情報処理学会論文誌 Vol.62 No.4 1019–1028 (Apr. 2021)

Javaテストコードの再利用による自動生成に向けた移植可能なテストメソッドの調査

西浦 生成[1,2,a),b)] 水野 修[1,c)] 崔 恩瀞[1,d)]

受付日 2020年8月3日, 採録日 2021年1月12日

概要：ソフトウェアの品質を高く保つため，ソフトウェアテストがさかんに行われている．テストコードを用いた自動テストは，テスト実行の労力を軽減するが，テストコードの作成には労力が必要となる．既存のテストコード自動生成技術は，入力として何らかの文書を必要とするか，でなければ非常に単純な単体テストのみしか作成できない．一方，既存のテストコードを再利用することで，使用されていた実績があり人手で作られたものに近い高品質なテストコードを手軽に生成できる可能性がある．我々は新たなJavaテストコード自動生成技術の開発に向け，その第1歩として，あるテストメソッドを実行可能性を保ちつつ自動的に移植する手法およびそれが可能となる条件を構想した．さらにオープンソースのJavaプロジェクトを調査し，テストコードの再利用元として使用できるリポジトリが1,862件存在することや，大半のテストメソッドはテスト対象コードにたかだか2つの依存を持つこと，そうした依存関係を保つ移植によって本来必要なテストメソッドの平均83%を生成できる可能性を実証的に示し，既存テストコードの再利用による自動生成手法が現実的かつ有益であることを示した．

キーワード：ソフトウェアテスト，リポジトリマイニング，ソースコード解析

An Investigation of Transplantable Test Methods toward Automatic Java Test Codes Generation by Reusing Existing Test Codes

Kinari Nishiura[1,2,a),b)] Osamu Mizuno[1,c)] Eunjong Choi[1,d)]

Received: August 3, 2020, Accepted: January 12, 2021

Abstract: Software developers test their software products. Automated testing with test codes lightens their efforts to test. However, writing appropriate test codes manually is difficult and a time-consuming activity. Several existing automatically create test codes techniques require to prepare some documents, otherwise, they can create only very simple unit tests. Reusing existing test code may easily generate high-quality test code. Toward the technique, we propose the transplantable condition for that a test method can keep its feasibility after transplanting. We investigate open source Java projects in GitHub. As the results, we have shown that there are available 1,862 repositories with test codes, and that most test methods have at most 2 dependencies on the code under test. We have empirically demonstrated the possibility that porting that maintains such dependencies can generate an average of 83% of the originally needed test methods. From the results, an automatic generation method by reusing existing test code is realistic and useful.

Keywords: software testing, repositories mining, source code analysis

[1] 京都工芸繊維大学
Kyoto Institute of Technology, Kyoto 606–8585, Japan
[2] 産業技術総合研究所
National Institute of Advanced Industrial Science and Technology, Ikeda, Osaka 563–8577, Japan
[a)] k-nishiura@se.is.kit.ac.jp
[b)] k-nishiura@aist.go.jp
[c)] o-mizuno@kit.ac.jp
[d)] echoi@kit.ac.jp

1. はじめに

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．そのため，リリース前にソフトウェアの不具合を検出することを目的としたソフトウェアテストが広く行われている．かつては手動による入力および出力の目視確認に

よってテストが行われていたが，この労力を軽減すべく，自動的にテストを行うためのプログラムであ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，動作が正しく行われているかを確認できる．テストコードを実行することで，テストを素早く繰り返し行うことができ，ソフトウェアテストの実行にともなうコストを軽減できる．

一方で，テストコードは基本的に手動で記述されるため，作成には相応の作業コストが要求される．この作業コストは自動テスト導入への障害となるほか，開発者の過労働やテストの作成不足による不具合発生を誘発する可能性がある．この問題に対し，テストコードの自動生成技術による緩和が期待されている．しかし，既存のテストコード自動生成技術 [2], [3], [4] は入力として何らかのドキュメントを必要とするか，でなければ非常に単純な単体テストのみしか作成できないという問題点を持つ．

そこで，我々は新たに，既存のテストコードを再利用することで Java テストコードの自動生成を行う手法を構想する．既存のテストコードを再利用することで，過去に人手によって作られ使用されていた実績のある，高品質なテストコードを手軽に生成できる可能性がある．本研究ではまず，ある既存のテストメソッドを別のプロジェクトに 1 つのテストコードとして移植する方法を提案する．この方法では，テストメソッドおよびそれに関連する部分コードを抽出し，テンプレートを埋める形で移植用のテストコードを新しく作成した後，プロジェクト固有のプロダクトコードにアクセスする識別子を移植先のプロダクトコードに由来する識別子に置換することで，対象のテストメソッドが持つすべての依存を保持し実行可能性を保つ．さらに，こうした識別子の置換が可能となる条件を提示することで，テストメソッドが移植可能となる条件を示す．

本研究ではさらに，我々が目標とする既存テストコードの再利用によるテストコード自動生成手法の実現可能性を探るため，再利用の素材となる既存テストコードを調査する．調査対象には GitHub 上に存在するテストコードを持つ Java オープンソースソフトウェア（以下，OSS）プロジェクトを用いた．これらの有用性および手法の実現可能性を調査するために，以下に示す 3 つの研究設問（Research Question，RQ）を設定する．

RQ1　テストコードの移植元として利用できる OSS はどの程度存在するか？

動機：再利用の素材として，テストコードを含む既存の OSS が多く必要となる．そうした OSS がどれほど存在するかを把握することで，テストコード再利用の現実性を示す．

結果：JUnit または TestNG を使用するテストコードを含み，一定の品質基準をクリアした GitHub リポジトリは全体で 1,862 件存在した．これらは合計約 40 万ファイルのテストコードを含む．またドメインで分類すると，多いもので約 300 件のリポジトリが同ドメインに属している．

RQ2　テストメソッドは他の参照先にどれほど依存しているか？

動機：テストメソッドが他の参照先，特にプロダクトコードで定義された参照先にどれほど依存しているかによって，その移植に要する条件は厳しくなる．こうした依存の大きさの傾向を把握することで，移植可能であるために要求される条件の傾向を示す．

結果：テストメソッドは，その実行にともなってテストのための補助的な呼び出しを平均 1.8 種類実行するため，それらをともに移植する必要がある．また平均 4.0 種類の呼び出しを通してプロダクトコードに依存しており，移植可能であるためにはそうした依存を移植先でも保つ必要がある．

RQ3　既存のテストコードには，テスト対象への依存関係が生成対象であるテストメソッドと共通するテストメソッドはどれほど存在するか？

動機：特定のリポジトリをテストコードの移植先および移植元とした場合に，移植先において本来必要となるテストメソッドが，移植元における移植可能なテストメソッドにどれだけ含まれるかを把握することで，既存テストコードの移植によるテストコード生成手法が達成できる効果の上限を示す．

結果：99 件の同ドメインの OSS からの移植を仮定した場合，移植先リポジトリが本来必要とするテストメソッドのうち平均 83%は，事前に定義した移植条件を満たすものが移植元に 1 つ以上存在するため，移植することで同様のテストメソッドを生成できる可能性がある．一方，残りの平均 17%では，移植による生成がそもそも不可能である．

本論文の以降の構成を示す．2 章ではテストコードの自動生成に関連する先行研究について説明する．3 章ではテストメソッド単位での移植によるテストコード自動生成手法の構想を述べる．4 章ではそれぞれの研究設問について調査し，調査結果を述べる．5 章ではそれをもとに再利用によるテストコード自動生成技術の実現可能性について考察する．6 章では妥当性への脅威について述べる．最後に，7 章で結言と今後の課題を述べる．

2. 関連研究

テストコードの自動生成を目的とした研究がいくつか行われている．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 Java テストコードを自動的に生成することができる [2], [5]．しかし，テストコードの質が

低いことが指摘されている [6]．また，ごく簡単な単体テストにしか対応していない．関数単位での回帰テスト用のコードを自動生成する Randoop [7] も同様の問題をかかえている．ほかにも，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 [3], [8], [9] や，振舞い駆動開発における製品の振舞いを独自の記法で記述したファイルからテストコードを自動生成する手法 [4], [10], [11] がいくつか提案されている．しかし，これらの手法はテストコードの原型となる文書の準備を要するため，純粋な自動生成とはいい難い．また，深層学習などによるプログラム合成手法は，研究用に定義されたドメイン固有言語を対象としているものが多く，あるいは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には至っていない [12]．このように，結合テストなどの複雑なテストコードを高精度かつ手軽に自動生成する手法はいまだ実現されていない．

一方，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知られている [13]．したがって，既存のテストコードを再利用するアプローチにも一定の効果が期待できる．テストコードではないが実際に，既存のソースコード片を再利用することでコード生成を実現するいくつかの方法が提案されている [14], [15]．また Sondhi ら [16] は，数種類の Java および Python の OSS 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可能性を示した．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移植する際には，内容の書き換えを手動により行っている．このように，テストコードを自動的に移植する手段についてはいまだ開拓されていない．

3. テストコード自動移植方法の構想

本章では，ある Java プロジェクトに存在するテストメソッドを別のプロジェクトに実行可能なテストコードとして自動的に移植する方法，および，そうした移植が可能となる条件について構想する．

3.1 移植の方針

移植方法の方針を以下にまとめる．

- テストの移植は，テスト機能の最小単位であるテストメソッドごとに行うこととし，1 つのテストメソッドにつきそれを実行するための独立した 1 つのテストコードを作成する．
- **図 1** に示すテストコードテンプレート（以下，T）を用いる．T 中の点線で囲まれた各部分を移植元プロジェクトから抽出したコードで埋め，また特定の変更を施すことで，移植先で動作する新たなテストコードを作成する．

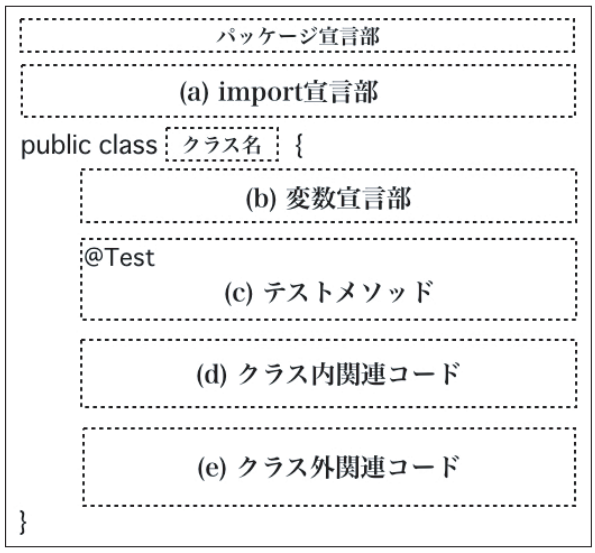

図 1 テストコードテンプレート
Fig. 1 Template of test code.

- 移植を試みるテストメソッドは，移植元プロジェクトに存在する (A) 関連テストコード，(B) テスト対象であるプロダクトコード，およびインポートして使用する (C) サードパーティ製のライブラリにそれぞれ依存を持ちうる．こうした依存を移植前と移植後で適切に保つことで，移植したテストメソッドの実行可能性が保たれる．
  - 上記のうち (A) をすべて抽出し，1 つのテストコード T にパッケージングする．
  - 上記のうち (B) に関する T 内の識別子をすべて，移植先プロジェクトのプロダクトコードにおける適切な識別子に置換する．
  - 上記のうち (C) に関するすべてのインポート宣言文を T に記述し，移植先プロジェクトでもそれらが利用できるよう整備する．

3.2 前提

テストメソッドの移植は，以下を前提とする．

- 移植を試みるテストメソッドは，移植元プロジェクトにおいて正しく動作している．
- 移植元プロジェクトと移植先プロジェクトの両方において，テストコードに依存するプロダクトコードは存在しない．
- 必要な設定を行うことで，移植元プロジェクトにおいてテストメソッドの実行のためにインポートされているサードパーティ製のライブラリを，移植先プロジェクトでも同様にインポートし使用することがつねに可能である．
- 移植を試みるテストメソッドを定義するクラスは，継承（extends）や実装（implements）を使用していない一般的なクラスである．

- 移植を試みるテストメソッドおよびそれに関連するテストコードは，テスト対象となるプロダクトコードに対して，メンバメソッド呼び出し，メンバ変数へのアクセス，インスタンス生成，データ参照型宣言の参照先，のいずれかの形でのみ依存を持ちうる．

3 番目の前提については，たとえば Eclipse や Maven などのビルドツールを使用して依存パッケージ管理を行ったとしても，現実的には必要な外部ライブラリの使用に何らかの理由で失敗する可能性が考えられるが，議論の簡素化のため，本論文ではつねに利用できるとするものである．また下 2 件の前提も同様に本研究における議論を簡素化するための制限であり，今後の発展としてこれらの前提外の対象への拡張を制限するものではない．

3.3 移植手順

移植を行う際の具体的な手順を説明する．移植元プロジェクトから移植を試みるテストメソッドを M とする．また，M を定義しているクラスを C とし，M および C が記述されているソースコードファイルを F とする．

(1) T 中の (c) に M を移植する．
(2) M が依存する C 内の要素をすべて T 中の (b) または (d) に移植する．ここには，メソッド，内部クラス，変数宣言文が含まれる．またここには，M 内で直接使用されるもののほか，何らかの要素を通じて間接的に使用するものも含まれる．たとえば，M が直接呼び出すプライベートメソッドがさらに呼び出す他のプライベートメソッドなどが該当する．また加えて，JUnit における Before メソッドなど，M の実行に際してフレームワーク固有の暗黙的な依存を持つ要素も含まれる．
(3) M が依存する，F 内かつ C 以外のクラス，および，F 以外のテストコードで定義されたクラスをすべて T における public クラスの内部クラスとして移植する．この依存は手順 (2) の場合と同様に，直接と間接を問わない．またこのとき，移植する各クラスはそのすべてのメンバ，メソッド，および内部クラスを含める必要はなく，M が依存するものだけでよい．
(4) F，および，手順 (3) で移植したクラスを本来定義していたテストコードから，T が依存するサードパーティ製のライブラリの import 宣言文を T 中の (a) にそれぞれ移植する．
(5) T には移植前プロジェクトにおけるテスト対象であるプロダクトコードへの依存がいまだ存在しているため，現段階の T を移植先プロジェクトに配置しても実行することはできない．したがって，T 中の識別子のうち，移植前プロジェクトにおけるプロダクトコードに由来するものをすべて，実行可能性を保ちつつ，移植先プロジェクトにおける識別子に置き換える必要がある．このように置き換え可能な識別子の組が移植先プロジェクトに存在している場合に限り，M は移植可能となる．
(6) 手順 (5) で置換した移植先プロジェクトの識別子に基づいて，必要な import 文を T 中の (a) に追加する．
(7) 最後に，他に移植するテストコードと重複しない public クラス名を設定し，それに応じたファイル名を設定する．また配置する場所に応じたパッケージ名を設定する．

以上の操作によって，テストメソッド M を対象に移植したテストコードが得られる．このテストコードは，3.1 節で述べた (A) から (C) の 3 種類の依存を保っているため，3.2 節の前提のうえで，移植先プロジェクトにおいても実行可能であることが期待される．

3.4 移植可能条件

移植を試みるテストメソッドが移植先で実行可能性を保つことができる場合，移植が可能であると考える．ここまでの議論から，テストメソッドが実行可能性を保つためには，3.3 節の手順 (5) における，プロダクトコードへの依存を保つ適切な識別子の置換が行える必要がある．したがって，置換可能な識別子を提供するプロダクトコード中の参照先が移植先プロジェクトにすべて存在していることが移植可能条件となる．

テストコード側から見ると，プロダクトコードの各参照先はブラックボックスとして振る舞う．したがって，テストコードがアクセスするプロダクトコード内の各参照先に対する入出力の関係が移植元と移植先で保たれてさえいれば，動作上の問題はない．

ここで，以下の場合に入出力が保たれると考える．メンバ変数へのアクセスの場合，そのメンバ変数のデータ型が一致する．メンバメソッドへのアクセスの場合，メソッドにおける引数のデータ型とその並び，および返り値のデータ型が一致する．インスタンス生成の場合，コンストラクタメソッドにおける引数のデータ型とその並び，および作成されるインスタンスのデータ型が一致する．データ型の一致について，プリミティブ型や，標準ライブラリおよびサードパーティ製のライブラリのクラスを参照する参照型は，完全一致する場合に限り一致と見なす．一方，移植元プロジェクトで定義された固有クラスへの参照型は，移植先プロジェクトで定義される任意の固有クラスへの参照型に一致すると見なす．ただし，こうした固有クラス間の依存関係が移植元と移植先で保たれる必要がある．

固有クラス間の依存関係について，具体例を用いて詳述する．あるテストメソッドを中心として関連参照先を抽出し作成した移植用テストコード（3.3 節の (4) 終了時に相当）を考える．**表 1** は，このテストコードに存在する移植元プロジェクトのプロダクトコードへの依存を表してい

表 1 移植用テストコードにおけるプロダクトコードへの呼び出し例
Table 1 Example of call for product codes in test code.

クラス名	メソッド名	引数のデータ型	返り値のデータ型
ClassA	convert	[long]	int
ClassA	setBtoA	[ClassB]	void
ClassB	getValue	[ ]	int

る．すなわち，2 種類のクラスに属する 3 種類のメソッド呼び出しを通じて依存している．また下線は，移植用テストコード中に存在する置換すべき識別子であることを表している．ここで，ClassB は引数を持たず int 型を返すメソッド getB を有している．したがって，移植先にも同様の入出力を持つメソッドを有するクラスが存在する必要がある．加えて，ClassB は ClassA の持つメソッド setBtoA の引数に含まれる参照型に参照されている．したがって実際に移植先プロジェクトに要求されるのは，こうした ClassA と ClassB 間の関係と同じ関係を持つ 2 クラスの組合せとなる．

3.5 利用者とのインタラクション

本技術は利用者が開発するプロジェクトの情報をもとに，適用可能なすべてのテストメソッドを既存プロジェクト群から検索し自動的にテストコードを生成することを目的とする．そのため必ずしも利用者が事前にテスト仕様を決定する必要はない．ただし，あらかじめテスト仕様をイメージしておくことで生成されたうち不必要なテストコードを除外することができる．また，場合によっては膨大な数のテストコードが生成されることも考えられる．この場合，利用者は候補として提示されたうち適切なテストコードを選択して使用することになる．将来的には生成数の上限を定めたり，よりテストしたい箇所をあらかじめ選択して重み付けしたりすることも考えたい．また，ただ実行可能であるだけで意味的には適切でないテストが生成される場合もある．現状の提案内容では利用者の目視判断によって選択する必要があるが，将来的には 5.2 節で述べるような改善が考えられる．

4. 調査

前章で述べた移植手法の開発に向け，その実現可能性を評価するため，以下に述べる 3 つの調査を行った．

4.1 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？

4.1.1 動機

テストコードからの移植を行う素材として，テストコードを含む既存のプロジェクトが多く必要となる．そうした Java プロジェクトがどれほど存在し利用できるかを把握することで，テストコードを再利用する手法がどれほど現実的かを調べる．またテストコードの移植において，共通のドメインに属するプロジェクトからの移植が有効であると考えられるため，得られたリポジトリのドメイン分類を行う．

4.1.2 調査対象

GitHub [17] に存在するすべてのリポジトリを対象として，リポジトリ情報を保管するサービスである GHTorrent [18]，および GitHub が提供する GitHub API (v3) [19] を用いて絞り込む．また本実験で扱うテストコードは，テストフレームワークとして JUnit4，JUnit5，TestNG を使用したものに限定する．これらのフレームワークでは，メソッドに@Test アノテーションを付することがテストメソッドの識別ルールとなる[*1]．GHTorrent のダンプデータは実験開始時に最新であった 2019 年 6 月登録のものを使用した．API による情報取得は 2019 年 9 月 2 日，リポジトリのダウンロードは 2019 年 11 月 15 日に実施した．

4.1.3 手法

調査 1-A から**調査 1-C** を順に行う．

調査 1-A：GHTorrent を用いて，GitHub に登録されているすべてのパブリックなリポジトリのうち，以下を満たすものを抽出する：(1) 主な言語が Java，(2) 削除済みでない，(3) フォークによって作られたものでない，(4) スターを 100 以上持つ．(3) と (4) の条件は，低品質なプロジェクトを除外する目的で設定した．次に GitHub API を用いて，これらのリポジトリの master ブランチにおける Tree オブジェクトファイル[*2]を参照することで，「test ディレクトリ下に存在する拡張子が．java のファイル」の数を取得する．最後に，このファイル数が 20 以上であるリポジトリをダウンロードし，ブランチが master でないものは master に切り替える．

調査 1-B：得られたリポジトリ内のファイルを静的に構文解析することで実際のテストコードを特定し，以下の条件のいずれかを満たすリポジトリを除外する．

- プロダクトコードが存在しない．ここでプロダクトコードとは，テストコードではない*.java ファイルすべてを指す．
- テストメソッドが存在しない．ここでテストメソッドとは，テストコード中に存在する@Test アノテーションが付与されたメソッドを指す．また，@Ignore アノテーションの有無は関与しない．
- org.junit または org.testng を名称に含むライブラリを 1 度もインポートしていない．
- 構文解析を行ううえで，デコードおよびパースが失敗し解析できなかった Java ファイルの割合が全体の 5%を上回る．あるいはこうした Java ファイルが 20

[*1] JUnit5 ではより細かいルールも存在するが，本研究では簡単のため主たるものに統一した．
[*2] リポジトリ内のディレクトリ構造が記されたファイル

表 2 絞り込みで得られた 1,862 リポジトリの情報
Table 2 Information of 1,862 extracted repositories.

(単位：個)	最小値	平均値	中央値	最大値
スター数	100	1,121.6	329	81,817
全 Java ファイル	22	898.3	338	57,851
テストコード	10	216.3	63	4,508
プロダクトコード	1	676.7	233	57,372

以上存在する．

- テストメソッドを含む Java ソースファイル数が 10 以下である．

上記の条件のうち最後の条件は，テストコードが極端に少ないリポジトリは移植元として明らかに不適なため，以降の調査から除外することを目的に設定した．静的解析には JavaParser の Python ラッパである javalang パッケージを使用した．

調査 1-C：得られたリポジトリのそれぞれに対して著者がドメインのラベル付けを手動で行う．ラベルには，GitHub 上の優れた Java プロジェクトを紹介する Awesome Java[*3]で用いられている 66 種類のドメインを参考に，著者の判断で 21 種類のドメイン（表 3）を使用する．各リポジトリの GitHub ページの description 文および README.md を目視で確認することで，それが属すると思われるドメインラベルを 3 つまで選択する．ドメインにつながる情報が得られないものや判断がつかないものは No Category とする．

4.1.4 結果

調査 1-A：全工程完了時，2,124 件のリポジトリが得られた．

調査 1-B：全工程完了時，1,862 件のリポジトリが得られた．これらのリポジトリの情報を**表 2** に示す．リポジトリに含まれるテストコードの合計は 402,751 ファイルとなった．

調査 1-C：ドメイン分類の結果を**表 3** に示す．各ドメインごとにそのラベルが貼られたリポジトリ数を示しているため，数字の合計が総リポジトリ数と異なっている．ソフトウェア開発に用いられることを示す Development ドメインに属するリポジトリが最も多く，次いで，データの処理や規格を扱うことを示す Data ドメインや，Web アプリケーションであることを示す Web ドメインが多かった．したがって，これらのドメインに属するソフトウェアには，既存 OSS のテストコードを再利用しやすいと期待できる．

4.2 RQ2：テストメソッドは他の参照先にどれほど依存しているか？

4.2.1 動機

テストメソッドが他の参照先，特にプロダクトコードで定義された参照先にどれほど依存しているかによって，その移植に要する条件は厳しくなる．こうした依存の大きさの傾向によって，移植可能であるために要求される条件の全体的な傾向を示す．

[*3] https://github.com/akullpp/awesome-java

表 3 ドメイン分類結果
Table 3 The classification result of the domain of repositories.

ドメイン	リポジトリ数	ドメイン	リポジトリ数
Development	339	Multimedia	100
Data	304	Testing	92
Web	256	Security	71
Network	201	Document	65
Distribution	159	Performance	46
Database	157	OS	44
Programing	137	Financial	41
Operation	135	Geospatial	31
Android	119	Middleware	30
Utility	104	Game	29
Science	101	No Category	54

4.2.2 調査対象

RQ1 の調査で得られた 1,862 件のリポジトリのうち，上位 6 つのドメインに属するリポジトリを，それぞれスター数の多い順に 100 件選択し，調査対象とする．これは，調査の規模と労力を抑えるため，再利用される可能性が高いリポジトリを代表させるという意図による．このとき，1 度選択された中でファイル数が突出して多い 3 件のリポジトリ[*4]は，調査結果の妥当性を損ねると判断し，これらを除外して再選択した．ドメイン間にリポジトリの重複があるため，選択されたリポジトリは合計で 509 件となった．

これらのリポジトリに含まれるテストコードからテストメソッドを抽出する際，3.2 節に示した前提のうち下 2 件の前提を満たさないものは調査範囲から除外した．その結果，合計 348,045 個のテストメソッドが調査対象として得られた．

4.2.3 手法

静的な構文解析によって，調査対象のテストメソッドが依存するテストコード内の参照先をすべて特定し，集計する．その後，同じく構文解析を行うことで，それらの持つプロダクトコードの参照先，あるいは，プロジェクト固有ではないライブラリへの依存をすべて特定し，集計する．

4.2.4 結果

テストメソッドの依存先を集計し分類した結果を**表 4** に示す．平均として，1 つのテストメソッドは合計 13.9 個の参照先への依存を有している．そのうち，テストコード中の参照先に対しては平均 1.8 個の依存を有している．これは，これは，あるテストメソッドを移植する際に，自身のほかに平均 1.8 個の参照先を同時に移植させる必要がある

[*4] aliyun/aliyun-openapi-jaba-sdk，JetBrains/MPS，aws/aws-sdk-java

表 4 各テストメソッドが依存する参照先の集計および定義場所の分類結果

Table 4 Dependencies and references of each test method.

参照先	平均値（個）	平均値の割合（%）	中央値（個）
すべて	13.9	100.0	9
テストコード	1.8	12.9	1
プロダクトコード	4.0	28.8	2
外部ライブラリ	8.1	58.3	5

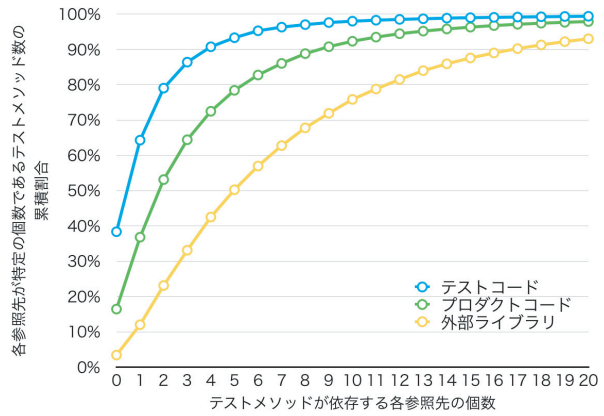

図 2 各定義場所への参照数が特定の個数であるテストメソッド数の累積割合グラフ

Fig. 2 Cumulative percentage graph of the number of test methods whose number of references to each definition location is a specific value.

ことを示している．また，プロダクトコード中の定義ファクタに対して平均 4.0 個の依存を有している．これは，テストメソッドの移植を行う際に移植元プロジェクトと移植先プロジェクトの間で平均 4.0 個のプロジェクト固有の識別子の対応づけを行う必要があることを示している．

より詳細な分析のため，それぞれの依存先への依存が各個数をとるテストメソッドの個数の累積割合グラフを**図 2** に示す．たとえば，緑のグラフで表されるプロダクトコード中の参照先への依存の個数 N が 1 のとき，テストメソッド数の累積割合は約 47%であることが読み取れる．これは，調査対象としたテストメソッドのうち約 47%が，プロダクトコード中の参照先への依存をたかだか 1 個しか持たないことを意味している．

この図から次の結果が観察できる．

テストコードへの依存について．約 40%のテストメソッドは他のテストコード中の参照先への依存を持たず，それ単体で機能することが分かる．他のテストコード中の参照先への依存が 4 つ以下のテストメソッドが全体の約 90%を占めており，大部分のテストメソッドは少量の補助コードをともなって移植用のテストコードを構成できる．

プロダクトコードへの依存について．プロダクトコード中の参照先への依存が 2 個以下であるテストメソッドが全体の約半数を占める．特に，約 20%のテストメソッドはプロダクトコードに対して単一の依存を持つ．これは EvoSuite などのツールで作成可能な単純な単体テストに相当する．また，約 17%のテストメソッドはプロダクトコードへの依存を持たない．これらのテストメソッドは標準ライブラリやサードパーティ製ライブラリで実現できる機能のテストのみを行っていると考えられる．

最後に，テストメソッドはプロジェクト固有ではないライブラリに対して平均 8.1 種類の呼び出しによる依存を有する．これらは，たとえば JUnit の提供するアサーション用メソッドや，String クラスなどの標準ライブラリが持つメソッドが多く該当した．依存量の平均的な割合は最も高い．移植先でもこれらのライブラリを使用できるよう設定できる前提のもとでは移植可能性に影響しないが，こうした依存量は設定作業にかかる労力の目安となる．

4.3 RQ3：既存テストコードには，テスト対象への依存関係が生成対象のテストメソッドと共通するテストメソッドはどれほど存在するか？

4.3.1 動機

移植元となるプロジェクトにおけるテストメソッドが持つプロダクトコードへの依存と，移植先のプロジェクトが本来必要とするテストメソッドが持つプロダクトコードへの依存が同一であるならば，テストメソッドを移植することで移植先で本来必要となるテストコードを生成できる可能性がある．特定のリポジトリを移植先および移植元とした場合に，そうしたテストメソッドがどれだけ存在するかを調べることで，移植によるテストコードが達成できる効果の上限を示す．

4.3.2 調査対象

RQ2 への調査で使用した 6 ドメインの各 100 件のリポジトリのうち，テストメソッド数が中央値をとるものを移植先リポジトリとして，また他の 99 件を移植元リポジトリとして用いる[*5]．各ドメインにおける移植先リポジトリのテストメソッド数，および移植元リポジトリの合計テストメソッド数を**表 5** に示す．

4.3.3 手法

テストメソッドの移植先として選ばれたリポジトリが保有しているテストメソッドを，そのプロジェクトが必要とするテストメソッドの正解データ（以下，正解テストメソッド）として用いる．構文解析によって，それらのテストメソッドが持つプロダクトコードへの依存を抽出する．次に，移植元として扱うリポジトリ群の各テストメソッドについても同様に依存を抽出し，正解テストメソッドのいずれかと依存が一致するものを集計する．この依存の一致判定は 3.4 節で示したように，依存する参照先の入出力の

[*5] 同ドメイン内で移植を行うことは移植条件において必須ではないが，少ないリポジトリから最大の効果が得られることが期待できる．

表 5 RQ3 での調査に用いる調査対象
Table 5 Target repositories of RQ3.

ドメイン	移植先リポジトリ	テストメソッド数	移植元の合計テストメソッド数
Development	FreeTymeKiyan/LeetCode-Sol-Res	202	95,919
Data	plutext/docx4j	321	99,869
Web	apache/nutch	242	62,258
Network	mitreid-connect/OpenID-Connect-Java-Spring-Server	251	63,519
Distribution	alibaba/Sentinel	319	70,435
Database	xetorthio/jedis	213	60,481

表 6 各ドメインの移植先リポジトリに存在する，プロダクトコードへの依存を 1 つ以上持つテストメソッド数（T_a），および，そのうち依存関係が一致するテストメソッドが移植元リポジトリに 1 つ以上存在するものの個数（T_b），またその割合（T_b/T_a）
Table 6 The number of test methods having one or more dependencies on product code in repositories of each domain (T_a), the number of test methods having one or more matching dependencies in the repository among them (T_b), and the percentage (T_b/T_a).

ドメイン	$\#T_a$	$\#T_b$	割合
Development	147	140	0.95
Data	317	285	0.90
Web	236	206	0.87
Network	240	154	0.64
Distribution	304	209	0.69
Database	208	191	0.92
平均	-	-	0.83

一致，およびそれらのクラス間が持つ関係に基づく．

4.3.4 結果

結果を**表 6** に示す．移植先における正解テストメソッドのうち，平均して 83%，最も高いものでは 95%の割合で，プロダクトコードへの同一の依存関係を持つものが，移植元とした 99 件のリポジトリに 1 つ以上存在している．したがって，プロダクトコードへの依存関係が同一であるこれらのテストメソッドは，適切に識別子を改変することで移植先プロジェクトでも実行可能であることから，移植先リポジトリが本来必要とするテストメソッドを作成することができると考えられる．一方，残りの平均 17%では，プロダクトコードへの依存関係が一致するテストメソッドが移植元に存在しないため，移植による生成がそもそも不可能である．

5. 議論

5.1 RQ への回答から得られる再利用によるテストコード自動生成の実現可能性への考察

RQ1 の結果から，1,862 件とけっして少なくない数の Java OSS リポジトリがテストコードの再利用素材として利用できる．また今回の調査では，スター数が 100 以上とやや厳しい基準を設けているが，より少ないスター数でも別のメトリクスを併用することで高品質なテストコードを含む他のリポジトリを選定できる余地が残されている．また，時間とともに再利用に適したリポジトリが増えていくことも考えられる．したがって，本調査で得られた数を下限として，実際にはより多数の再利用素材の確保が見込める．こうした事実は，再利用によるテストコード自動生成の実現可能性を高める一因となる．

RQ2 の結果から，約半数のテストメソッドはプロダクトコードにたかだか 2 種類の依存を有しており，これらを適切に保つような移植元と移植先の間の識別子の対応関係を探すことで，実行可能性を保ったまま移植が可能であるといえる．この結果は，テストコードの再利用の間口が広く設けられていることを示している．一方で，呼び出しが 9 種類以上とプロダクトコードへの複雑な依存を持つテストメソッドも約 1 割程度存在している．

RQ3 の結果から，移植先プロジェクトに本来必要となるテストメソッドについて，プロダクトコードへの依存の形が同一であるようなテストメソッドが既存プロジェクト群に高い割合で存在している．ただし，平均 17%，最も高いもので 36%のテストメソッドは，用意した同ドメインに属する 99 件の既存 OSS からは再利用によって生成されえないことが示された．こうしたテストメソッドを再利用によって生成するには，用意するプロジェクトを増やすか，あるいは識別子の変更のみに限定しない再利用方法を考える必要がある．

以上 3 つの RQ の結果から，再利用の対象となりやすい既存テストコードが多く存在しており，また目的のテストコードを得られる可能性も高い．したがって，既存テストコードの再利用による自動生成手法の構想は現実的かつ有益であると考えられる．

5.2 自動生成に向けて解決すべき課題

あるテストメソッド T が移植条件を満たすかを判定する場合，T の持つプロダクトコードへの依存に関して，移植元と移植先で代替可能な識別子の有無を判定する必要がある．これは，移植先の全プロダクトクラス間の依存グラフに，T が依存するプロダクトクラス間の依存グラフが含まれるかという，部分グラフ同型判定としてのアプローチが

考えられるが，具体的なアルゴリズムへの落とし込みは今後の課題である．

また現段階での提案内容では，移植したテストメソッドの実行可能性のみを議論の対象としており，意味的に適切なテストが移植できるか否かを問題としていない．そのため，意味的に適切なテスト操作を提供できるよう，テストメソッド中の識別子の対応づけ方法を改善する必要がある．ここで，テストメソッドが依存する参照先について，識別子名の意味的な類似度や，構文木の形状などを利用した処理内容の類似度などを算出し，候補を順位づけることで，より意味的に類似した識別子を探し出し，書き換えの対象とすることができる．こうすることで，移植元と移植先の間で意味的な類似性を保ちつつ適切なテスト操作手順を提供できると期待できる．そうした中で，どのような方法が最も効果的かを今後明らかにする必要がある．

また，移植したテストコードが適切なテスト操作手順を提供できる場合であっても，結果を照合するための適切な期待値を備えていない場合も考えられる．テスト期待値の適切さを評価することは難しいため，利用者によって複数の候補から適切なものを選択したり，期待値部分を容易に変更できたりするような仕組みを整えることが求められる．さらに将来的には，本手法によって適切なテスト操作を，また他の手法によって適切なテストオラクルを検索し，両者を合成させることで自動的に適切なテストコードを作成することも考えられる．

6. 妥当性への脅威

6.1 内的妥当性

本論文で実施した調査実験の大部分を，著者の作成したプログラムによって行っている．そのため，意図しないプログラムの動作不備が存在し実験結果に影響を与えている可能性は否定できない．また，RQ1 におけるリポジトリのドメイン分類は著者の判断かつシングルチェックで行われたため，理想的なドメイン分類とは異なっている可能性がある．また RQ3 の結果について，テストコード実行に必要な外部ライブラリのインポートが必ず成功するという前提のもとでの結果であり，この前提が事実と異なる場合には合致するテストメソッドが示された結果よりも少なくなる可能性がある．

6.2 外的妥当性

RQ1 の調査はある時点での GitHub 上のすべての Java リポジトリを対象としたため，外的妥当性への懸念は少ない．一方で，RQ2 での調査は特定の基準で選択された 509 個のリポジトリのみを，また RQ3 での調査は移植元と移植先のパターンとして各ドメインで 1 組のみを対象としているため，この調査結果をすべてのリポジトリに対して一般化することは難しい．

6.3 構成概念妥当性

今回提案した内容は構想段階であるため，我々が定義した移植条件を満たすことで実際に移植先で実行可能性が保たれるかを実際に検証することはできておらず，我々の認識していない条件がさらに存在する可能性は否定できない．そうした検証も今後の課題であり，本論文では構想にとどまった提案内容をより洗練できる可能性がある．

7. おわりに

本研究では，既存プロジェクトからの再利用による Java テストコードの自動生成手法の提案に向け，移植後の実行可能性に焦点を当てたテストメソッドの移植方法の構想に基づいて，GitHub 上の OSS における既存テストコードの移植可能性について調査した．その結果，既存テストコードを再利用素材として使用できるリポジトリが GitHub 上に 1,862 件存在することや，大半のテストメソッドはテスト対象のプロダクトコードにたかだか 2 つの依存しか持たないこと，この依存関係を保つ移植によって本来必要なテストメソッドの平均 83%が生成できる可能性があることなどを実証的に示し，既存テストコードの再利用による自動生成手法が現実的かつ有益であることを示した．

今後の課題として，現実的な移植を試みるテストメソッド対象を拡大することや，意味的な要素を考慮した適切な識別子の対応づけ手法を考案し，動作可能なテストコードを作成するツールを実際に開発することなどがあげられる．

謝辞 本研究は JSPS 科研費 JP19K20240，JP18H04094 の助成を受けた．

参考文献

[1] Orso, A. and Rothermel, G.: Software Testing: A Research Travelogue (2000–2014), *Proc. Future of Software Engineering* (*FOSE*), pp.117–132 (2014).
[2] Fraser, G. and Arcuri, A.: Whole Test Suite Generation, *IEEE Trans. Softw. Eng.*, Vol.39, No.2, pp.276–291 (2013).
[3] Tan, S.H., Marinov, D., Tan, L. and Leavens, G.T.: @tcomment: Testing javadoc comments to detect comment-code inconsistencies, *Proc. IEEE 5th International Conference on Software Testing, Verification and Validation* (*ICST*), pp.260–269 (2012).
[4] Li, N., Escalona, A. and Kamal, T.: Skyfire: Model-Based Testing with Cucumber, *Proc. 9th International Conference on Software Testing, Verification and Validation* (*ICST*), pp.393–400 (2016).
[5] Almasi, M.M., Hemmati, H., Fraser, G., Arcuri, A. and Benefelds, J.: An Industrial Evaluation of Unit Test Generation: Finding Real Faults in a Financial Application, *Proc. 39th International Conference on Software Engineering: Software Engineering in Practice Track* (*ICSE-PT*), pp.263–272 (2017).
[6] Shamshiri, S., Rojas, J.M., Galeotti, J.P., Walkinshaw, N. and Fraser, G.: How Do Automatically Generated Unit Tests Influence Software Maintenance?, *Proc. 11th International Conference on Software Testing, Verifi-*

cation and Validation (*ICST*), pp.250–261 (2018).
[7] Pacheco, C. and Ernst, M.D.: Randoop: Feedback-Directed Random Testing for Java, *Proc. 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion* (*OOPSLA*), pp.815–816 (2007).
[8] Goffi, A., Gorla, A., Ernst, M.D. and Pezzè, M.: Automatic Generation of Oracles for Exceptional Behaviors, *Proc. 25th International Symposium on Software Testing and Analysis* (*ISSTA*), pp.213–224 (2016).
[9] Motwani, M. and Brun, Y.: Automatically Generating Precise Oracles from Structured Natural Language Specifications, *Proc. 41st International Conference on Software Engineering* (*ICSE*), pp.188–199 (2019).
[10] JBehave (online), available from ⟨https://jbehave.org⟩ (accessed 2020-07-30).
[11] Cucumber (online), available from ⟨https://cucumber.io⟩ (accessed 2020-07-30).
[12] Balog, M., Gaunt, A.L., Brockschmidt, M., Nowozin, S. and Tarlow, D.: DeepCoder: Learning to Write Programs (2016).
[13] Kapser, C.J. and Godfrey, M.W.: "Cloning considered harmful" considered harmful: patterns of cloning in software, *Empirical Software Engineering*, Vol.13, No.6, pp.645–692 (2008).
[14] Hummel, O., Janjic, W. and Atkinson, C.: Code Conjurer: Pulling Reusable Software out of Thin Air, *IEEE Software*, Vol.25, No.5, pp.45–52 (2008).
[15] Lu, Y., Chaudhuri, S., Jermaine, C. and Melski, D.: Program Splicing, *Proc. 40th International Conference on Software Engineering* (*ICSE*), pp.338–349 (2018).
[16] Sondhi, D., Rani, D. and Purandare, R.: Similarities Across Libraries: Making a Case for Leveraging Test Suites, *Proc. 12th International Conference on Software Testing, Validation and Verification* (*ICST*), pp.79–89 (2019).
[17] GitHub (online), available from ⟨https://github.com⟩ (accessed 2020-07-30).
[18] Gousios, G.: The GHTorent dataset and tool suite, *Proc. 10th Working Conference on Mining Software Repositories* (*MSR*), pp.233–236 (2013).
[19] GitHub API (v3) (online), available from ⟨https://developer.github.com/v3/⟩ (accessed 2020-07-30).

西浦 生成 （学生会員）

平成 30 年京都工芸繊維大学大学院工芸科学研究科博士前期課程情報工学専攻修了．修士（工学）．同大学院工芸科学研究科博士後期課程設計工学専攻在学中．平成 28 年より国立研究開発法人産業技術総合研究所リサーチアシスタント．ソフトウェアのテスト・不具合局所化に関する研究に従事．

水野 修 （正会員）

平成 11 年大阪大学大学院基礎工学研究科助手．平成 13 年博士（工学）大阪大学．平成 22 年京都工芸繊維大学大学院工芸科学研究科准教授．平成 29 年同情報工学・人間科学系教授．主にソフトウェアのバグ検出手法，ソフトウェアリポジトリのマイニングに関する研究に従事．IEEE 会員．

崔 恩瀞 （正会員）

平成 27 年大阪大学大学院情報科学研究科博士後期課程修了．同年同大学院国際公共政策研究科助教．平成 28 年奈良先端科学技術大学院大学情報科学研究科助教．平成 30 年より同大学先端科学技術研究科助教（改組による）．平成 30 年より京都工芸繊維大学情報工学・人間科学系助教．博士（情報科学）．コードクローン管理やリファクタリング支援手法に関する研究に従事．