JTDog: 動的テストスメル検出のためのGradleプラグイン

谷口 真幸　柗本 真佑　楠本 真二

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，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のに，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．本論文では，動的スメルの検出を行うGradleプラグインであるJTDogを提案する．ビルドツールへの組み込みにより，JTDogは高い可搬性を有している．GitHub上の205のプロジェクトに対する適用実験の結果，JTDogの可搬性を確認することができた．また，実際に66のプロジェクトから1,117個の動的スメルを検出できた．

The concept of the test smell represents potential problems with the readability and maintainability of the test code. Common test smells focus on static aspects of the source code, such as code length and complexity. These are easy to detect and do not cause problems in terms of test execution. On the other hand, dynamic smells, which are based on test runtime behavior, lead to misunderstanding of the test results. For example, they give developers the false impression that the test was passed without any problems, even though the test was poorly executed. Therefore, we should detect dynamic smells and take countermeasures as early as possible through the development. In this paper, we propose JTDog, a Gradle plugin for detecting dynamic smells. JTDog has high portability due to its integration into the build tool. We applied JTDog to 205 projects on GitHub and confirmed that the JTDog plugin has high portability. In addition, JTDog detected 1,117 dynamic smells in 66 projects.

1　はじめに

プロダクトコードにおけるコードスメルと同様，テストコードにおける設計上の問題を示す指標としてテストスメルが知られている [10]．テストスメルは可読性や保守性などに対する潜在的な問題を示唆しており，テスト品質の低下につながる恐れがある．そのため，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 [20]．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，テストコード固有のスメルも存在する．例えば，テストコード内での制御文の使用箇所は一種のスメルと見なされる．制御文はテストの複雑化を招くうえ，テストコード自体のバグにつながる可能性があるためである [20]．

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いるが，テスト実

JTDog: a Gradle Plugin for Detecting Dynamic Test Smells
Masayuki Taniguchi, Shinsuke Matsumoto, Shinji Kusumoto, 大阪大学大学院情報科学研究科, Graduate School of Information Science and Technology, Osaka University.
コンピュータソフトウェア, Vol.39, No.4 (2022), pp.50–60.
[ソフトウェア論文] 2021年11月29日受付.

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 [9] 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，当該アサーションによる動作確認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てしまう．このように，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，テストの品質に重大な影響を及ぼす．また，動的スメルを検出するためには，実行時解析のために対象プロジェクトの実行情報（クラスファイルや外部ライブラリへのパスなど）の明示的な指定が必要となる．そのため，素朴な設計の動的スメル検出ツールは環境と密に結合し，可搬性が低くなる傾向にある．

これまでに数多くのテストスメルに関する研究 [2,3,24,26] が実施されており，TestQ [7] や tsDetect [25] など，複数種の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．しかし，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．さらに，動的スメルを検出するツールも提案されているが [5,19,28]，いずれも単一の動的スメルを対象としており，我々の知る限り，複数種の動的スメルをまとめて検出するツールは存在しない．

本論文では，高い可搬性を有し，同時に複数種の動的スメルの検出が可能である動的スメル検出ツール JTDog を提案する．JTDog はビルドツールである Gradle のプラグインとして組み込んだため，Gradle を利用するプロジェクトであれば容易に適用できる高い可搬性を有している．また，JTDog は単一コマンドで即座に実行可能であるため，CI（継続的インテグレーション）のような自動化技術との親和性も高い．JTDog は 3 つの動的スメル Rotten Green Test [9]，Flaky Test [18]，Dependent Test [28] を検出する．1 つのツールで同時に複数種の動的スメルを検出することができるため，開発者の生産性向上につながる．

JTDog の可搬性を確かめるため，GitHub 上の 205 のプロジェクトに対して適用実験を行った．その結果，171 のプロジェクトでプラグインが正常に動作し，そのうち 66 のプロジェクトから合計 1,117 個の動的スメルを検出できた．

JTDog は GitHub 上で OSS として開発しており [†1]，プラグイン自体も Gradle Plugin Portal で公開している [†2]．

2 動的スメル検出の自動化

本章では，まず具体的な動的スメルの特徴と及ぼす影響について紹介する．さらに複数種の動的スメル検出ツールの設計にあたっての課題と問題点について議論する．

2.1 動的スメル

動的スメルは，テスト実行の経路，順序，結果など，テスト実行時の振る舞いに基づいたスメルである．動的スメルの一種として，開発者に十分にテストしているという印象を与えながら，実際にはほとんどバグを検出できないようなパターンも知られており，テストの品質に重大な影響を及ぼすことがある．動的スメルを検出するためには，実行時解析のために対象プロジェクトの実行情報（クラスファイルや外部ライブラリへのパスなど）の明示的な指定が必要となる．そのため，素朴な設計の動的スメル検出ツールは環境と密に結合し，可搬性が低くなる傾向にある．現状，Rotten Green Test [9]，Flaky Test [18]，Dependent Test [28] の 3 種が存在する．

2.1.1 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．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

†1 https://plugins.gradle.org/plugin/com.github.m-tanigt.jtdog

†2 https://github.com/kusumotolab/JTDog

```
1  //誤ったテストデータ
2  URL data = getResource("testData.xml");
3  DataBase db = new DataBase(data);
4  //実際は空のリスト
5  List<User> users = db.getUsers();
6  for(User user : users){
7    //実行されない
8    assertNotNull(user.getName());
9  }
```

図 1　Rotten Green Test の例

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，2 行目のテストデータが誤っており，5 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，8 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．

2.1.2　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 [18].

2.1.3　Dependent Test

Dependent Test 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．

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 Dependent Test 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 [28].

2.2　既存のテストスメル検出ツールの問題点

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 [2,3,24,26]，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．しかし，動的スメルの検出に焦点を当てると，既存のテストスメル検出ツールには問題がある．

まず，既存の動的スメル検出ツール [5,19,28] は利用のための環境設定が煩雑である．動的スメル検出には実行時解析が必要であり，ツール利用のために対象プロジェクトの実行情報（クラスファイルや外部ライブラリへのパスなど）の明示的な指定が必要なためである．これら実行情報の継続的な保守はソフトウェア進化における必須工程の一つであり，環境設定の煩雑さは動的スメル検出ツールの利用を阻害する要因となりうる．ひいてはソフトウェア全体の品質低下につながる可能性がある．

さらに，複数種の動的スメルを一度に検出可能なツールは存在しないという問題もある．複数種のテストスメル検出を目的としたツールは TestQ [7] や tsDetect [25] などが存在するが，どちらも静的スメルの検出に限定されている．また動的スメルがテスト実行を必要とする性質から，これら既存ツールの動的スメル検出への拡張は容易とはいえない．動的スメルの検出という観点では，RTj [19] や DTDetector [28] が存在するが，それぞれ単一の動的スメル検出を目的としている．よって，3 種の動的スメルをすべて検出するために既存の動的スメル検出ツールを利用する場合，それぞれのツールについて煩雑な環境設定が必要なため，開発者の生産性低下，もしくはテスト品質の低下につながる恐れがある．

3　JTDog

JTDog は，Java プロジェクトを対象とした動的なテストスメル検出のための Gradle プラグインである．Gradle への組み込みにより，高い可搬性を有している．単一コマンドで実行可能であるため，CI のような自動化技術とも親和性が高い．JTDog は，3 種類の動的スメル Rotten Green Test [9]，Flaky Test [18]，Dependent Test [28] を検出し，その結果を JSON ファイルに出力する．高可搬性の実現によって利用

が容易であるうえ，テストの品質に重大な影響を及ぼす動的スメルをまとめて検出できるため，JTDog は開発者の生産性向上に寄与する．

3.1 使用方法

JTDog の利用に当たっては，まず開発者はビルドファイルである `build.gradle` 内に以下のように JTDog の利用を宣言する．

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

この宣言により，動的スメルを検出する `sniff` という名称の Gradle タスクが提供される．プロジェクトの実行情報は `build.gradle` 内に記述されているため，JTDog はその内容を自動的に読み取り，コンパイルやテスト実行などの動的スメル検出に必要な処理を実行する．開発者は `gradle sniff` というコマンドを 1 行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．

3.2 設計

3.2.1 動的スメルの検出

JTDog は 3 つの動的スメル Rotten Green Test [9], Flaky Test [18], Dependent Test [28] を検出する．これら複数種の動的スメルを単一ツールで検出することで，動的スメル検出にかかる労力の削減を狙う．本項では，各スメルの検出手法を示す．

Rotten Green Test Delplanque ら [9] 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．

Flaky Test 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，テストを複数回再実行し，その成否が安定するか調べるという手法が挙げられる [18]．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，機械学習を用いて失敗したテストを分類する方法がある [5, 14]．

JTDog は，テストを複数回再実行することで Flaky Test の検出を行う．

一般的には，テストの再実行による検出では失敗テストのみが再実行されることが多い [18]．これは，検出のためのテスト実行コストを低減するためである．例えば Google では，Flaky Test 検出のためのコストを最小化するために，失敗テストのみを再実行の対象としたうえで，さらにその中で問題があると思われるテストのみを再実行している [21, 22]．本研究では正確な Flaky Test 検出のために，失敗テストだけでなく成功テストも再実行する．

テストの再実行回数はデフォルトで 10 回であり，JTDog の利用者による変更も可能である．この 10 回という値は既存の Flaky Test 検出研究 [18] を参考にした．

Dependent Test 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という検出方法が知られている [28]．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．Flaky Test の検出と同様，テストの再実行回数はデフォルトで 10 回で，JTDog の利用者により変更できる．

3.2.2 ワークフロー

前項に示した検出手法を用いて，以下の手順で動的スメルの検出を行う．

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. 成功したテストについて，計測した実行経路情報を用いて未実行アサーションの有無を調べ，Rotten Green Test か否かを判別する．
5. テストを通常の順序で複数回再実行し，テスト結果に変化が生じるかを調べ，Flaky Test を検出する．

6. 通常の順序でのテスト実行終了後，ランダムに順序を変更してテスト実行を行い，Dependent Test を検出する．

3.3 実装

3.3.1 動的スメルの検出

動的スメルの検出のため，まず，Eclipse JDT を利用してテストコードの AST を生成し，これを解析してメソッド情報を収集する．その後，JaCoCo を用いて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み，JUnit を用いて通常の順序でテストを実行する．そして，成功したテストについて，Rotten Green Test であるかを調べる．事前に収集したメソッド情報から該当テストメソッドが含むアサーションの行番号を取得し，その行の命令が実行されているかをカバレッジデータから取得することで，未実行のアサーションを含むかを判断できる．なお，テストがメソッド呼び出しを含む場合，そのメソッドについても再帰的に未実行のアサーションを含むかを調べる．

次に，通常の順序でテストを再実行するという処理を複数回行う．この再実行において 1 度でもテスト結果に変化が生じた場合，当該テストが Flaky Test と判断できる．なお，テスト実行はそれぞれ別 JVM 上で実施する．これは，同一 JVM 上で再実行する場合，複数テスト間での static 変数の共有に起因する Dependent Test など，JVM の状態に依存する Dependent Test のテスト結果に変化が生じ，Flaky Test として誤検出する恐れがあるためである．

通常の順序でのテスト実行終了後，順序をランダムに入れ替えてのテスト実行を複数回行う．Flaky Test の検出と同様，テスト実行はそれぞれ別 JVM 上で実施する．各テストメソッドについて，テスト結果を通常のテスト実行時の結果と比較し，異なる場合に Dependent Test と判断する．ただし，このテストが既に Flaky Test として判定されている場合は，順序に依存せずにテスト結果が変わるため，Dependent Test ではないと判断する．

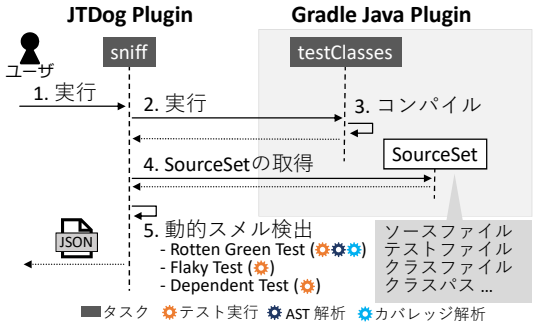


図 2 Gradle API を利用した検出処理の概要

3.3.2 Gradle の利用

前述の動的スメルの検出処理を行うためには，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なため，動的スメル検出技術を 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，高い可搬性を実現できる．

図 2 に Gradle API を利用した検出処理の概要を示す． Gradle を利用した 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 プラグインを適用する．JTDog が提供する `sniff` タスクは，Java プラグインが提供する `testClasses` タスクに依存している．これにより，`sniff` タスクを実行する際，先に `testClasses` タスクが実行され，Java ファイルのコンパイルなどの前処理が行われる．

また，Java プラグインはソースセット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ファイル，クラスパスなどを取得することができる．JTDog はまず，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テスト結果やカバレッジデータを取得する．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

JTDog の可搬性及び動的スメル検出能力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．

4.1 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．以上の条件に基づき，205 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．

4.2 実験方法

適用対象の各プロジェクトについて，3.1 節で示したように 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，検出結果が出力された JSON ファイルの内容を確認する．

なお，Gradle では JVM の最大ヒープサイズを標準で 1024MB としているが，本実験では 2048MB とした．これは，非常に大規模なビルドの場合，Gradle のシステム動作のためにより大きなメモリが必要な場合があり，本実験では様々な規模の OSS を対象とするためである．

4.3 実験結果

4.3.1 可搬性の確認

適用対象として選択した 205 のプロジェクトに対して JTDog を適用した結果，そのおよそ 83%である 171 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 34 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 1 に示す． 表 1 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．前述の通り，本実験では JVM の最大ヒープサイズを Gradle の標準設定よりも大きい 2048MB としたが，それにもかかわらずヒープ領域のメモリ不足が発生した．この事実と，JTDog が動的スメル検出の際にメソッド情報を収集・保持し，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ことを考えると，現在の実装では非常に大規模なプロジェクトへの JTDog の適用は難しいと考えられる．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．

4.3.2 動的スメルの検出結果

まず，JTDog が正常に動作した 171 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 2 に示す． この結果より，適用に成功した 171 のプロジェクトのうち 66 件で動的スメルを検出したことがわかる．そして，検出した 3 種の動

表 1 JTDog 適用失敗の原因

適用失敗の原因	プロジェクト数
タスク作成失敗	10
メモリ不足	9
検出結果が不正	6
静的/動的解析が実行されない	6
ファイル読み込み失敗	3

表 2 動的スメル検出結果

	検出数	プロジェクト数
Rotten Green Test	806	55
Flaky Test	181	10
Dependent Test	130	14
合計	1,117	66

的スメルについて，それぞれサンプリングし，動的スメルであることを目視で確認したため，JTDog は動的スメル検出能力を十分有していると言える．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 2019 年に発表された論文 [9] で紹介された新しいスメルであり，その存在が周知されていないことが原因の一つであると考えられる．Rotten Green Test は，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，まったくテストをしないよりも悪質である [9]．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 3，4，5 に，それぞれのプロジェクトでの各動的スメルの検出数について，スメル検出数上位 5 件のプロジェクトを抜粋して示す．Rotten Green Test について，表 3 に示す検出数の多いプロジェクトは，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．

表 4 の Flaky Test の検出数について，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ためと考えられる．

表 3　Rotten Green Test 検出プロジェクト

プロジェクト名	検出数
traccar/traccar	224
elki-project/elki	110
iBotPeaches/Apktool	58
nats-io/nats.java	50
JetBrains/xodius	39

表 5 の Dependent Test 検出数上位 5 件のプロジェクトに関して，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であるといえる．

4.4　動的スメルの検出例

本節では，JTDog が検出した各動的スメルの実例を紹介する．

4.4.1　Rotten Green Test

図 3 に示す RxJavaInterop プロジェクトのテストメソッド `c1c3IsDisposed()` は，JTDog が実際に検出した 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，このメソッドは実行されない．そのため，439 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．このようなコールバック処理は，単純な手続き

表 4　Flaky Test 検出プロジェクト

プロジェクト名	検出数
SonarSource/sonarlint-intellij	162
cc-tweaked/CC-Tweaked	5
auth0/java-jwt	4
ZZMarquis/gmhelper	3
bulldog2011/bigqueue	2

表 5　Dependent Test 検出プロジェクト

プロジェクト名	検出数
ehcache/ehcache3	57
kevinsawicki/http-request	28
quicktheories/QuickTheories	12
micrometer-metrics/micrometer	11
baomidou/mybatis-plus	6

```
@Test
public void c1c3IsDisposed() {
  toV3Completable(Completable.complete())
  .subscribe(new CompletableObserver() {
    @Override
    public void onSubscribe(Disposable d) {
        //実行されない
        assertFalse(d.isDisposed());
    }
    ...
  });
}
```

図 3 検出した Rotten Green Test の例 †3

的な記述と比べて実際に呼び出されるかどうかの判断が難しく，Rotten Green Test となる可能性が高いと考えられる．JTDog は，こういったコードレビューでは気づきにくい問題の検出を支援する．

4.4.2 Flaky Test

図 4 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．`concurrentVerify()` メソッド（93 行目）内でマルチスレッド処理を行う際に `RejectedExecutionException` という例外が発生してテストが失敗する場合がある．この例外は，マルチスレッド処理において各タスクを実行する ThreadPoolExecutor が，何らかの理由で，シャットダウンしている，またはスレッド数が最大かつタスク実行のキューが飽和状態である場合に発生する．開発者は失敗の理由を特定する必要があるが，実行環境など，テスト以外に原因がある可能性もあり，多大な労力を費やす必要があると考えられる．しかし，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．そのため，検出ツールにより，早期に発見することが望ましい．

4.4.3 Dependent Test

図 5 に jwt-spring-security-demo プロジェクトに含まれる Dependent Test である `getCurrentUsernameForNoAuth()`（24 行目）と，このテストが依存するテスト `getCurrentUsername()`（14 行目）を示す．

```
@Test
public void shouldPassHMAC256Verification()
      throws Exception {
  Algorithm algorithm = HMAC256("secret");
  JWTVerifier verifier = ...;
  String token = ...;

  concurrentVerify(verifier, token);
}

@SuppressWarnings("Convert2Lambda")
private void concurrentVerify(...) throws
      Exception {
  final Waiter waiter = new Waiter();
  List<VerifyTask> tasks = ...;
  executor.invokeAll(tasks, TIMEOUT, ...);
  waiter.await(TIMEOUT, REPEAT_COUNT);
}
```

図 4 検出した Flaky Test の例 †4

```
@Test
public void getCurrentUsername() {
  SecurityContext securityContext = ...;
  securityContext.setAuth("admin", "admin");
  SecurityContextHolder.setContext(...);
  String username = getCurrentUsername();

  assertThat(username).contains("admin");
}

@Test
public void getCurrentUsernameForNoAuth() {
  String username = getCurrentUsername();

  assertThat(username).isEmpty();
}
```

図 5 検出した Dependent Test の例 †5

通常は `getCurrentUsernameForNoAuth()` 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，28 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 †6．そのため，これらのテストは偶然どちらも成功する順序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

†3 https://github.com/akarnokd/RxJavaInterop/blob/ce111/src/test/java/hu/akarnokd/rxjava3/interop/RxJavaInteropTest.java#L432

†4 https://github.com/auth0/java-jwt/blob/90bce/lib/src/test/java/com/auth0/jwt/ConcurrentVerifyTest.java#L84

†5 https://github.com/szerhusenBC/jwt-spring-security-demo/blob/dcf71/src/test/java/org/zerhusen/security/SecurityUtilsTest.java#L25

†6 https://github.com/junit-team/junit4/wiki/Test-execution-order

順序が変わると，`getCurrentUsernameForNoAuth()` 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．JTDog を利用すれば，このような潜在的問題を検出することができる．

5 関連研究

現在までに数多くのテストスメル検出ツールが提案されている．Breugelmans と Rompaey は，検出ツールとして TestQ を開発した [7]．このツールは，開発者がテストスイートを視覚的に探索するためのインターフェースを提供する．Greiler らは，テストフィクスチャに関連する新しいテストスメルを導入し，検出ツールである TestHound を構築した [12]．Greiler らはその後の研究で，TestHound を Git や SVN のリポジトリを解析できるように拡張し，TestEvoHound を作成した [13]．Palomba らは，テキスト分析によって 3 種類のテストスメルを検出するツール TASTE を開発した [23]．Bavota らは 9 種類のテストスメルを検出するテストスメル検出ツールを作成し [4]，Peruma らは，19 種類のテストスメルを検出可能なツール tsDetect を提案した [25]．これらのツールはいずれも静的スメルのみ検出することができ，JTDog と補完関係にある．双方のツールを組み合わせることで，より良いテストの作成を支援できる．

我々が動的スメルに分類した 3 つのテストスメルそれぞれについて，その検出を目的としたツールも提案されている．Delplanque らは，新たなテストスメルとして Rotten Green Test を提案し，検出ツールとして DrTest を考案した [9]．DrTest は Pharo という言語で記述されたプログラムを対象とするため，Martinez らは，Java プログラムを対象とした Rotten Green Test 検出ツールである RTj を開発した [19]．Dependent Test を検出するために，Zhang らは DTDetector を [28]，Gambi らは PRADET を構築した [11]．Bell らは，Flaky Test の新たな検出手法を提案し，DeFlaker を実装した [5]．これらはすべてスタンドアロンツールである．我々は各動的スメルの検出機能を統合して Gradle プラグインとした．ビルドツールへの統合により高可搬性を有し，1 つのツールですべての動的スメルを検出することができるため，開発者の生産性向上につながる．

テストスメル検出手法の IDE への統合も行われている．Bleser らは，6 種類のテストスメルを検出できる IntelliJ プラグイン SoCRATES を開発した [6]．Lambiase らは，3 種類のテストスメルを検出し，自動リファクタリングを行うことができる IntelliJ プラグイン DARTS を構築した [17]．Eclipse プラグインも作成されており，Baker らは TReX を [1]，Santana らは RAIDE [27] を開発した．これらのプラグインについても，自動リファクタリングがサポートされている．また，Koochakzadeh と Garousi が開発した TeReDetect [15] と TeCReVis [16] は，CodeCover [8] という Eclipse プラグインに統合されている．テストスメル検出技術を IDE に統合することで，ユーザのテストスメル検出のコストを削減することができる．同様に，JTDog は，動的スメル検出手法を Gradle に統合することで，ユーザが容易にテストスメルを検出できるよう支援する．

6 おわりに

本論文では，動的なテストスメルを検出する Gradle プラグイン JTDog を紹介した．適用実験では GitHub 上の 205 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．その結果，JTDog は 66 プロジェクトから 1,117 個の動的スメルを検出することができた．今後の課題としては，動的スメル検出手法の改善や，JTDog に静的スメルの検出機能を追加して広範なテストスメル検出ツールとすること，TestNG などの他の Java テストフレームワークのテストを解析を可能にすることが挙げられる．

謝辞 本研究の一部は，JSPS 科研費 JP21H04877 および JP21K11829 による助成を受けた．

参 考 文 献

[1] Baker, P., Evans, D., Grabowski, J., Neukirchen, H., and Zeiss, B.: TRex - The Refactoring and Metrics Tool for TTCN-3 Test Specifications, *Testing: Academic & Industrial Conference - Practice And Research Techniques*, 2006, pp. 90–94.

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

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

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

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

[6] Bleser, J. D., Nucci, D. D., and Roover, C. D.: SoCRATES: Scala Radar for Test Smells, *Symposium on Scala*, 2019, pp. 22–26.

[7] Breugelmans, M. and Rompaey, B. V.: TestQ: Exploring Structural and Maintenance Characteristics of Unit Test Suites, *International Workshop on Advanced Software Development Tools and Techniques*, 2008.

[8] CodeCover: `http://codecover.org/index.html`.

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

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

[11] Gambi, A., Bell, J., and Zeller, A.: Practical Test Dependency Detection, *International Conference on Software Testing, Verification and Validation*, 2018, pp. 1–11.

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

[13] Greiler, M., Zaidman, A., van Deursen, A., and Storey, M.: Strategies for avoiding text fixture smells during software evolution, *Working Conference on Mining Software Repositories*, 2013, pp. 387–396.

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

[15] Koochakzadeh, N. and Garousi, V.: A Tester-Assisted Methodology for Test Redundancy Detection, *Advances in Software Engineering*, Vol. 2010(2010).

[16] Koochakzadeh, N. and Garousi, V.: TeCReVis: A Tool for Test Coverage and Test Redundancy Visualization, *Testing – Practice and Research Techniques*, 2010, pp. 129–136.

[17] Lambiase, S., Cupito, A., Pecorelli, F., Lucia, A. D., and Palomba, F.: Just-In-Time Test Smell Detection and Refactoring: The DARTS Project, *International Conference on Program Comprehension*, 2020, pp. 441–445.

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

[19] Martinez, M., Etien, A., Ducasse, S., and Fuhrman, C.: RTj: A Java Framework for Detecting and Refactoring Rotten Green Test Cases, *International Conference on Software Engineering*, 2020, pp. 69–72.

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

[21] Micco, J.: Flaky Tests at Google and How We Mitigate Them, `https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html`, 2016.

[22] Micco, J.: The State of Continuous Integration Testing Google, `https://research.google.com/pubs/pub45880.html`, 2017.

[23] Palomba, F., Zaidman, A., and Lucia, A. D.: Automatic Test Smell Detection Using Information Retrieval Techniques, *International Conference on Software Maintenance and Evolution*, 2018, pp. 311–322.

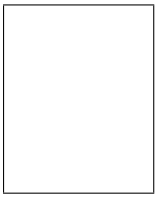
[24] Peruma, A.: What the Smell? An Empirical Investigation on the Distribution and Severity of Test Smells in Open Source Android Applications, Master's thesis, Rochester Institute of Technology, 2018.

[25] Peruma, A., Almalki, K., Newman, C., Mkaouer, M., Ouni, A., and Palomba, F.: tsDetect: An Open Source Test Smells Detection Tool, *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Association for Computing Machinery, 2020, pp. 1650–1654.

[26] Rompaey, B. V., Bois, B. D., and Demeyer, S.: Characterizing the Relative Significance of a Test Smell, *International Conference on Software Maintenance*, 2006, pp. 391–400.

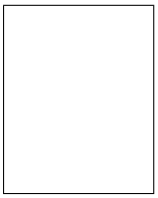
[27] Santana, R., Martins, L., Rocha, L., Virgínio, T., Cruz, A., Costa, H., and Machado, I.: RAIDE: A Tool for Assertion Roulette and Duplicate Assert Identification and Refactoring, *Proceedings of the 34th Brazilian Symposium on Software Engineering*, 2020, pp. 374–379.

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



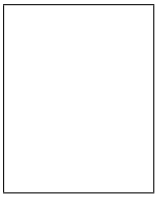
谷口真幸

2021 年大阪大学基礎工学部情報科学科卒業．同年より同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博士前期課程在学中．ソフトウェアテストの分析に関する研究に従事．



柗本真佑

2010 年奈良先端科学技術大学院大学博士後期課程修了．同年神戸大学大学院システム情報学研究科特命助教．2016 年大阪大学大学院情報科学研究科助教．博士（工学）．エンピリカルソフトウェア工学の研究に従事．



楠本真二

1988 年大阪大学基礎工学部卒業．1991 年同大学大学院博士課程中退．同年同大学基礎工学部助手．1996 年同講師．1999 年同助教授．2002 年同大学大学院情報科学研究科助教授．2005 年同教授．博士（工学）．ソフトウェアの生産性や品質の定量的評価に関する研究に従事．電子情報通信学会，IEEE，IFPUG 各会員．