一般社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report

汎用的なテストアサーション命令に対する自動リファクタリングの提案

野田　拡志†　　柗本　真佑†　　楠本　真二†

† 大阪大学大学院情報科学研究科
E-mail: †{hir-noda,shinsuke}@ist.osaka-u.ac.jp

あらまし　ソフトウェアテストは，プロダクトの信頼性を保証するための重要な要素であり，通常のソースコードと同様に可読性が求められる．テスト内で実行結果を確認するアサート部では，可読性を下げる要因として，アサーションに含まれる間接的な表現が挙げられる．間接的な表現は可読性の低下だけでなく，アサーション失敗時の原因特定能力の低下も招く．特に真偽値検証や同値検証の検証用メソッドは汎用性が高く，間接的な表現を招きやすい．そこで本研究では汎用的なアサーションから具体的なアサーションへの自動リファクタリングを提案する．3 種類の汎用アサーションをリファクタリング対象として定義し，それら以外にも 1 種類をリファクタリング対象として定義する．提案手法では，定義したリファクタリング対象ごとに異なる自動リファクタリングを行う．評価実験の結果，提案手法によって対象の 1,557 プロジェクトから 4,418 箇所のリファクタリングに成功した．

キーワード　ソフトウェアテスト，汎用アサーション，自動リファクタリング，可読性

1. はじめに

ソフトウェア開発においては，プログラムによってプログラムの正しさを検証する自動テスト [1] が欠かせない．プログラムに対する検証作業そのものの自動化によって，バグの迅速な発見が可能となる．またテストコードは，検証対象となるプロダクトを理解するための一種のドキュメントとしても活用できる．テストは継続的インテグレーション [2] や DevOps [3] などの自動化プロセスとも親和性が高く，テストを満たすようにプロダクトを記述するテストファースト開発や，テスト自体を開発作業の基準として活用するテスト駆動開発 [4] などの開発形態の必須構成要素である．

自動テストで用いられるテストコードには，一般的なプロダクト側のソースコードと同様，可読性等の品質が求められる．意図が汲み取りにくいテストコードは，検証内容そのものの誤りにも繋がる．テストコード固有のプラクティスも多数存在しており，テストには分岐を含めないこと（過度に複雑化しないこと）[5]，1 つのテストには 1 つのシナリオのみを記載すること [6]，などが広く知られている．また，テストコード内の設計や実装の問題箇所を表すテストスメル [7] という概念も提案されている．スリープ文を含むテストや条件式を含むテストはスメルの一種である．

本研究では, テストコードの可読性低下の要因となりうる汎用アサーションについて考える．アサーションとは, テスト内での実測値と期待値の比較処理であり, 複数の実現方法が存在する．このアサーションの記法において, `assert(…).isEqualTo(…)` や `assert(…).isTrue()` などの様々な場面で利用できる汎用アサーションは，可読性が低くなる傾向にあると考えられる．例えば null 参照を防ぐためのアサーション，`assert(actual != null).isTrue()` はテストにおける典型的なアサーションではあるが可読性は低い．意味論として捉えると「実測値が null でないという条件式の結果が真」であり，間接的な表現であるといえる．この汎用アサーションを，null 参照チェックに特化した具体アサーション `assert(actual).isNotNull()` と変換すれば，「実測値が null ではない」という等価でより直接的な表現へのリファクタリングが可能となる．

汎用アサーションは可読性だけでなく，テスト失敗時の原因特定能力の低下にも繋がると考えられる．具体アサーションは汎用アサーションに対する単なる等価な構文ではない．呼び出す検証用メソッドが異なるため，検証用メソッドが吐き出すテスト失敗時のメッセージも異なる．先述の null 参照の例では，汎用アサーションの場合は「対象が真だと期待したが偽であった」というメッセージが得られる．この対象とは `actual != null` の評価結果，すなわち真偽値そのものである．よって検証用メソッドは検証対象がどのような式であったかは知り得ず，検証失敗時のメッセージには null 参照の検証に失敗したという情報は含まれない．他方，具体アサーションであれば検証対象として実測値そのものを渡し，null 参照に特化した検証用メソッドによる検証が可能となる．結果的に，「期待値が null であった」というより問題箇所の特定に役立つメッセージが得られる．

本研究では，テストコードの可読性とテスト失敗時の原因特定能力の向上を目的として，汎用アサーションに対する自動的なリファクタリング手法を提案する．本稿では汎用アサーションを 4 種類に分類し，各アサーションの性質や悪影響について議論する．この分類結果に基づき，各汎用アサーションの具体アサーションへの自動的なリファクタリング方法を提案する．提案手法の適用可能性を調べるために OSS プロジェクトを題材とした評価実験を行う．実験により，対象プロジェクトに含まれる全 239,324 個のアサーションの中から，4,418 箇所（約

This article is a technical report without peer review, and its polished and/or extended version may be published eleswhere.
Copyright ©2025 by IEICE

テストコード

```
1 @Test public void test01() {
2   List l = {4, 8, -5, 0};
3   List a = getPositives(l);
4   assert(a.size()).isEqualTo(2);
5   assert(a.get(0)).isEqualTo(4);
6   assert(a.get(1)).isEqualTo(8);
7 }
```

```
1 @Test public void test02() {
2   List l = {-5, 0};
3   List a = getPositives(l);
4   assert(a != null).isTrue();
5   assert(a.size() == 0).isTrue();
6 }
```

テスト対象(プロダクトコード)

```
1 // リストから自然数を返す
2 List getPositives(List l) {
3   List a = new List();
4   for(v in l)
5     if (v > 0) a.add(v);
6   return a;
7 }
```

図 1: テストコードの例

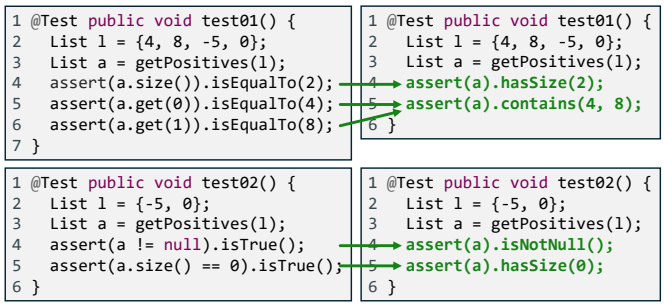

図 2: 汎用アサーションに対するリファクタリングの例

1.8%）に対してリファクタリングの成功を確認した．また，ある種類の汎用アサーションに対するリファクタリング結果が別の種類の汎用アサーションに該当する，つまり，複数回のリファクタリングができるアサーションが存在する．実験では，204 箇所に対して複数回のリファクタリングに成功した．

2. 準　　備

2.1 自動テスト

自動テストとは自動的なソフトウェアの検証方法であり，プログラムを用いてプログラムの正しさを確認する． 図 1 に JUnit を用いた Java の単体テストの具体例を示す．この例はテスト対象となるプロダクトコードと，2 個のテストケースで構成されてる．テスト対象は整数のリストを受け取り，自然数のみを含むリストを返す `getPositives()` メソッドである．

一般的にテストケースは以下 2 つの要素で構成されている．

- テスト対象（プロダクト）を実行する実行部
- 実行結果が期待値と同一かを検証するアサート部

1 つ目のテストケース `test01()` では 2 行目から 3 行目が実行部である．`getPositives()` への入力リストの定義とメソッド呼び出しを行っている．4 行目から 6 行目がアサート部である．`getPositives()` の実行により得られた実測値 `a` を，`assert()` メソッドによって期待値との比較を行っている．全てのアサート文が成功，すなわち期待値との比較に成功した場合はテストは成功であり，`getPositives()` が正しい振る舞いを行っていると判断される．期待値との比較に失敗した場合は，テストは失敗となり．テスト対象である `getPositives()` の振る舞いが不適切であると判断できる．

アサート部を構成するアサーション文（以降，単にアサーションと呼ぶ）は，以下の 3 つの要素で構成される．

- 実測値
- 期待値
- 検証用メソッド

例えば，`test01()` の `assert(a.size()).isEqualTo(2)` に対しては `a.size()` が実測値であり，2 が期待値，`isEqualTo()` が検証用メソッドである．また `test02()` にある `isTrue()` ように期待値を持たない検証用メソッドも存在する．検証用メソッド単体で期待値を表現しているといえる．

検証の実現方法も様々であり，数多くのアサーションライブラリが提案されている．JUnit フレームワークが標準で提供する検証用メソッドだけでなく，Hamcrest や AssertJ 等が広く利用されている．特に AssertJ は，メソッドチェーンに基づいた多くの検証用メソッドを提供しており，豊かな表現力を持つ．AssertJ のコンセプトは *“fluent assertions”* であり [8]，流れるようなアサーションが可能という特徴である．具体的には `assertThat(firstPrimeNumber).isEqualTo(2)` のような英文法のような，自然な表現が可能である．

2.2 汎用アサーションによる可読性の低下

実測値と期待との比較には様々な記述方法が存在しており，中には可読性を低下させるような記述もある．本研究ではその一つとして，汎用的な検証用メソッドによって記述されたアサーション（汎用アサーション）について考える． 図 1 で示していたアサーションは全て汎用的なアサーションで表現されており，可読性の低下を引き起こしている．

図 2 は， 図 1 の汎用アサーションをリファクタリングした結果である．一例として `test01()` の 4 行目のアサーションを考える．このアサーションでは期待値としてリスト `a` の要素数を取り出し，その値と数値 2 の一致を `isEqualTo()` により確かめている．`isEqualTo()` は 2 つのオブジェクトの等価性を判定する検証用メソッドであり，様々なアサーションに利用できる汎用的な方法だといえる．他方，右側のテストコードでは，リスト要素の要素数検証に特化した `hasSize()` によってアサーションを実現している．これによって検証対象となる実測値をリスト変数 `a` に固定した上で，リストに特化した検証用メソッドによる内部構造の検証が可能となる．意味論として考えると，リファクタリング前は「`a` の要素数が 2 と同じである」というやや間接的な表現であるのに対し，リファクタリング後は「`a` は 2 個の要素を持っている」という直接的な表現となっている．結果的に可読性の改善に繋がると考えられる．

汎用アサーションから具体アサーションへのリファクタリングは可読性の改善みならず，テスト失敗時の原因箇所の特定能力の改善に繋がる可能性もある．テスト失敗時には検証用メソッドが AssertionError 等の例外を出す．この例外にはどのような検証に失敗したのか，という原因が文字列として含まれている．つまり検証の内容は同じでも，検証の方法（アサーション）によって得られるメッセージは異なる．

図 3 にアサーションに失敗した際の例外メッセージの一例を示す(注1)．この図は 図 2 の 4 行目（要素数検証）に失敗した

(注1)：AssertJ で実行した場合のエラーメッセージを示している．左側では

汎用アサーション

```
org.junit.ComparisonFailure:
expected:<[2]> but was:<[3]>
```

具体アサーション

```
java.lang.AssertionError:
Expected size: 2 but was: 3
in: [4, 8, 0]
```

図 3: 汎用アサーションと具体アサーションの例外メッセージ

際の例外メッセージであり，左が汎用アサーションの，右が具体アサーションのメッセージ内容である．汎用アサーションでは「2 を期待したが 3 であった」というメッセージが得られている．しかし何に対する 2 であるかは明記されておらず，検証失敗の原因を判別しにくい．具体アサーションでは「要素数 2 を期待したが 3 であった」となっており，要素数に関する検証失敗であることが容易に判別できる．またその下部にはリスト内部の要素が列挙されている．この情報から，`getPositives()` の結果に誤って 0 が含まれてしまっている，というバグの原因箇所の類推も可能である．

3. 汎用アサーションに対するリファクタリング

3.1 概　　要

本研究の目的は自動テストにおける可読性，および検証失敗原因の特定能力を低下させる汎用アサーションに対する自動リファクタリングの実現である．本節では，まずリファクタリング対象の汎用アサーションを分類する．この分類に従い，各汎用アサーションの検出と変換の方法を説明する．

3.2 汎用アサーションの分類

リファクタリング対象となる汎用アサーションを 4 種類に分類する．この 4 種類はソースコードの言語やアサーションライブラリに依存しない，一般的な考えである．

3.2.1 条件式を含む実測値（条件式アサーション）

2 で示した `assert(a != null).isTrue()` 等の実測値に条件式を含むアサーションである．実測値そのものが比較演算等を用いた式として表現されており，実測値の中で期待値との比較を行っているとも見なせる．結果的に検証用メソッドは `isTrue()` や `isFalse()` などの汎用的な検証用メソッドとなる．この場合，AssertJ が示す流れるようなアサーションが実現できず，可読性低下の要因となると考えられる．

テスト失敗時のメッセージには，条件式の比較結果という間接的な情報のみが含まれるため，検証失敗の原因が特定しにくい．

3.2.2 アクセス演算子を含む実測値（限定的アサーション）

実測値にメソッド呼び出しのドット"."や配列アクセス"[]"等のアクセス演算子を含むアサーションである．`assert(msg.length()).isEqualTo(4)` や `assert(a[0]).isEqualTo(1)` などが該当する．検証対象である実測値を直接利用せず，実測値の内部要素を取り出して（限定して）から検証する，という観点で限定的アサーションと呼ぶ．先述の条件式アサーションと同様，流れるようなアサーションを破壊する要因となる．

実測値の中で要素を限定しているため，テスト失敗時のメッセージに実測値そのものの情報は出力されない．結果的に，問題箇所の限定能力は低くなりがちである．

3.2.3 特殊値を含む期待値（特殊値アサーション）

期待値に `true` や `false`，`null`，`0` 等の特殊値が含まれるアサーションである．例えば，`assert(a).isEqualTo(true)` や `assert(a).hasSize(0)` などのアサーションが該当する．`true` や `false` 等の特殊値はそれ以外の値と比べて特殊な意味を持つケースが多く，専用のアサーションが設けられている場合がある．`assert(a).isEqualTo(true)` に対しては `assert(a).isTrue()` と，`assert(a).hasSize(0)` は `assert(a).isEmpty()` へと変換できる．いずれも，特殊値を用いた表現に比べて，直接的だといえる．

なお，変換前後の両方で，テスト失敗時に「true を期待していたが false であった」といったメッセージが得られるため，問題個所の限定能力は変わらない．

3.2.4 同一実測値の連続したアサーション（連続アサーション）

ある特定の実測値に対する複数の独立したアサーション文である．例えば実測値 a について，`assert(a)…; assert(a)…;` と記述している場合が該当する．テストのプラクティスの一つとして，一つのテストケースには一つのアサーションとすべき，という考えがある [9]．テストケース自体を単一のシナリオ，かつ小さく明確にするための指針である．連続アサーションはこのプラクティスに従わない記法である．AssertJ ではメソッドチェーンを用いたアサーションが強く推奨されており，連続アサーションを単一アサーションに変換可能である．

なお，検証用メソッドは変化しないため，テスト失敗時のメッセージは変わらず，問題個所の限定能力は変わらない．

3.3 自動リファクタリングの流れ

汎用アサーションの定義に基づき，具体的な自動リファクタリングの方法について考える．3 つの行程を順に紹介する．

3.3.1 アサーションの検出

テストコードに対する静的解析により 4 種類の汎用アサーションを検出する．まずは，テストコード内に含まれる全てのアサーションを特定する．この特定は，各種アサーションライブラリが提供するアサーション文の冒頭のメソッド（`assertEquals()` や `assertThat()` など）の完全限定名に基づいて行う．次に発見したアサーションから，実測値と期待値，検証用メソッドの 3 要素を取り出す．

3.3.2 汎用アサーションの識別

各アサーションを構成する 3 要素に基づき，汎用アサーションの種類を識別する．識別方法は分類の定義の通りである．例えば，実測値に式を含む場合，条件式アサーションである．

3.3.3 具体アサーションへの変換

等価な具体アサーションへの変換方法は，汎用アサーションの種類ごとに異なる．4 種類の変換方法について説明する．

条件式アサーション：式の比較演算子（==や!=など）と検証用メソッド（`isTrue()` や `isFalse()`）を確認し，等価な検証用メソッド（`isEqualTo()`，`isNotEqualTo()` など）への変

JUnit のエラーが出ているが，これは AssertJ 内で JUnit が呼び出されているためである．

表 1: 対象とする汎用アサーションの具体的な条件

分類	条件	一例
条件式	実測値に一致比較	`assert(a == …).isFalse()`
	実測値に不一致比較	`assert(a == …).isTrue()`
	実測値に型比較	`assert(a instanceof …).isTrue()`
限定的	集合の要素数の一致	`assert(a.size()).isEqualTo(…)`
	文字数の一致	`assert(a.length()).isEqualTo(…)`
	集合の個別要素の一致	`assert(a.get(…)).isEqualTo(…)`
特殊値	true/false/null の比較	`assert(…).isEqualTo(true)`
		`assert(…).isNotEqualTo(null)`
連続	同一実測値の連続比較	`assert(a)…; assert(a)…;`

換を試みる．変換可能な場合は実測値の式の右辺がリテラルならば右辺を，左辺がリテラルならば左辺を，どちらにも当てはまらなければ右辺を実測値とする．

限定的アサーション：アクセス元（`list.size()` では `list`）の型または継承元と，アクセス先（`list.size()` では `size()`）の組み合わせで，アクセスしている具体的な要素を特定する．例えば，アクセス先 `size()` に対してアクセス元が java.util.Collection である，もしくは継承していた場合，実測値は集合の要素にアクセスしていると特定できる．特定した具体的な要素と，検証用メソッド（`isEqualTo()` など）から，等価な検証用メソッド（`hasSize()` など）への変換を試みる．変換可能な場合，アクセス元単体を実測値とする．

特殊値アサーション：期待値と検証用メソッドを見て，等価な特殊値専用の検証用メソッドへの変換を試みる．変換可能な場合，期待値は不要のため検証用メソッドの引数を無くす．

連続アサーション：連続するアサーションの実測値を見る．実測値が同じ場合，AssertJ の場合はメソッドチェーンを用いて，Hamcrest の場合は `allOf()` メソッドを用いて複数の検証用メソッドを 1 つのアサーションで記述する．

3.4 リファクタリング対象の条件

各種類の汎用アサーションの中でも様々な比較内容が存在する．例えば，特殊値アサーションでは `assert(a).isNotEqualTo(null)` 等の「null の一致比較（不一致比較）」の他に，`assert(a).hasSize(0)` 等の「要素数が 0 であるかの比較」も該当する．しかし，理論上リファクタリング可能であるが，実践的なプロジェクトにおいて多く登場する比較内容は限られている．そこで，事前に数個の OSS を対象として，どのような比較内容の汎用アサーションが多いかを調査し，それに基づいて各種類においてリファクタリングすべき対象の条件を決定した．先述の例では，前者の方が OSS 中に多く登場する． 表 1 は提案手法によるリファクタリング対象の条件である．

3.5 リファクタリングの適用順序

4 種類の汎用アサーションのリファクタリングにおいては，どの種類に対するリファクタリングを優先するかが重要である．ある種類の汎用アサーションのリファクタリングが別種類の汎用アサーションに該当する場合があるためである．例えば，`assert(a != null).isTrue()` は，条件式アサーションの変換を適用すると `assert(a).isEqualTo(null)` となる．この結果は，期待値に特殊値を含む特殊値アサーションであると判断できるので，特殊値アサーションの変換を適用し `assert(a).isNotNull()` となる．もし特殊値アサーションを優先して適用してしまうと，上記の結果は得られない．元のアサーションでは期待値側に特殊値 null を含まないためである．さらに，実測値部分が `a != null` から `a` と変換されるため，新たに連続アサーションによって前後のアサーションと統合可能になる場合もある．

本稿では，条件式アサーションから限定的アサーション，特殊値アサーション，連続アサーションの順に変換を実施する．`assert(a == null).isEqualTo(true)` 等の例では特殊値アサーションと条件式アサーションの順序を逆にすべきだが，提案手法では考慮しない．この例では一致比較の検証用メソッドを使用しているにも関わらず，実測値部分でも一致比較を行っている．つまり二重に一致比較を行っているといえる．このような検証は極めて少ないと考えたため，考慮外とした．

4. 実験設定

4.1 実験概要

提案するリファクタリング手法の適用可能性を実験的に確かめる．題材はオープンソース（OSS）に含まれるテストコードである．JUnit 標準アサーション（JUnit(注2)）と Hamcrest, AssertJ の各ライブラリごとに汎用アサーションをリファクタリングする．なお，ライブラリによっては変換先がない汎用アサーションが存在する．例えば，AssertJ や Hamcrest にはリストに対して要素数を検証する検証用メソッド `hasSize()` が存在するが，JUnit には存在しない．そのため，限定的アサーションに該当する `assertEquals(3, list.size())` をリファクタリングできない．異なるライブラリを変換先にする手法も考えられるが，本研究では異なるライブラリへのリファクタリングは行わない．本研究では，あくまでライブラリに依存しないリファクタリング手法の評価を目的としているためである．変換先が存在しない場合には検出数として計測する．

4.2 評価指標

4.2.1 リファクタリング成功回数と検出数

提案手法を適用した結果の変換数を計測する．この指標は，提案手法によって OSS 中の汎用アサーションをどれだけ改善できるかを表している．

また，変換先が存在しない汎用アサーションの数も検出数として計測する．検出数は，可読性に問題があるが，ライブラリの仕様上改善できない汎用アサーションが OSS 中にどの程度存在するかを表している．

4.2.2 リファクタリング前後の等価性

リファクタリングは *"behavior preserving code transformation"* [10] であるため，変換前後のアサーションの等価性が必

(注2)：JUnit はテストフレームワークでありアサーションライブラリではないが，本稿では便宜上 JUnit と省略する．厳密には org.junit.jupiter.api.Assertions や org.junit.Assert である．

要である．ここではOSSに提案手法を適用した後，検出と変換が行われたものをライブラリごと，及び汎用アサーションの種類ごとにランダムで10個取り出す．取り出されたものに対して，リファクタリング前後のテストコードを目視で確認し，リファクタリング前後の等価性を評価する．また，検出のみが行われる汎用アサーションに対しても正しい検出かを目視で確認する．ただし，この評価によって完全に等価性が保証されるわけではないため，今後追加の実験が必要であると考えられる．例えば，変換前後でテストの実行結果が同じか，バグを含んだプログラムに対しテストが失敗するかなどの実験が必要だと考えられる．

4.3 実験題材

実践的なソフトウェア開発を行うプロジェクトを実験対象とするため，以下の選定基準を設けた．この基準に従うOSS中の1,557プロジェクトをリファクタリングの対象とした．

- GitHub上で公開
- 言語がJava
- star数上位100リポジトリ
- MavenかGradleを利用
- ディレクトリ構成がビルドツールの初期構成に従う
- フォルダ名やプロジェクト名に"sample"や"example"，"tutorial"の文字列を含まない．

上記プロジェクトの中からテストコードを抽出し，さらにアサーションを取り出した．最終的に得られた239,324個のアサーションが本実験におけるリファクタリングの対象である．

5. 実験結果と考察

5.1 総リファクタリング成功箇所数と成功回数

実験対象とした全アサーション239,324個に対して，4,418個（1.84%）のアサーションのリファクタリングに成功した．割合としては1.8%程度に留まっているが，239,324個はリファクタリング対象ではないアサーションも数多く含まれている．OSSのテストコードの約1.8%には，可読性や問題特定能力の低下を引き起こす汎用アサーションが含まれているともいえる．また，リファクタリング成功回数（リファクタリングができた回数）は4,622回であった．リファクタリング箇所よりリファクタリング成功回数が多いことから，複数回のリファクタリングが行われているアサーションの存在が分かる．

5.2 プロジェクト別リファクタリング成功箇所数

個別のプロジェクトに着目すると，アサーションの数が100個以上あった237プロジェクトの中で，最も多かったプロジェクトでは640箇所のリファクタリングに成功した．また，アサーションの総数に対するリファクタリングできた割合が最も多かったものでは，26.0%が変換された．提案手法によって大きく可読性を改善できるプロジェクトも存在していると分かる．

5.3 汎用アサーション分類別リファクタリング成功箇所数

汎用アサーションの種類ごとのリファクタリング成功箇所数と検出数を各ライブラリごとに説明する．

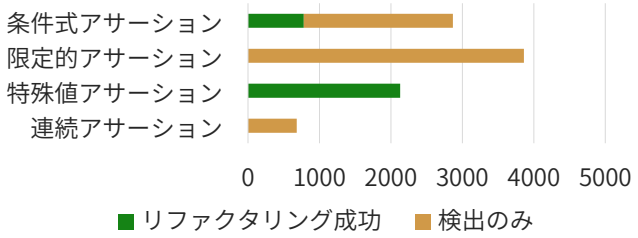

図4: JUnitに対するリファクタリング成功箇所数と検出数

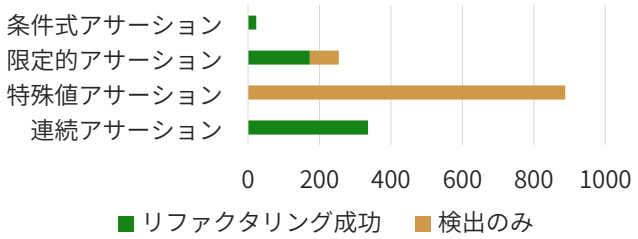

図5: Hamcrestに対するリファクタリング成功箇所数と検出数

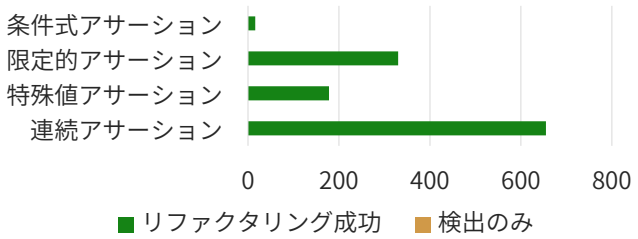

図6: AssertJに対するリファクタリング成功箇所数と検出数

5.3.1 JUnitのリファクタリング成功箇所数と検出数

図4はJUnitの変換結果である．汎用アサーションの種類ごとに分けて表されている．特殊値アサーション，条件式アサーションの順にリファクタリング成功箇所数が多い．また，リファクタリング成功箇所数に対して検出数が多く，ライブラリの機能の少なさによって可読性を改善できない汎用アサーションが多いことが分かる．

5.3.2 Hamcrestのリファクタリング成功箇所数と検出数

Hamcrestのリファクタリング結果は図5の通りである．連続アサーションのリファクタリング成功箇所数が最も多く，次いで限定的アサーションが多い．一方で条件式アサーションのリファクタリング成功箇所数は23箇所と少ない．

5.3.3 AssertJのリファクタリング成功箇所数と検出数

AssertJのリファクタリング結果を図6に示す．Hamcrestと同様に，連続アサーション，限定的アサーションの順にリファクタリング成功箇所数が多い．また，条件式アサーションも16箇所であり，Hamcrestと同様に少ない．

5.4 リファクタリング順序の妥当性

5.1で先述したように，複数回のリファクタリングに成功したアサーションも確認された．限定的アサーションに対するリファクタリングが適用されたのちに連続アサーションへのリファクタリングが適用されたアサーションを76件，条件式

アサーションのリファクタリング後に特殊値アサーションのリファクタリングが行われたアサーションは 128 件存在した．変換順序の工夫によってリファクタリング成功回数が増加する場合があると分かる．なお，条件式アサーションの後に特殊値アサーションのリファクタリングより先に行った場合，特殊値アサーションと条件式アサーションのリファクタリングが適用されたアサーションは 27 件であった．この順序でリファクタリングできるアサーションは考慮外としていたが，無視できないほどの数があると分かる．また，条件式アサーションの前に特殊値アサーションに対する変換を追加し，条件式アサーションへの変換を 2 回行えば，更にリファクタリング成功回数が増加すると分かる．

5.5 リファクタリング前後の等価性実験

リファクタリング前後でのアサーションの等価性を評価指標の通り評価した結果，取り出したすべての変換で等価性が保たれていた．また，複数回の変換が適用される場合でも各変換で等価性が保たれていた．例えば，alibaba / canal プロジェクト内のサブプロジェクト parse(注3) に存在した `assertTrue(table.findColumn("value1") != null)` のリファクタリング経過を見ると，条件式アサーションに対する変換の適用後は `assertNotEquals(null, table.findColumn("value1"))` となり，特殊値アサーションに該当する．このアサーションは特殊値アサーションへのリファクタリングが適用され，`assertNotNull(table.findColumn("value1"))` となった．複数回のリファクタリングにおいても等価な変換に成功していると分かる．よって，提案手法の適用前後でアサーションの動作は変化しないと考えられる．

6. 制約と展望

6.1 多対一のリファクタリング

本研究では，連続アサーション以外において，汎用アサーションと具体アサーションは一対一であったが，複数の汎用アサーションを 1 つの具体アサーションに変換できる場合もある．例えば，図 1 に示した `test01()` の 4 行目から 6 行目は `assertThat(a).containsExactly(4, 8)` という等価な記述に変換できる．間接的な表現が消えるだけでなく，3 つの検証用メソッドが 1 つの検証用メソッドにまとまっており，可読性の大きな向上が期待される．

6.2 他言語への適用

本稿で提案する汎用アサーションの分類は言語に依存しないため，他言語のアサーションも同様の手法で改善できると考える．他言語でも，間接的なアサーションによる可読性の低下は指摘されている [11] [12]．この 2 つの記事では 4 種類の汎用アサーションの内，連続アサーションと特殊値アサーションに該当する汎用アサーションが指摘されているが，他 2 種の変換も他言語で適用可能だと考える．他言語にも AssertJ や Hamcrest のような，多種の検証用メソッドを提供するライブラリが存在している．例えば，本研究の実験で変換対象としたアサーションライブラリの Hamcrest は，Python や Ruby などの Java 以外の言語でも実装されている(注4)．また，Python では AssertJ を参考に開発された assertpy というライブラリも存在する(注5)．そのため，機能の少ない JUnit では変換できなかった限定的アサーションや連続アサーションについても他言語で変換の需要があると考えられる．

7. おわりに

本研究では汎用アサーションから具体アサーションへの変換手法を提案した．可読性に悪影響を与えるアサーションを 4 種類定義し，それぞれを等価かつ具体的なアサーションへの変換を行った．また，適用実験の結果，提案手法で OSS 中の約 1.8% の汎用アサーションに対してリファクタリングできた．

今後の課題としては，変換条件に当てはまらない比較内容に対する実験が挙げられる．特に，hasSize() は OSS 中に多く登場したため，hasSize(0) から isEmpty() への変換のような，hasSize() を含む汎用アサーションのリファクタリング成功箇所数は計測されるべきだと考える．

謝辞 本研究の一部は，JSPS 科研費（JP24H00692, JP21H04877, JP21K18302）による助成を受けた．

文　献

[1] E. Dustin, J. Rashka, and J. Paul, Automated software testing: introduction, management, and performance, Addison-Wesley Longman Publishing Co., Inc., 1999.
[2] P.M. Duvall, S. Matyas, and A. Glover, Continuous Integration: Improving Software Quality and Reducing Risk, Pearson Education, 2007.
[3] C. Ebert, G. Gallardo, J. Hernantes, and N. Serrano, “Devops,” IEEE Software, vol.33, no.3, pp.94–100, 2016.
[4] K. Beck, Test Driven Development: By Example, Pearson Education, 2022.
[5] L. Koskela, Effective Unit Testing: A guide for Java developers, Shelter Island, NY : Manning, 2013.
[6] M. Noback, “The single responsibility principle,” pp.3–10, Apress, 2018.
[7] W. Aljedaani, A. Peruma, A. Aljohani, M. Alotaibi, M.W. Mkaouer, A. Ouni, C.D. Newman, A. Ghallab, and S. Ludi, “Test smell detection tools: A systematic mapping study,” In Proceedings of the International Conference on Evaluation and Assessment in Software Engineering, pp.170–180, 2021.
[8] AssertJ, “Assertj - fluent assertions java library,” https://assertj.github.io/doc/, Accessed at 2025-02-03.
[9] R.C. Martin, Clean Code: A Handbook of Agile Software Craftsmanship, Prentice Hall PTR, 2008.
[10] W.F. Opdyke and R.E. Johnson, “Creating abstract superclasses by refactoring,” In Proceedings of conference on Computer science, pp.66–73, 1993.
[11] JetBrains, “Equality assertion can be simplified,” https://www.jetbrains.com/help/inspectopedia/RsAssertEqual.html, Accessed at 2025-02-03.
[12] ashleyfrieze, “Lower level assertion,” https://codingcraftsman.wordpress.com/2024/08/12/lower-level-assertion/, Accessed at 2025-02-03.

(注3)：https://github.com/alibaba/canal/tree/master/parse

(注4)：https://hamcrest.org/

(注5)：https://github.com/assertpy/assertpy