメル数は変化しており，1 つ前のバージョンと比べて増加している場合でも，リファクタリングは行われている．Logging 以外のソフトウェアの傾向として，小数点のバージョンアップではなく，1 の位がバージョンアップされる場合にはリファクタリングが多くみられ，逆にそれ以外ではリファクタリングが行われているものの，スメル全体では増加していく傾向がみられる．

5.1　スメルの分類

5.1.1　ソフトウェア全体に存在するスメルの傾向

図 4 の結果から，S1，S2，S3 が，この順番に多く発見されたスメルである．S1 は throw 文で例外処理を行っているが，throws 文で実行されることのない例外処理を残したままにしていることが原因であると考えられる．S2 は，ジェネリック型の型パラメータが指定されないまま使用されることが多く確認された．これはコンパイル時にエラーとなることはないが，何の型のデータが格納されているかわからず，意図している型ではないデータが格納される可能性がある．S3，S4 は将来的に必要のなくなるメソッド等には「@Deprecated」が書かれているため，それを将来的に消すことを忘れないように発生しているスメルである．この 2 つのスメルは機能追加にともなって，多く発生しやすいスメルであると考える．

15 種類の中で 5 つのソフトウェアすべてに発見された種類は S2，S6，S9，S14 を除いた 11 種類である．つまり，この 11 種類のスメルはソフトウェア開発の中で特に存在しやすいスメルの種類であると考えられる．

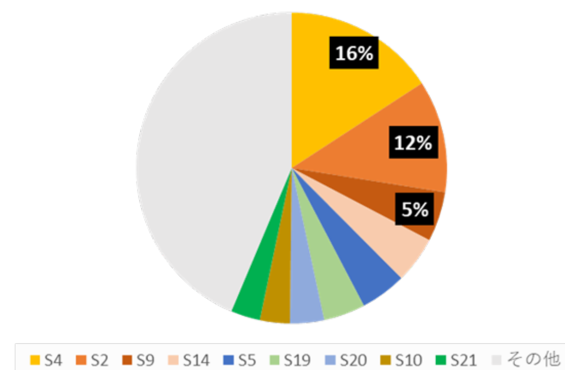

図 10　5 つのソフトウェアでリファクタリングされたスメルの割合

5.1.2　各ソフトウェアに存在するスメルの傾向

ソフトウェアごとにスメルを分けてみるとソフトウェアによって異なるスメルが発見できた．スメルが検出された箇所のソースコードを精査した．

(1) IO

IO は input と output の機能開発を支援するためのライブラリであり，ファイルを操作するための機能が開発されている．IO には File のコピーや文字列を操作するようなメソッドが多く存在する．そのようなメソッドで処理を行

う時，文字列の格納に List 型や Collection クラスを使用しているが，型パラメータが指定されていないものがあることを目視で確認した．このため，S2 が多く発見されたと考えられる．

表 5 各スメルのリファクタリング率

リファクタリング率 [%]	0	0 < 40	40 < 70	70 < 100	100
スメルの種類	S17	S1 S3 S6 S7 S10 S11 S12 S13 S15 S18 S20	S2 S5 S8 S14 S16	S4 S9 S19 S21	
	60 種類	39 種類	23 種類	8 種類	11 種類

（2）Lang

Lang は Java に標準で備わっている文字列やオブジェクト関係のメソッドを拡張して開発されたライブラリである．Lang では文字列を操作するメソッドを多く扱っているため，String Buffer クラスを多く使用している．しかし，String Buffer クラスは同期を考えた設計になっているというメリットはあるものの，処理が遅くなってしまうというデメリットがあるため，Lang では S5 が多く発見されたと考える．

（3）Logging

Logging は Log4j や java. util. logging などの他のロガー実装に処理を委譲してくれるライブラリである．Logging にはログを出力するメソッドが存在するが，その引数として Class を指定しなければならない．Class クラスはジェネリクス型であるため，S2 が多く発見されていると考える．さらに，Logging では Hashtable クラスが使用されており，String Buffer と同様にスレッドセーフであるため，HashMap クラスに変更するように S5 が多く発見されている．S16 については，動作に問題はないが，決められた順番で定義がされていないと，コードの可読性が低下するため，スメルとして多く発見されている．

（4）Math

Math は数学や統計関係の機能を含んだライブラリである．Math は他の 4 つとは異なっており，S1 の割合がとても高いため，図 1 にも大きく影響している．S1 が多く発生している原因としては，1 つのメソッドに対して多くの例外処理が行われており，余分な例外処理が含まれている可能性が考えられる．S17 は開発の途中で Java のバージョンが変更により，ジェネリクス型をインスタンス化するときに型引数を省略できるように変更されたことが原因であると考えられる．

（5）Net

Net は FTP や SMTP など多くの基本的なインターネットプロトコルのクライアント側の実装を行うための機能を含んだライブラリである．Net には通信プロトコル関係のメソッドが多く存在しており，例えば通信の途中でエラーが発生した場合は使用しているユーザーに早急に通知する必要がある．したがって，ログファイルに記述するのではなく，標準出力する必要があり，S6 が多く発生していると考える．他の 4 つのソフトウェアでは多く発見されていない，S14 は Net ではユーザー名やパスワードなどを String 型に格納しているため，変数の宣言が多いことが原因である可能性が考えられる．

5.2 リファクタリングの分類

5.2.1 リファクタリング数の多いスメルの傾向

図 10 の結果からリファクタリング数が多かったのは S4，S2，S9 の順番である．S4 は非推奨のメソッドの定義が 1 つ削除されるたびに複数の場所で削除を行わないといけないため，最も多くリファクタリングされたと考えられる．S2 は IO と Logging で多く発見されたスメルである．commons-io-2.0 で List 型や Collection クラスに型パラメータを割り当てることで，残っているすべての S2 に対してリファクタリングが行われているため，全体のリファクタリング数に影響を与えていると考える．Logging はバージョンが 2．0 になる前に開発が停止したため，リファクタリングが行われておらず，スメルとして残っていると考える．S9 は@Override が書かれていないメソッドが発見されており，そのメソッドに対して，@Override を書き加えることで，コードの可読性を高めるリファクタリングが行われた．S9 のリファクタリングはとても容易に行うことができるため，リファクタリングされやすいと考える．

5.2.2 リファクタリング率の高いスメルの傾向

表 4 の結果からリファクタリングが 100%行われているスメルの種類は 11 種類発見できた．しかし，5 つのソフトウェアで多く発見されたスメルの中にリファクタリングが 100%行われていたスメルはない．発見された 11 種類は，すべて総スメルの割合は 1%以下であり，数は少ないが優先的にリファクタリングされた，もしくは容易にリファクタリングができる種類であると考えられる．70%〜99%未満のスメルである S4，S9 は総スメル数の割合が高いが，そのままスメルを残した状態にはせず，リファクタリングが積極的に行われていると考えられる．反対に 1%〜40%未満のスメルで S1，S3，S6，S7，S10，S11，S12，S13，S15 は総スメルの割合が高いため，リファクタリングが追い付

© 2021 Information Processing Society of Japan

かず，今後も増加していく可能性が考えられる．0%である 60 種類のスメルはそもそもスメルとして認識されていないか，リファクタリングするにもコストがかかりすぎるため，対応していない可能性が考えられる．

6. 関連研究

スメルについては様々な研究が行われており，特にスメルの検出に関する研究は多く行われている．Fontana らは複数のスメル検出ツールに対してそれぞれの違いについてまとめている [17]．彼らの研究では一つのソフトウェアを利用し，そのバージョン毎でスメルの変化を分析しそれぞれのツールの違いを明らかにした．また，Deep Learning を利用したスメルの検出 [18] や，多ラベル分類を利用したスメルの検出 [19] など，機械学習を利用したスメルの検出手法について多くの研究が行われている．Deep Learning を利用したスメルの検出においては，直接学習と転移学習での違いについての研究も行われている [20]．我々の研究では複数のソフトウェアを対象とし，スメルの変化を分析し，特にリファクタリングについて研究している．

Palomba らは 13 に分類されたスメルに対して，それぞれのスメルがどのスメルと同時に発生するかについて分析している [21]．我々は SonarQube で利用されている 386 種類の規約から発見されたスメルを対象とし，それぞれのスメルのリファクタリングについて分析を行っている．

OSS のリファクタリング実施履歴をもとにどのようなスメルを含んだクラス，またはメソッドがリファクタリングされやすいか分析する研究がある [22]．クラスやメソッド単位でのリファクタリングが想定されている．一方，本研究はスメルの種類をさらに細かく分類し，スメルの種類単位でのリファクタリングを考え，リファクタリングされる可能性が高いスメルの種類を分析している．

7. おわりに

本研究では GitHub の OSS を対象に SonarQube を用いて解析を行うことで，OSS に存在するスメル数とリファクタリング数を確認できた．その結果，どの OSS にもスメルは存在しており，同時にリファクタリングも行われていた．しかし，スメルはバージョンが上がるごとに増加している傾向がみられ，多く存在しているスメルの種類を特定するために，スメルの分類を行い分析した．その結果，ソフトウェアに共通して，存在するスメルとして「不要な throw 文の使用」,「ジェネリクス型を型パラメータなしで使用」,「非推奨メソッドの定義」が多く発見され，開発者としてはスメルの増加を防ぐために優先して気を付けるべきスメルである．各ソフトウェアに存在するスメルの傾向からはソフトウェアの特徴ごとに異なるスメルの存在を確認することができた．

リファクタリングの計測結果から，リファクタリングされたスメルの数を分析した結果,「非推奨メソッドの使用」,「ジェネリクス型を型パラメータなしで使用」,「オーバーライドが示されていない」が多く発見された．加えて，リファクタリング率による分析の結果からも,「非推奨メソッドの使用」,「オーバーライドが示されていない」は多くリファクタリングされていることが確認でき，開発者がスメルを減らすうえで優先してリファクタリングするべきスメルである．

今後の課題としては，リファクタリングの計測に時間がかかりすぎてしまうため，SonarQube で決められている多くのルールに合わせたリファクタリング計測ツールの開発が必要である．そして，対象とするソフトウェアを増加させることで，調査結果の信憑性を向上させる必要がある．

謝辞 本研究は JSPS 科研費 JP19K20242 の助成を受けたものです．

参考文献

[1] Fowler, M.: *Refactoring: improving the design of existing code*, Addison-Wesley Professional (2018).
[2] : Code Quality and Code Security — SonarQube, `https://www.sonarqube.org/`.
[3] : Rules explorer, `https://rules.sonarsource.com/java/type/Code%20Smell`.
[4] : Boolean comparisons in CharRange by mureinik · Pull Request #289 · apache/commons-lang, `https://github.com/apache/commons-lang/pull/289/`.
[5] : apache/commons-io: Mirror of Apache Commons IO, `https://github.com/apache/commons-io`.
[6] : apache/commons-lang: Mirror of Apache Commons Lang, `https://github.com/apache/commons-lang`.
[7] : apache/commons-logging: Apache Commons Logging, `https://github.com/apache/commons-logging`.
[8] : apache/commons-math: Miscellaneous math-related utilities, `https://github.com/apache/commons-math`.
[9] : apache/commons-net: Apache Commons Net, `https://github.com/apache/commons-net`.
[10] : apache/drill: Apache Drill is a distributed MPP query layer for self describing data, `https://github.com/apache/drill`.
[11] : apache/mahout: Mirror of Apache Mahout, `https://github.com/apache/mahout`.
[12] : apache/ozone: Scalable, redundant, and distributed object store for Apache Hadoop, `https://github.com/apache/ozone`.
[13] : apache/phoenix: Mirror of Apache Phoenix, `https://github.com/apache/phoenix`.
[14] : apache/tika: The Apache Tika toolkit detects and extracts metadata and text from over a thousand different file types (such as PPT, XLS, and PDF)., `https://github.com/apache/tika`.
[15] : google/mug: A small Java 8 util library, complementary to Guava (BiStream, Substring, MoreStreams, Parallelizer)., `https://github.com/google/mug`.
[16] : yahoo/Oak: A Scalable Concurrent Key-Value Map for Big Data Analytics, `https://github.com/yahoo/Oak`.
[17] Fontana, F. A., Mariani, E., Mornioli, A., Sormani, R. and Tonello, A.: An experience report on using code smells detection tools, *2011 IEEE fourth international*

conference on software testing, verification and validation workshops, IEEE, pp. 450–457 (2011).
[18] Liu, H., Jin, J., Xu, Z., Bu, Y., Zou, Y. and Zhang, L.: Deep Learning Based Code Smell Detection, *IEEE Transactions on Software Engineering*, pp. 1–1 (online), DOI: 10.1109/TSE.2019.2936376 (2019).
[19] Guggulothu, T. and Moiz, S. A.: Code smell detection using multi-label classification approach, *Software Quality Journal*, Vol. 28, No. 3, pp. 1063–1086 (2020).
[20] Sharma, T., Efstathiou, V., Louridas, P. and Spinellis, D.: Code smell detection by deep direct-learning and transfer-learning, *Journal of Systems and Software*, Vol. 176, p. 110936 (2021).
[21] Palomba, F., Bavota, G., Di Penta, M., Fasano, F., Oliveto, R. and De Lucia, A.: A large-scale empirical study on the lifecycle of code smell co-occurrences, *Information and Software Technology*, Vol. 99, pp. 1–10 (2018).
[22] 雑賀翼, 崔恩瀞, 吉田則裕, 春名修介, 井上克郎: 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, 電子情報通信学会技術研究報告; 信学技報, Vol. 115, No. 508, pp. 91–96 (2016).