特別研究報告

題目

JTDog: 動的テストスメルの検出を目的とした Gradle プラグイン

指導教員
楠本 真二 教授

報告者
谷口 真幸

令和 3 年 2 月 9 日

大阪大学 基礎工学部 情報科学科

令和 2 年度 特別研究報告

JTDog: 動的テストスメルの検出を目的とした Gradle プラグイン

谷口 真幸

内容梗概

テストコードの可読性や保守性に対する潜在的な問題を表す概念として，テストスメルが知られている．一般的なテストスメルはテストの長さや複雑さといったソースコードの静的な側面に着目しており，検出が容易な一方で常に問題につながるとは限らない．他方で，テスト実行時の振る舞いに基づく動的スメルは，テストの実施が不十分であるにも関わらず，問題なくテストが成功したという誤った認識を開発者に与えるなど，テスト結果の誤解につながることが多い．そのため，可能な限り早期に検出し対策することが望ましい．本研究では，動的スメルを検出する仕組みを，ビルドツールである Gradle のプラグインとして実現する．実現にあたって，そのプロトタイプとして JTDog を実装した．ビルドツールへの組み込みにより，高い可搬性を実現したうえで，テストの潜在的な問題を検出可能となる．GitHub 上の 150 のプロジェクトに対して行った適用実験の結果，JTDog の可搬性を確認することができた．また，実際に 55 のプロジェクトから 958 個の動的スメルを検出できた．

主な用語

ソフトウェアテスト，テストスメル，動的解析，ビルドツール，Gradle，プラグイン

目次

1 はじめに 1

2 テストスメル 3

2.1 概要 . . . 3

2.2 静的スメル . . . 3

2.3 動的スメル . . . 3

2.4 静的スメルと動的スメルの比較 . . . 5

3 提案手法 6

3.1 概要 . . . 6

3.2 動的スメルの検出 . . . 6

3.3 Gradle プラグインの実現 . . . 7

4 適用実験 11

4.1 実験の目的 . . . 11

4.2 適用対象 . . . 11

4.3 実験方法 . . . 11

4.4 実験結果 . . . 11

4.5 動的スメルの実例 . . . 14

5 おわりに 17

謝辞 18

参考文献 19

図目次

1 Rotten Green Test の例 4
2 動的スメル検出の流れ 8
3 動的スメル検出処理の概要 9
4 検出した Rotten Green Test の例[*1] 14
5 検出した Flaky Test の例[*2] 15
6 検出した Dependent Test の例[*3] 16

表目次

1 静的スメルと動的スメルの比較 5
2 JTDog 適用失敗の原因 12
3 動的スメル検出結果 12
4 Rotten Green Test 検出プロジェクト（上位 5 件抜粋） 13
5 Flaky Test 検出プロジェクト（全 5 件） 13
6 Dependent Test 検出プロジェクト（上位 5 件抜粋） 14

1　はじめに

ソフトウェアの品質保証のための開発工程にソフトウェアテストがある．テストはソフトウェア開発において重要な役割を担っており，テスト駆動開発や継続的インテグレーションのような，テストを中心とした開発手法も広く用いられている [1][2]．通常，開発者はテスト実施のために，プロダクトコードとは別にテストコードを記述する．テストコードの品質は，ソフトウェア開発のスループットや生産性と正の相関関係がある [3]．そのため，テストコードの品質はプロダクトコードの品質と同様，開発者にとって重要である．

テストコードの可読性や保守性などに対する潜在的な問題を表す概念として，テストスメルが知られている [4]．テストスメルは，テストの品質低下につながる問題の兆候であり，早期に検出しリファクタリングなどのコード改善によって排除されることが望ましい．代表的なスメルとして，長すぎるコードや重複したテストが知られている [5]．これらのスメルがプロダクトコードでも共通する潜在的問題であるのに対し，テスト設計固有の考えに基づくスメルも存在する．例えば，テストコードにおける制御文の使用は，テストの複雑化のみならず，テストコード自体のバグにつながる可能性がある [5]．

これら一般的なテストスメルはテストコードの静的な側面に着目しており，その悪影響の対象も保守性などの静的な観点に限定される．以降，本論文ではこれらのスメルを静的スメルと呼ぶ．例えば先の例に挙げた，長すぎるテストコードは，理解の困難さや保守コストの増大などにつながる問題ではあるものの，テスト実行結果への影響はない．制御文を含むテストコードも同様に，テストの複雑さやテスト自体のバグにつながる潜在的な問題を示しており，テスト実行という側面での問題にはならない．

その一方で，テストの実行結果に影響を与えるスメル（以降，動的スメル）も存在する．例えば Rotten Green Test[6] は，成功したテストのうち，未実行のアサーションを含むテストのことであり，当該アサーションが行うべきであった動作確認を行っていないにも関わらず，単に成功したテストとして報告される．そのうえ，無意味なカバレッジの増加を引き起こし，開発者にテストが問題なく成功したという誤った認識を与えてしまう．Rotten Green Test のように，動的スメルはテストの実行結果に関する問題を引き起こし，テスト結果の誤解につながるため，静的スメルよりも深刻な問題につながりやすい．

これまでに数多くのテストスメルに関する研究 [7][8][9][10] が実施されてきたが，静的スメル，動的スメルといった分類を考慮した研究は存在しない．また，tsDetect[11] や TestHound[12] など，テストスメル検出を目的としたツールが多く提案されているが，これらのツールが検出可能なスメルはすべて静的スメルに限定される．我々の知る限り，動的スメル検出を目的としたツールは存在しない．

本研究の目的は，テスト実行結果に影響を与える動的スメルの早期検出の支援である．そのために，テストスメルを静的スメルと動的スメルに分類し，これらの性質を整理したうえで，動的スメルを検出する仕組みをビルドツールである Gradle のプラグインとして実現する．実現にあたり，そのプロトタ

イプとして JTDog を実装した.

静的スメルがテストコードの静的解析のみによって検出可能であるのに対して，動的スメルの検出には動的解析を要し，テスト実行やテストコードのカバレッジ解析など様々な処理を行う必要がある．加えて，動的スメル検出ツールの利用者は，テスト実行に必要なすべての情報を入力として設定する必要があり，すぐに利用することは難しい．よって，静的スメルを検出する場合と比べると，動的スメル検出ツールの可搬性は低くなる傾向にある．しかし，ビルドツールへの組み込みにより，動的スメル検出のための動的解析による可搬性の低下を防いだうえで，テストの潜在的な問題を検出可能となる.

JTDog の可搬性を確かめるため，GitHub 上の 150 のプロジェクトに対して適用実験を行った．その結果，122 のプロジェクトでプラグインが正常に動作し，そのうち 55 のプロジェクトから合計 958 個の動的スメルを検出できた.

2 テストスメル

2.1 概要

テストコードの可読性や保守性の低下など，テスト品質に影響する潜在的な問題を表す概念としてテストスメルがある．開発者にとっては，テストコードのリファクタリングの実施を促す存在であり，どういった場合にどのような手法を用いてリファクタリングを行うべきかの判断基準となる [4]．テストスメルに関しては多くの既存研究が存在しており [7][8][9][10]，これまでに様々な検出手法や検出ツールが提案されている．

本研究では，テストコードの静的な側面に基づくスメルを静的スメル，テスト実行時の振る舞いに基づくスメルを動的スメルと呼ぶ．本章ではこの 2 種類のスメルについて説明し，比較を行う．

2.2 静的スメル

静的スメルは，テストコードの静的な側面に基づくテストスメルである．例えば Verbose Test[5] は，長すぎるテストコードのことであり，コードの可読性の低下を引き起こす．その結果，テスト内容の理解を困難にする可能性があり，コードの保守性に悪影響を与える．Verbose Test がプロダクトコードでも共通する潜在的問題である一方で，テスト設計固有の考えに基づいたスメルも存在する．例えば Conditional Test Logic[5] は，テストコードに制御文を含むべきではないという考えに基づいたスメルである．制御文による条件分岐はテストコードの複雑化につながり，テスト内容の理解を困難にするため，テストの保守性の低下につながる．さらに，条件分岐により実行されない文が存在する場合，プロダクトのバグを検出できない可能性がある．

上記のように，静的スメルはテストに悪影響を及ぼすが，テストコードの静的な側面に着目しているという性質ゆえ，影響の対象も保守性などの静的な観点に限定される．例えば先の例に挙げた Verbose Test は，理解の困難さや保守コストの増大などにつながる問題ではあるものの，テスト実行結果への影響はない．Conditional Test Logic も同様に，テストの複雑さやテスト自体のバグにつながる潜在的な問題を示しており，テスト実行という側面での問題にはならない．

2.3 動的スメル

動的スメルは，テスト実行時の振る舞い，すなわち，テスト実行の経路，順序，結果に基づいたスメルである．テスト結果に悪影響を及ぼし，多くの場合，テスト結果の誤解などにつながる．現状，Rotten Green Test[6]，Flaky Test[13]，Test Dependency[14] の 3 種が存在する．

```
URL data = getResource("testData.xml"); //誤ったテストデータ
DataBase db = new DataBase(data);
List<User> users = db.getUsers(); //実際は空のリスト
for(User user : users){
  assertNotNull(user.getName()); //実行されない
}
```

図 1　Rotten Green Test の例

2.3.1　Rotten Green Test

Rotten Green Test は，成功したテストのうち，未実行のアサーションを含むテストのことである．テストはプロダクトコードの実行と動作確認という 2 つの要素で構成される．プロダクトコードの実行はプロダクトクラスやメソッドを呼び出す系列の部分であり，動作確認はアサーションなどの命令を用いて実行時の動作が期待通りであるかを確認する部分である．しかし，Rotten Green Test は，未実行のアサーションによる動作確認が一切行われていないにも関わらず，常に成功したテストとして報告される．例えば図 1 は，アサーションにより，リスト中の各ユーザの名前が null でないかを確認するテストである．しかし，1 行目のテストデータが誤っており，3 行目で取得するリストが空となるため，for 文内の処理が実行されず，5 行目のアサーションは実行されない．また，Rotten Green Test は，本質的でないテスト成功数の増加や無意味なカバレッジの増加といった問題も引き起こす．そのため，テストの実施が不十分であるにも関わらず，開発者にテストは問題なく成功したという誤った認識を与えてしまう．

2.3.2　Flaky Test

Flaky Test は，同じコード，同じ実行環境であるにも関わらず，実行するたびにテストの成否が変化するテストである．通常，開発者はテスト結果が変化した原因をテストコードや実行環境の変更によるものと考えるため，Flaky Test による変化であることを特定するためだけに，多大な時間や労力を費やすこととなる．また，Flaky Test による失敗が頻繁に発生する場合，開発者はその失敗を無視する傾向があるため，バグを見逃すきっかけともなる [13]．

2.3.3　Test Dependency

Test Dependency は，テストの実行順序が変わるとテスト結果が変わってしまうような，テスト間に存在する依存関係に基づくスメルである．このような実行結果が他のテストに依存するテストは Dependent Test と呼ばれる．

テスト順序の変更によりテストが失敗した場合，多くの場合はテスト実行前の初期化処理の失敗が原因である．しかし，開発者が Test Dependency の概念を知らない場合，原因の特定は困難であり，開発者は Dependent Test のテスト対象のプロダクトコードがバグを含むと判断してしまう恐れがある．また，テスト順序の変更によるテスト結果の変化は，通常の実行順序では表面化しないバグを見逃すきっかけとなる [14].

2.4　静的スメルと動的スメルの比較

静的スメルと動的スメルを，危険性やスメル検出の観点から比較した結果を表 1 に示す．この比較結果から，静的スメルが可読性や保守性への影響に限定される一方で，動的スメルはテスト結果の誤解などテストの実行結果に悪影響を及ぼす．よって，動的スメルは静的スメルよりも重大な問題を引き起こすため，早期に発見し，対策することが望ましい．しかし，tsDetect や TestHound などの既存のテストスメル検出ツールは静的スメルしか検出できず，動的スメル検出を目的としたツールは存在しない．

また，静的スメルがテストコードの静的解析のみによって検出可能であるのに対し，動的スメルの検出には動的解析を要し，テスト実行やテストコードのカバレッジ解析など様々な処理が必要である．さらに，動的スメル検出ツールの利用者は，プロダクトコードやテストコードに加え，リソースファイルやクラスファイル，外部ライブラリのパスなど，テスト実行に必要なすべての情報を入力として設定する必要があり，すぐに利用することは難しい．よって，静的スメルを検出する場合と比べると，動的スメル検出ツールの可搬性は低くなる傾向にある．

表 1　静的スメルと動的スメルの比較

	静的スメル	動的スメル
問題の影響範囲	可読性や保守性	テストの実行結果
解析手法	静的解析	静的解析と動的解析
解析に必要な情報	テストコードのみ	テスト実行に必要な情報すべて
検出ツール	tsDetect[11], TestHound[12] など	–

3　提案手法

3.1　概要

本研究では，動的スメルを検出する仕組みをビルドツールである Gradle のプラグインとして実現することを提案する．ビルドツールへの組み込みにより，高い可搬性を実現したうえで，動的スメルを検出可能となる．

3.2　動的スメルの検出

提案手法における動的スメル検出の流れを図 2 に示す．以下，動的スメル検出の流れについて順に説明する．各動的スメル検出手法の詳細は 3.2.1 項，3.2.2 項及び 3.2.3 項に示す．

1. テストコードを静的解析し，メソッド情報を収集する．
2. テストクラスのバイトコードにカバレッジ計測命令を埋め込む.
3. 通常の順序で各改変済みテストを実行する．
4. テストに成功した場合，Rotten Green Test であるかを調べる．失敗した場合，Flaky Test であるかを調べる．
5. 通常の順序でのテスト実行終了後，順序を変更してテスト実行を行い，Dependent Test を検出する．

3.2.1　Rotten Green Test の検出

本研究では，Delplanque ら [6] の手法に従って Rotten Green Test の検出を行う．まず，テストコードを静的解析し，各メソッドが持つ要素（アサーションやメソッド呼び出しなど）を調べる．そして，各テストを実行し，成功した場合にカバレッジデータの解析を行い，テストが未実行のアサーションを含むかを確認する．

3.2.2　Flaky Test の検出

Flaky Test の検出手法は近年数多く提案されている．最も単純な手法としては，複数回テスト実行を繰り返し，その成否が安定するか調べるという手法が挙げられる [13]．その他，コード変更によるカバレッジの変化を監視する方法や，機械学習により失敗したテストを Flaky Test やコード変更によるものなどに分類する方法がある [15][16]．本研究では，テストに失敗した場合に複数回再実行することで Flaky Test の検出を行う．

3.2.3 Dependent Test の検出

Dependent Test は，1 度通常の順序でテストを実行した後，テスト順序を，逆順にする，ランダムに入れ替えるなどして再実行し，テスト結果が通常の順序の場合と異なるかを調べることで検出可能である [14]．本研究では，ランダムな順序で複数回再実行することで Dependent Test の検出を行う．

3.3 Gradle プラグインの実現

3.3.1 ビルドツール統合による可搬性の確保

動的スメルの検出を行う場合，プロダクトコードやテストコードに加え，クラスファイルや外部ライブラリのパスなど様々な情報が必要である．ビルドツールである Gradle はソースフォルダの管理や，依存関係の解決などを自動で行うことが可能である．前述の動的スメル検出手法を Gradle へ組み込むことで，実行に必要なすべての情報を Gradle から取得でき，また，高い可搬性を実現することができる．

3.3.2 実装プラグイン：JTDog

提案手法を実現するためのプロトタイプとして JTDog を実装した．JTDog とは，Java Test Dog の略称で，Gradle の Java プロジェクトに存在する動的スメルを検出するプラグインである．JUnit3/4/5 で記述されたテストに動的スメルが含まれるかを調べ，その結果を JSON ファイルに出力する．JUnit のバージョンは実行時に指定し，デフォルトでは JUnit3/4 のテストを検出対象とする．

JTDog は，動的スメルを検出する `sniff` という名称のタスクを提供する．開発者は，`gradle sniff` のようなコマンドを実行するだけでプロジェクトに存在する動的スメルの検出が可能である．さらに，JTDog は，以下のように，ビルドファイルである `build.gradle` 中の `plugins` ブロックに利用宣言文を 1 行追加するだけで利用できる．

```
plugins {
  id 'com.github.m-tanigt.jtdog' version '1.0.0'
}
```

Gradle プロジェクトでは，クラスパスなどプロジェクト固有の各種ビルド情報は `build.gradle` ファイルに記載済みであるため，JTDog の利用宣言以外に追記の必要はない．

動的スメルの検出は，Gradle API を利用して行う．図 3 に，その処理概要を示す．

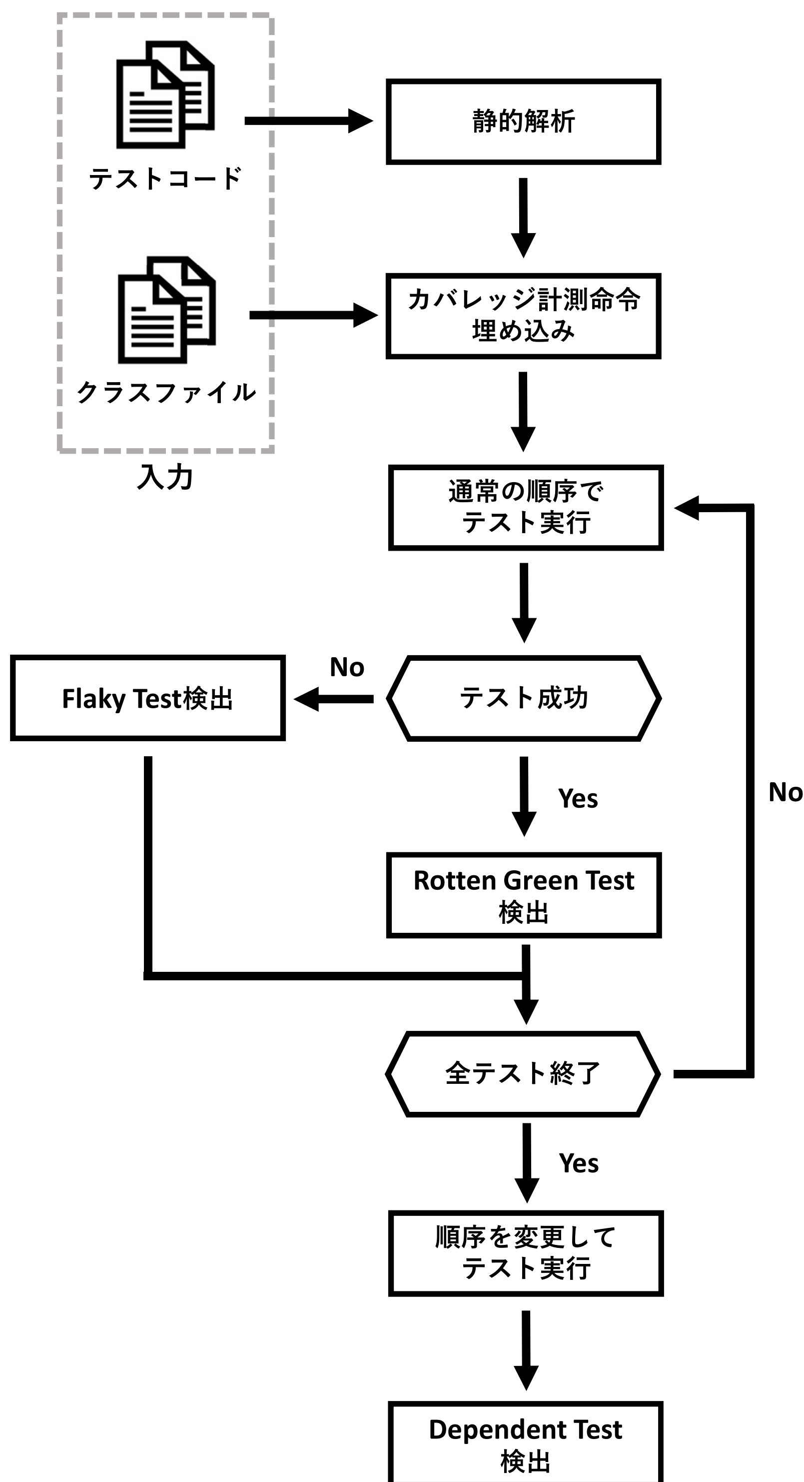

図 2　動的スメル検出の流れ

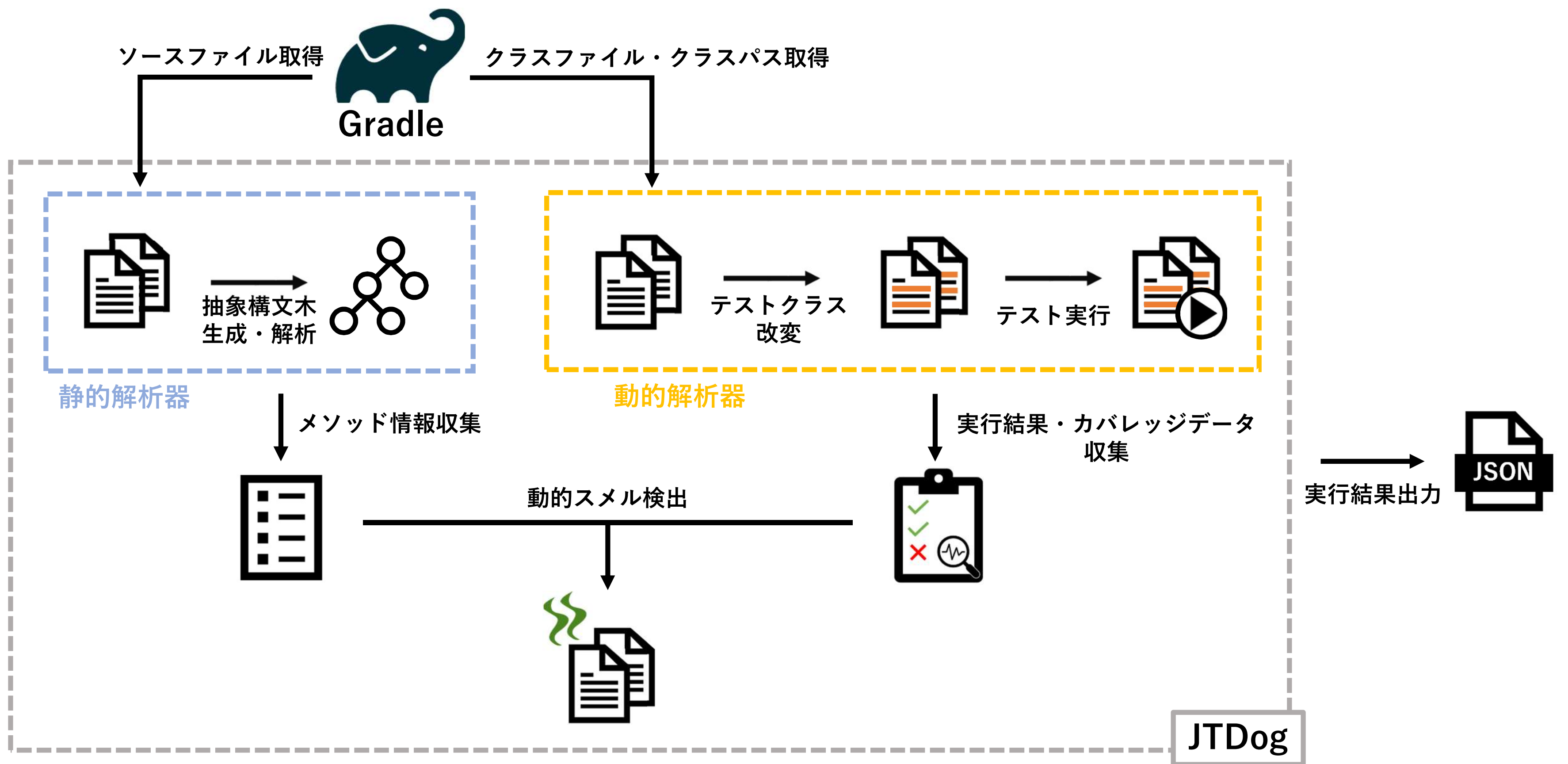

図 3　動的スメル検出処理の概要

Gradle の Java プロジェクトでは，Java のコンパイルやテストを行うために Java プラグインを適用し，ソースセットの概念をプロジェクトに導入する．ソースセットとは，ソースおよびリソースファイルのコンパイルに関する情報を結びつける概念である．ソースセットを利用することで，ソースファイルや，クラスパス，クラスファイルなどを取得することができる．JTDog は，ソースセットからプロダクトコードとテストコードを取得し，テストコードの抽象構文木を生成および解析することでメソッドの情報を収集する．そして，ソースセットからテストクラスファイルとクラスパスを取得し，テストの改変と実行を行うことで実行結果やカバレッジデータを収集する．収集したこれらのデータを合わせることで動的スメルを検出することができる．

4 適用実験

4.1 実験の目的

JTDog の可搬性及び，実際に動的スメルを検出可能かの確認を目的として，GitHub 上の Gradle プロジェクトに対する適用実験を行う．

4.2 適用対象

GitHub 上のプロジェクトから，以下を満たすプロジェクトを選択した．

- ビルドツールとして Gradle を使用している，あるいは，Gradle の変換機能により問題なく Gradle プロジェクトに変換可能な Maven プロジェクトである
- 主要言語が Java であり，Gradle の Java プラグインを適用している
- テストを 1 つ以上含む
- テストフレームワークに JUnit を用いている

上記の条件に加えて，一定以上の人気があるプロジェクトに絞るため，Star が 100 以上であるプロジェクトに限定した．Star とは，GitHub におけるプロジェクトの人気を表す指標の一つであり，Star の数が多いほど，多くの人が当該プロジェクトを利用していると言える．

以上の条件に基づき，150 のプロジェクトを適用対象プロジェクトとして選択した．

4.3 実験方法

適用対象の各プロジェクトについて，3.3.2 項で示したように JTDog の利用宣言文を `build.gradle` ファイルに追記する．そして，`sniff` タスクを実行し，実行結果の JSON ファイルの内容を確認する．

4.4 実験結果

4.4.1 可搬性の確認

適用対象として選択した 150 のプロジェクトに対して JTDog を適用した結果，そのおよそ 81% である 122 のプロジェクトで正常に動作した．プラグイン利用のための一文を追加するのみで動作しており，JTDog は様々なプロジェクトに適用可能であるといえる．

適用に失敗した 28 のプロジェクトの失敗原因を目視により確認した．その内訳を表 2 に示す．表 2 のうち，メモリ不足による失敗の理由を調べたところ，プロジェクトのメソッド数が膨大であることが原因であった．JTDog はメソッドの情報を収集，保持して動的スメルの検出を行うため，メソッド数

が多いほど，情報保持のためにメモリを使用する．ゆえに，不必要になった情報を適宜破棄するように処理を変更するなど，JTDog のリファクタリングによりメモリ不足による失敗を防ぐことが可能であると考えられる．メモリ不足以外の原因については，現状，その理由は把握しきれておらず，今後の詳細な分析が必要である．しかし，ツールの実装に起因する問題は少ないと考えられ，JTDog は高い可搬性を有するといえる．

4.4.2　動的スメルの検出結果

まず，JTDog が正常に動作した 122 のプロジェクトでの動的スメル検出数とスメルを検出したプロジェクトの数を表 3 に示す．この結果から，122 のプロジェクトのうち 55 件でスメルを検出したため，適用に成功したプロジェクトのおよそ 45% は動的スメルを少なくとも 1 つ含んでいることがわかった．また，Rotten Green Test は Flaky Test や Dependent Test と比べ，スメル検出数，検出プロジェクト数のいずれにおいても多い結果となった．これは，Rotten Green Test が ICSE2019 で発表された論文 [6] で紹介された新しいスメルであり，ほとんどの開発者が知らないためと考えられる．Delplanque ら [6] 曰く，テストの実施が不十分であるにも関わらず開発者にテストが問題なく成功したという誤った認識を与えるため，Rotten Green Test はまったくテストをしないよりも悪質である．ゆえに，Rotten Green Test の検出が可能なツールは重要である．

次に，プロジェクトごとの各スメルの傾向を調べる．表 4 に，それぞれのプロジェクトでの Rotten

表 2　JTDog 適用失敗の原因

適用失敗の原因	プロジェクト数
メモリ不足	10
タスク作成失敗	9
検出結果が不正	6
ファイル読み込み失敗	3

表 3　動的スメル検出結果

	検出数	プロジェクト数
Rotten Green Test	781	48
Flaky Test	66	5
Dependent Test	111	13
合計	958	55

Green Test 検出数について，スメル検出数上位 5 件のプロジェクトを抜粋して示す．検出数の多いこれらのプロジェクトは，同じ原因のスメルを多く含んでいた．例えば検出数が最も多い traccar プロジェクトでは，224 個もの Rotten Green Test のほとんどは `if` 文内のアサーションが実行されないことによるものであった．JTDog を使用すれば，開発者はどのようなコードが Rotten Green Test であるかを知ることができ，今後の開発に役立てることができるため，やはり検出ツールは重要であるといえる．

表 5 に，Flaky Test を検出した全 5 プロジェクトでの検出数を示す．http-request プロジェクトでの検出数が突出しているものの，それ以外のプロジェクトではほとんど検出されなかった．これは，JTDog がテストの再実行という最も単純な検出手法を採用していることに加え，この検出手法を Jenkins[17] などのフレームワークがサポートしており，多くのプロジェクトがこのスメルに対処しているためと考えられる．

表 6 に，Dependent Test 検出数上位 5 件のプロジェクトにおけるスメル検出数を示す．いずれのプロジェクトにおいても，テスト間で `static` 変数を共有することが主な原因であった．このようなテストは，同じ `static` 変数を使用するテストの追加などに影響を受け，テスト結果が変わる可能性がある．よって，Dependent Test は早期に検出し対策することが望ましく，JTDog のような検出ツールは重要

表 4　Rotten Green Test 検出プロジェクト（上位 5 件抜粋）

プロジェクト名	検出数
traccar	224
elki	110
Apktool	58
nats.java	50
xodius	39

表 5　Flaky Test 検出プロジェクト（全 5 件）

プロジェクト名	検出数
http-request	59
java-jwt	4
spring-statemachine	1
eureka	1
litiengine	1

```
432    @Test
433    public void c1c3IsDisposed() {
434      toV3Completable(rx.Completable.complete())
435      .subscribe(new io.reactivex.rxjava3.core.CompletableObserver() {
436        @Override
437        public void onSubscribe(io.reactivex.rxjava3.disposables.Disposable d
               ) {
438            assertFalse(d.isDisposed());
439        }
440        ...
441      });
442    }
```

図 4　検出した Rotten Green Test の例[*1]

であるといえる．

4.5　動的スメルの実例

本節では，JTDog が検出した各動的スメルの実例を紹介する．

4.5.1　Rotten Green Test

図 4 に示す RxJavaInterop プロジェクトのテストメソッド `c1c3IsDisposed()` は，Rotten Green Test である．イベントリスナのアクションである onSubscribe() メソッド（437 行目）は内部にアサーションを持つが，このメソッドは実際には実行されない．そのため，438 行目のアサーションによる動作確認が行われず，テストの実施は不十分である．しかし，JUnit は単に成功したテストとして報告するため，開発者がテストに問題があると気づくことは困難である．

表 6　Dependent Test 検出プロジェクト（上位 5 件抜粋）

プロジェクト名	検出数
ehcache3	57
http-request	22
micrometer	11
mybatis-plus	6
testcontainers-java	3

```
84    @Test
85    public void shouldPassHMAC256Verification() throws Exception {
86      Algorithm algorithm = Algorithm.HMAC256("secret");
87      JWTVerifier verifier = JWTVerifier.init(algorithm).withIssuer("auth0").
            build();
88      String token = ...;
89
90      concurrentVerify(verifier, token);
91    }
```

図 5　検出した Flaky Test の例[*2]

4.5.2 Flaky Test

図 5 に示す java-jwt プロジェクトのテストメソッド `shouldPassHMAC256Verification()` は，Flaky Test である．アサーションを含む `concurrentVerify()` メソッド（90 行目）実行時に，`RejectedExecutionException` という例外が発生し，テストが失敗する場合がある．開発者は，この例外が発生した原因が，プロダクトとテストのいずれにあるかを調べるために時間を費やすこととなる．さらに，テスト結果が不安定なだけと考え，失敗原因を特定しない場合，プロダクトのバグを見逃す可能性がある．

4.5.3 Dependent Test

図 6 に jwt-spring-security-demo プロジェクトに含まれる Dependent Test である `getCurrentUser-nameForNoAuthenticationInContext()`（25 行目）と，このテストが依存するテストである `getCurrentUsername()`（14 行目）を示す．通常は `getCurrent...Context()` が先に実行されるが，実行順を入れ替えると，`getCurrentUsername()` により `static` 変数 `username` の値が`"admin"`に設定されるため，29 行目でアサーションエラーが発生する． JUnit4 はデフォルトでは実行順序をテストメソッド名のハッシュ値に基づいて決定する[*4]．そのため，これらのテストは偶然どちらも成功する順番で実行されているだけであり，メソッド名の変更やテストの追加などで実行順序が変わると，`getCurrent...Context()` のテストは失敗する．しかし，開発者はすぐには実行順序の変更がテスト

[*1] https://github.com/akarnokd/RxJavaInterop/blob/3.x/src/test/java/hu/akarnokd/rxjava3/interop/RxJavaInteropTest.java#L432
[*2] https://github.com/auth0/java-jwt/blob/master/lib/src/test/java/com/auth0/jwt/ConcurrentVerifyTest.java#L84
[*3] https://github.com/szerhusenBC/jwt-spring-security-demo/blob/master/src/test/java/org/zerhusen/security/SecurityUtilsTest.java#L25
[*4] https://github.com/junit-team/junit4/wiki/Test-execution-order

```
14    @Test
15    public void getCurrentUsername() {
16      SecurityContext securityContext = SecurityContextHolder.
          createEmptyContext();
17      securityContext.setAuthentication(new
          UsernamePasswordAuthenticationToken("admin", "admin"));
18      SecurityContextHolder.setContext(securityContext);
19
20      Optional<String> username = SecurityUtils.getCurrentUsername();
21
22      assertThat(username).contains("admin");
23    }
24
25    @Test
26    public void getCurrentUsernameForNoAuthenticationInContext() {
27      Optional<String> username = SecurityUtils.getCurrentUsername();
28
29      assertThat(username).isEmpty();
30    }
```

図 6　検出した Dependent Test の例*3

の失敗とは考えないため，原因の特定に時間を費やすこととなる．

5　おわりに

本研究では，テストスメルを静的スメルと動的スメルに大別し，その性質を整理したうえで，動的スメル検出手法を実現するために Gradle プラグイン JTDog を実装した．適用実験として GitHub 上の 150 プロジェクトに対して JTDog を使用し，本プラグインの可搬性および検出機能の確認を行った．

今後の課題としては，JTDog に静的スメルの検出機能を追加し，広範にテストスメルを検出可能にすることや，JTDog の適用失敗理由の詳細分析，フレームワークとして再編することによる拡張性の向上が考えられる．その他，本実験を行う中で新たに発見したスメルの詳細な定義とその検出や，GitHub 上のプロジェクトから検出したスメルについて，開発者にプルリクエストを送ることによってツールの評価を行うなどが挙げられる．

謝辞

本研究を行うにあたり，多くの方々のご支援，ご協力を賜りました．

研究の中間報告では，楠本真二教授，肥後芳樹准教授より，貴重なご指導とご助言を賜りました．ここに感謝の意を表します．

主指導教員である柗本真佑助教には，研究の方針や構想，実現方法，調査，論文執筆と多くのご指導を賜りました．心より感謝申し上げます．

最後に，研究を進めるにあたり，様々な形でご協力いただいた楠本研究室の皆様にお礼申し上げます．

参考文献

[1] K. Beck. *Test-Driven Development: By Example.* Addison-Wesley Professional, 2003.

[2] M. Fowler. Continuous Integration, accessed 2021-02-05. https://martinfowler.com/articles/continuousIntegration.html.

[3] D. Athanasiou, A. Nugroho, J. Visser, and A. Zaidman. Test Code Quality and Its Relation to Issue Handling Performance. *IEEE Transactions on Software Engineering*, Vol. 40, No. 11, pp. 1100–1125, 2014.

[4] A.V. Deursen, L. Moonen, and A. Berghand G. Kok. Refactoring Test Code. In *International Conference on Extreme Programming and Flexible Processes in Software Engineering*, pp. 92–95, 2001.

[5] G. Meszaros. *xUnit test patterns: Refactoring test code.* Pearson Education, 2007.

[6] J. Delplanque, S. Ducasse, G. Polito, A. P. Black, and A. Etien. Rotten Green Tests. In *International Conference on Software Engineering*, pp. 500–511, 2019.

[7] B. Van Rompaey, B. Du Bois, and S. Demeyer. Characterizing the Relative Significance of a Test Smell. In *International Conference on Software Maintenance*, pp. 391–400, 2006.

[8] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley. An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In *International Conference on Software Maintenance*, pp. 56–65, 2012.

[9] G. Bavota, A. Qusef, R. Oliveto, A. Lucia, and D. Binkley. Are Test Smells Really Harmful? An Empirical Study. *Empirical Software Engineering*, Vol. 20, No. 4, p. 1052–1094, 2015.

[10] A. Peruma. What the Smell? An Empirical Investigation on the Distribution and Severity of Test Smells in Open Source Android Applications. Master's thesis, Rochester Institute of Technology, 2018.

[11] A. Peruma, K. Almalki, C.D. Newman, M.W. Mkaouer, A. Ouni, and F. Palomba. *tsDetect: An Open Source Test Smells Detection Tool*, p. 1650–1654. Association for Computing Machinery, 2020.

[12] M. Greiler, A. van Deursen, and M. Storey. Automated Detection of Test Fixture Strategies and Smells. In *International Conference on Software Testing, Verification and Validation*, pp. 322–331, 2013.

[13] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov. An Empirical Analysis of Flaky Tests. In *International Symposium on Foundations of Software Engineering*, pp. 643–653, 2014.

[14] S. Zhang, D. Jalali, J. Wuttke, K. Muşlu, W. Lam, M.D. Ernst, and D. Notkin. Empirically Revisiting the Test Independence Assumption. In *International Symposium on Software Testing and Analysis*, p. 385–396, 2014.

[15] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov. DeFlaker: Automatically Detecting Flaky Tests. In *International Conference on Software Engineering*, pp. 433–444, 2018.

[16] K. Herzig and N. Nagappan. Empirically Detecting False Test Alarms Using Association Rules. In *International Conference on Software Engineering*, p. 39–48, 2015.

[17] Jenkins RandomFail annotation, accessed 2021-02-05. https://github.com/jenkinsci/jenkins-test-harness/blob/master/src/main/java/org/jvnet/hudson/test/RandomlyFails.java.