移植可能性に着目した再利用によるJavaテストコード自動生成に向けた調査

Introduction

ソフトウェアの品質を高く保つことは広く要求されている。例えばリリース後にソフトウェアの不具合が明らかになった場合、ユーザの安全への脅威や経済的損失の発生、加えて開発者に対するユーザの信頼低下を招く恐れがある。 これを抑制するため、ソフトウェアの不具合をリリース前に検出するソフトウェアテストが広く行われている。

ソフトウェア開発黎明期、全てのテストは手動による入力および出力の目視確認によって行われていた[1]。 この労力コストを軽減すべく、テストを行うためのプログラムを記述するテストコードが作成されるようになった[1]。 テストコードには通常、テスト対象プログラムへの入力あるいは操作とその期待出力が記述され、実際の出力と期待出力を照合することでテストの成否を判断する。 開発者はテストコードを作成し実行することで自動的に何度でもテストを行うことができ、ソフトウェアテストに伴うコストを軽減できる[2]。

テスト対象が正しく動作しているか確認する

> [1] J. Meerts, “The History of Software Testing”. http://www.testingreferences.com/testinghistory.php
> [2] A. Orso and G. Rothermel, “Software Testing: A Research Travelogue (2000–2014),” Proceedings of the on Future of Software Engineering, pp.117–132, FOSE 2014, ACM, New York, NY, USA, 2014.

こうした自動テストを支援する様々な技術が生まれている。 例えばJavaの単体テストフレームワークであるJUnitを使用することで関数単位のテストコードを簡潔に記述することができる[3]。 また、Eclipse等のIDEではテストフレームワークによって記述されたテストを自動的に検出でき、開発者によるテストを行いやすくしている[要引用]。 Maven等のビルド管理ツールではビルド時にテストコードによるテストを自動的に実行することができる[要引用]。 またこれを利用して、リモートリポジトリへのコミット時に自動的にテストを実行でき、常に動作可能な状態に保つ継続的インテグレーション~~（CI）~~ツールも存在している[要引用]。 他にも、テストコードを先に記述してからプロダクトコードの開発を始めるテスト駆動開発（TDD）や振る舞い駆動開発（BDD）と呼ばれるプラクティスも提案されている[要引用]。

> [3] “JUnit5”. https://junit.org/junit5/

手動で

一方で、テストコードの記述は基本的には人手によって行われている[要引用]。テストコードの記述作業は時間的なコストが高く、メンテナンスコストや学習コストの高さとも相まって自動テストを導入する大きな障害となっている[要引用]。また、テストコード記述の作業コストの高さは開発者の過労働やテスト漏れによるリリース後の不具合発生を誘発させている[要引用]。テストコードの自動生成技術はこれらの問題を解決しうる。

複数行われている

テストコードの自動生成を目的とした研究がいくつか断片的に成されている。 EvoSuiteはいくつかの選択可能な基準で最適化された関数単位でのテストコードを自動的に生成することができる[a]。しかし、ソー

開発者の労力を必要としないとは言い難い。

スコードの質が低いことが指摘されている[b]。また、関数単位でのごく簡単なテストシナリオにしか対応していない。 BDDにおける製品の振る舞いを中間言語で記述したファイルや、JavaDocなどの構造化された自然言語で書かれた仕様書からテストコードを自動生成する手法がいくつか提案されている[c-f]。これらのアプローチにはテストの原型となる文書が必要であり、汎用的であるとは言い難い。 深層学習によってソースコードを自動生成するプログラム合成のような技術をテストコードに適用した研究は現時点では確認できない。またプログラム合成は研究用に定義されたドメイン固有言語を対象にしているものが大多数であったり、コード生成の精度が十分でないなど、実用的なテストコード生成への見通しが悪い[要引用]。

多く

であるとは言い難い

> [a] Gordon Fraser, Andrea Arcuri, “EvoSuite: Automatic Test Suite Generation for Object-Oriented Software”, 19th ACM SIGSOFT Symposium
> [b] Sina Shamshiri, José Miguel Rojas, Juan Pablo Galeotti, Neil Walkinshaw and Gordon Fraser, “How Do Automatically Generated Unit Tests Influence Software Maintenance?” , 2018 IEEE 11th ICST
> [c] Manish Motwani and Yuriy Brun, Automatically Generating Precise Oracles from Structured Natural Language Specifications, ICSE 2019
> [d] Tradacu
> [e] Jdoctor

テストコードの再利用においても、その効果が期待される

既存のプロジェクトに存在するテストケースを再利用するアプローチは、上に挙げた問題を解決しうる。既存のコードを再利用することで開発効率や信頼性が向上することが一般に知られており[要引用]、テストコードを再利用することは実用性の上でも効果的であると考えられる。SondhiらはJavaおよびPythonのオープンソースライブラリのテストコードの再利用性について調査し、類似した機能へのテストコードが存在することや、テストコードを移植することで同様のテストが行えることを示した[要引用]。ただし、あるライブラリのテストコードを別のライブラリに適用する際には手動で書き換えを行っている。このことから、テストコードの移植可能性には未だ制限がある。

既存の

手法

> [] Devika Sondhi, Diva Rani, Rahul Purandare, Similarities Across Libraries: Making a Case for Leveraging Test Suite, ICST 2019

既存のプロジェクトのテストコードを別のプロジェクトに移植する際の課題について、我々は以下のように考える。 まず、移植したテストコードが移植先のプロジェクトの機能を正しくテストできるか？ この質問はテストコードの移植の成功を前提としているため、現時点では答えられない。 ただ、移植したテストコードが正しくテストを行うためには、そのテストコードが移植した先でも実行可能な状態にする必要がある。 では、移植したテストコードを移植先のプロジェクトに対して適切に動作するよう改変できるか？ テストコードの実装はテスト対象の実装に依存するため、同様の機能を提供していてもその実装が異なる場合、テスト内容を保ったまま適切に複雑な改変を施すのは困難であると考える。 であるならば、再利用のために移植するテストコードの持つべき条件とは何か？ それは機械的な処理を施すだけで、移植先のプロジェクトにおいても実行可能となるような実装を持つことであると考える。

本研究では、再利用ベースのテストコード自動生成技術の提案に向けた、既存プロジェクトの持つテストコードの移植可能性について調査する。 調査対象にはGitHub上に存在するテストコードを持つJavaオープンソースシステム（以下、OSS）プロジェクトを用いた。 Javaは実装言語としても研究対象言語としても

広く使われており、GitHubに多くのJavaプロジェクトが登録されている点や、テストフレームワークとしてJUnitが支配的に使用されている点を考慮して選択された。
広く
また本研究ではあるテストコードを別のプロジェクトに移植できる条件を提案する。 これはテストメソッド中で呼び出されるメソッドの呼び出し先を確認し、それと同質のものが移植先のプロジェクトに存在するか確認することをベースにしている。移植可能性を満たすテストメソッドは、そこで呼び出される補助メソッドとともに、識別子の変更などを行ったのち一つのテストコードとして移植が可能となる。

本研究の目的は、我々の提案する移植可能条件を満たすテストコードがどの程度存在し、移植可能なテストコードが実際に移植先でどの程度テストを行えるかを明らかにすることである。また加えて、プロジェクトの属するドメインや用意する移植元プロジェクトの数が結果に与える影響を調べる。この目的達成のため、様々な基準で選定した1900個のJavaOSSリポジトリを19個のドメインに分類し、それらを元に以下の研究設問に答える。
要検討

- RQ1 テストコードの移植元として使用できるプロジェクトはどの程度存在するか。
- RQ2 テストメソッドにはテスト対象~~であるプロダクトコードに~~依存する呼び出しがどの程度存在するか。
- RQ3 テストメソッドの移植に伴って移植が必要となるテスト補助メソッドはどの程度存在するか。
- RQ4 移植可能性を満たすテストメソッドはどの程度存在するか。またプロジェクトのドメインによって結果は変化するか。
要検討
- RQ5 移植されたテストコードはどの程度のカバレッジを持つか。
- RQ6 移植されたテストコードは実際に実行できるか。
- RQ7 移植されたテストコードはテスト対象の機能を正しくテストするか。
- RQ8 移植元として用意するリポジトリの数を変えた時、結果はどのように変化するか。
どのくらいの数が最も効率がよいか。

調査の結果、（調査結果）が示された。

本論文の以降の構成を示す。 二章では調査対象であるテストコードの構造について詳述する。 三章では提案するテストコードの移植可能条件について述べる。 四章ではそれぞれの研究設問について実際に調査し、調査結果を述べる。 五章では妥当性への脅威について述べる。 六章では関連研究を紹介し、七章で結言と今後の課題を述べる。

テストコードの構造

JUnitを例に挙げて説明する。

単純な単体テストの例

netty/codec/src/test/java/io/netty/handler/codec/LengthFieldPrependerTest.java

```
public class LengthFieldPrependerTest {

    private ByteBuf msg;

    @Before
    public void setUp() throws Exception {
        msg = copiedBuffer("A", CharsetUtil.ISO_8859_1);
    }

    @Test
    public void testPrependLength() throws Exception {
        final EmbeddedChannel ch = new EmbeddedChannel(new LengthFieldPrepender(4));
        ch.writeOutbound(msg);
        ByteBuf buf = ch.readOutbound();
        assertEquals(4, buf.readableBytes());
        assertEquals(msg.readableBytes(), buf.readInt());
        buf.release();

        buf = ch.readOutbound();
        assertSame(buf, msg);
        buf.release();
    }
```

複数のテスト対象メソッドとテスト補助メソッドを含む例

テストメソッド移植可能条件

の機械的な移植を可能にする条件

要検討

本章では、我々の提案するテストメソッド移植可能条件について説明する。 前提として、テストメソッドが移植可能であるとは、あるプロジェクトに属するテストメソッドが、それが依存する他のテスト補助メソッドと共に、メソッド名やクラス名などの識別子のみを変更した状態で別のプロジェクトに移動した場合に、コンパイルエラーを引き起こすことなく、その成否を問わずテストが正常に実行できることを指す。 テストメソッドの移植可能性はそれのみでは判断できず、ある移植元プロジェクトからある移植先プロジェクトに移植を行う上での移植可能性として判断される。

テストコードはプロダクトコードとは独立したものとして開発される。そのプロジェクトの提供したい機能はプロダクトコードによって実現され、テストコード自体はプロジェクトの機能実現に与しない。そのため、テストコードとプロダクトコードは適切に分離されており、プロダクトコードの一切の機能はテストコードに依存しない。このことから、テストコードからプロダクトコードのメソッドが呼び出されることはあるが、プロダクトコードからテストコードのメソッドが通常呼び出されることはない。従って、テストコードとプロダクトコード間の依存関係を調べるには、テストコード中に出現するプロダクトコードのメソッド呼び出しに注目すればよい。ここで、メソッド呼び出しにはクラスのインスタンス化を含む。

また、テストメソッド中のメソッドの呼び出しは以下の三種類に分類できる。

- A: Privateなメソッドなど、そのテストメソッドと同じクラス内に存在するメソッド、あるいは、別の

テスト用のクラスに存在する補助メソッド。
- B: 外部からインポートしたライブラリに含まれるクラスのメソッド。
- C: テスト対象のプロダクトコードに含まれるクラスのメソッド。

このうち、Aはテストメソッドの移植に伴って同様に移植することでその依存関係を保つことができるため、移植可能条件には関与しない。またBも同様に、移植先で同じようにインポートすることで依存関係を保つことができるため、移植可能条件に関与しない。従って、テストメソッドの移植可能条件を考える際にはテストコード中に含まれるメソッド呼び出しのうち、Cのみを考えればよいことがわかる。

まず、最も厳格な移植可能条件は、移植元のプロジェクトにおいてテストメソッド中で呼び出される全てのCのメソッドおよびそれを所有するクラスがが移植先のプロジェクトにも存在していることである。

例えば、〜〜。

しかし、クラス名、およびメソッド名は、開発者が好きに命名できるものであり、それらがたまたま全て一致すると考えるのは現実的ではない。

前提の通り、テストメソッド中のテスト対象に依存するクラス名およびメソッド名はテスト対象である移植先リポジトリに存在するクラス名およびメソッド名に改変することが可能であると考えた場合、それが実行可能になる条件を考える。

条件1 テストメソッド中で呼ばれるプロダクトコードのメソッドと同じインターフェースを持つメソッドが全て移植先のプロジェクトに存在する

ここでインターフェースとは、メソッドが要求する引数の個数およびそれぞれの型、またメソッドの返す値の型を指す。

例えば、〜〜。 移植するやつ

また引数あるいはメソッド自体の型をTとすると、Tの種類として次のパターンが考えられる。

- a: int, bool, longなど、Primitive型
- b: String, File, HttpSearverなど、外部からインポートしたライブラリで定義されたクラス型
- c: 移植するテストコードで定義されたクラス型
- d: 移植元のプロジェクトのプロダクトコードで定義されたクラス型

このうち、Tがaまたはcである場合、移植先でテストメソッドを移植することへの制限は特にない。 一方、Tがbであるメソッドmbを含むテストメソッドを移植しても実行が可能であるためには、移植先にmbと同じインターフェースを持つメソッドmb'を定義しているプロダクトコードが、bをインポートしている必要がある。 また、Tがdであるメソッドmdを含むテストメソッドを移植しても実行が可能であるためには、移植先にmdと同じインターフェースを持つメソッドmd'を定義しているプロダクトコードが移植先プロジェクトに存在する必要がある。

特にTがdである場合、移植元のプロダクトコードで定義されているクラスが移植先のプロダクトコードでも定義されていると考えるのは現実的ではない。 このことから、移植するテストメソッドに含まれる移植

先プロダクトコードのメソッド `m` の呼び出しを移植先プロジェクトのプロダクトコードに含まれるメソッド `m'` に置き換える場合、次の条件を満たす必要がある。

条件2 `m` と `m'` が型として扱うテスト対象のプロダクトコード内で定義されたクラスは、そのテストメソッドが使用する範囲において、同じ役割を持つ。

例えば、〜〜。

さらに、テストメソッド中で呼び出されるメソッドで型として扱われるプロダクトコードで定義されたクラス間において、次の条件を満たす必要がある。

C c = A.hoge(B b)

条件3