特別研究報告

題目

テストケースに含まれる汎用アサーションに対する自動リファクタリング

指導教員

楠本 真二 教授

報告者

野田 拡志

令和 7 年 2 月 6 日

大阪大学基礎工学部情報科学科

令和 6 年度 特別研究報告

テストケースに含まれる汎用アサーションに対する自動リファクタリング

野田 拡志

内容梗概

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と保守性が求められる．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，アサーションに間接的な表現を用いていることが挙げられる．例えば，`assert(a != null).isTrue()` は「a が null でないという条件式の結果が真である」という間接的な表現である．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アサート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．評価実験として，1,557 個のオープンソース開発プロジェクトで使われている実際のテストコードに対して適用実験を行った．その結果，提案手法によって対象の全プロジェクトから 4,418 箇所のリファクタリングに成功した．

主な用語

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

目次

図目次

表目次

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（CI）[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（TDD）[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，可読性や保守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象アサーション `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．具象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検証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，具象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象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 1.8%）で 4,622 回の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 箇所で複数回のリファクタリングに成功した．

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

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．テスト対象は数値配列を受け取り，自然数のみを含む数値配列を返す `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()` への入力配列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，メソッドチェインに基づいた多くの検証用メソッドを提供しており，豊かな表現応力を持つ．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．実際に AssertJ は，JUnit 標準の検証用メソッドや Hamcrest と比較した際の，テストの理解に要する時間の短さや生産性の高さが示されている [9][10]．

2.2 テストコードの品質

テストコードは一種のソースコードであるため，プロダクトコードと同様に品質が重要である．Athanasiou らの研究によれば，テストコードの品質は開発者の生産性と相関があると結論付けられている [11]．しかし，テストコードの品質はプロダクトコードに比べ軽視されやすい．例えば，テストコードの可読性はプロダクトコードより低いという傾向も報告されている [12][13]．また，テストコードにおける品質低下の要因は様々である．Spadini らの研究では，テストスメル（テストコード内の設計や実装の問題箇所）とソフトウェアの品質の関係を調査している [14]．調査の結果，Indirect Testing と Eager Test，Assertion Roulette の 3 つのテストスメルが特にテストコードのメンテナンス頻度との相関が高いと報告されている．Indirect Testing は本来検証したいオブジェクトに別のオブジェクトを介していてテストしているなどの間接的なテストであり，Eager Test は 1 つのテストケースで，テスト対象オブジェクトの複数のメソッドを検証している状態を指す．以上の 2 つは実行部における品質低下の要因である．それに対し，Assertion Roulette はテスト失敗時のメッセージが設定されていない複数のアサーションが存在する状態を指し，これはアサート部において品質低下の要因となる．本研究では，Assertion Roulette と同様にアサート部の品質に注目する．

2.3 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アサーションもある．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

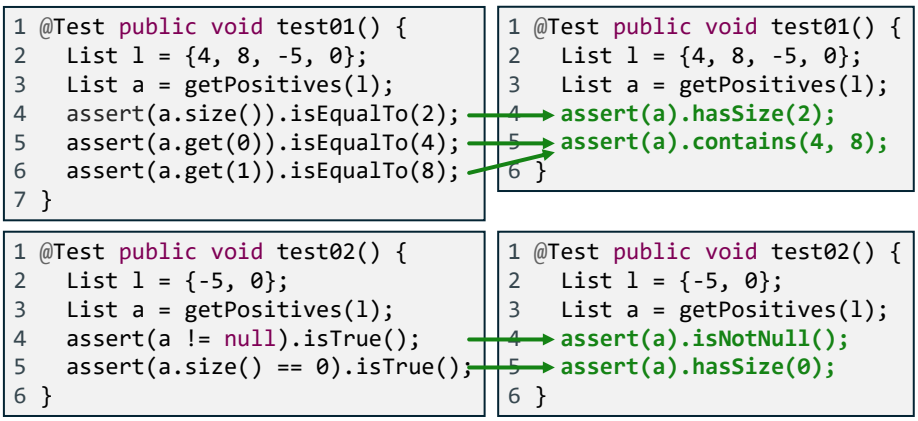

図 2　汎用アサーションに対するリファクタリングの例

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配列 `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，配列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．

また `test02()` の 4 行目では期待値自体が式（`a != null`）となっており，その評価の結果を `isTrue()` という汎用的な検証用メソッドで検証している．これに対しては `isNotNull()` という null 検証用の目的特化な検証用メソッドに変換している．「ある変数が null と等価でないという条件式が真」という間接的なアサーション表現を，「ある変数が null ではない」という直接的な表現への置き換えが可能となる．結果的にアサーションの意図が明確となり可読性が向上すると考えられる．

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 ComparisonFailure や 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．つまり検証の内容自体は同じでも，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[*1]．この図は 図 2 の 4 行目（要

[*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 JUnit のエラーが出ているが，これは AssertJ 内で

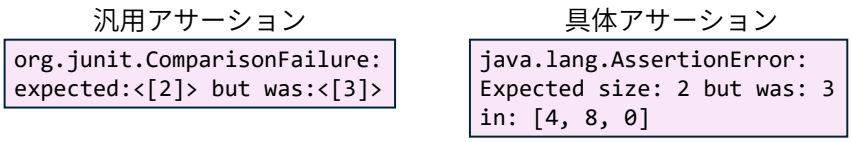

図 3　汎用アサーションと具体アサーションの例外メッセージ

素数検証）に失敗した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．またその下部には配列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．

JUnit が呼び出されているためである．

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となる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について説明する．

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 が示すような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．また，失敗時のメッセージでは条件式の比較結果という間接的な情報のみとなるため，検証失敗の原因が特定しにくい．

この汎用アサーションに対しては，実測値を単一の変数やインスタンスとした上で，等価かつ具体的な検証用メソッドへの変換で可読性を確保できる．例えば，`assert(a.size() == 0).isTrue()` の場合は実測値 `a.size()` と期待値 `0` の一致を検証すればよいため，`assert(a.size()).isEqualTo(0)` と変換できる．

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．また，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は実測値そのものは出力されない．結果的に，問題箇所の限定能力は低くなりがちである．

これらに対しては条件式アサーションと同様，実測値を単一の変数やインスタンスとした上で等価で具体的な検証用メソッドへ変換する．例えば，String インスタンスに対する文字長のアサーション

`assert(msg.length()).isEqualTo(4)` に対しては，`assert(msg).hasSize(4)` に変換可能である．AssertJ の場合，テスト失敗時のメッセージには実測値そのものの情報（リストであれば要素の中身）が含まれるため，問題箇所の限定能力は高くなる．

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．

特殊値アサーションの修正においては，検証用メソッドを等価な特殊値専用の検証用メソッドに変換すればよい．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 [15]．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 ではメソッドチェイ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．

連続アサーションの検出と変換は容易である．同一の実測値かつ連続したアサーションを検出し，メソッドチェインに置き換えれば良い．ただし，実測値の中で副作用を含むメソッドを呼び出している場合，変換によってテストが破壊される場合がある．例えば，`assert(queue.pop())..; assert(queue.pop())..;` では，`queue.pop()` がオブジェクトに対する作用を持っているため，変換によって振る舞いが変わってしまう．本稿では作用の有無は調べず，実測値にメソッド呼び出しを含む場合を変換対象外とする．

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．

- 実測値に式を含む場合，条件式アサーション
- 実測値にアクセス演算子を含む場合，限定的アサーション
- 期待値に `true` や `false` など特殊な値を含む場合，特殊値アサーション
- 実測値が同一かつ連続したアサーションがある場合，連続アサーション
- 上記のどれにも該当しない場合，汎用アサーションがないと判断し，リファクタリング対象外

3.3.3 具体アサーションへの変換

等価で具体的なアサーションへの変換方法は，汎用アサーションの種類によって異なる．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，期待値は必要なくなる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

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
条件式アサーション	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()`
限定的アサーション	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(…)`
特殊値アサーション	true/false/null の比較	`assert(…).isEqualTo(true)`
		`assert(…).isNotEqualTo(null)`
連続アサーション	同一実測値の連続比較	`assert(a)…; assert(a)…;`

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．ただし，実測値がメソッドの場合は先述の通りの理由により，変換を行わない．例えば AssertJ の場合，`assert(list).isNotNull(); assert(list).hasSize(0);` は `assert(list).isNotNull().hasSize(0);`，Hamcrest の場合，`assert(list, notNull()); assert(list, hasSize(0));` は `assert(list, allOf(notNull(), hasSize(0)));` と変換される．

なお，アサーションライブラリによっては変換先となる等価な検証用メソッドが存在しない場合がある．例えば，AssertJ や Hamcrest には，リストの実測値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())` を変換できない．本稿では，このような場合をリファクタリング不可と捉える．

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)` から `assert(a).isNotNull()` 等の「null の一致比較（不一致比較）」のリファクタリングの他に，`assert(a).hasSize(0)` から `assert(a).isEmpty()` 等の「要素数が 0 であるかの比較」のリファクタリング等も可能である．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．なお，条件式アサーションの欄の一致比較と不一致比較には，オブジェクトの参照先が同じかを比較している場合も含まれる．String 型に対して==で比較を行う場合などである．

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によって，別種類の汎用アサーションのリファクタリングが可能とな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[*2]）と Hamcrest，AssertJ の各ライブラリごとに汎用アサーションのリファクタリングを行う．なお，先述の通りライブラリによっては変換先が存在しない汎用アサーションが存在する．異なるライブラリを変換先にする手法も考えられるが，本研究では変換前と異なるライブラリへの変換は行わない．本研究では，あくまでライブラリに依存しない変換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．なお，AssertJ への変換は OpenRewrite というツールが提供しており[*3]，提案手法の適用の前にこのツールを適用すれば，ライブラリの都合でリファクタリングできない汎用アサーションは無くなると考えられる．

4.2 評価指標

4.2.1 リファクタリング成功数と検出数

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”*[16] であるため，変換前後のアサーションの等価性が必要である．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

[*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．

[*3] `https://docs.openrewrite.org/recipes/java/testing/hamcrest/migratehamcresttoassertj`, `https://docs.openrewrite.org/recipes/java/testing/hamcrest/migratehamcresttoassertj`

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っている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 プロジェクト内の汎用アサーションを変換と検出の対象とした．

- 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．

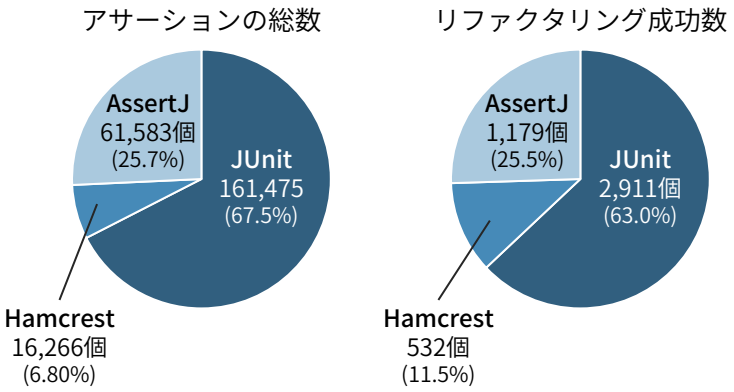

図 4　アサーション総数とリファクタリング成功数の内訳

5　実験結果と考察

5.1　総リファクタリング成功数

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数（リファクタリングが行われた回数）は 4,622 回であった．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，最も多かったリファクタリング成功数は 644 回であった．また，アサーションの総数に対するリファクタリング成功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　ライブラリ別リファクタリング成功数

図 4 はライブラリごとのアサーション総数，リファクタリング成功数の内訳である．使用されているアサーションの数は JUnit が最も多いが，変換先のないアサーションも多いため変換数の割合はアサーション総数の割合より小さい．また，Hamcrest はこの中で最もアサーション総数と変換数の割合の差が大きく，アサーション総数に対する変換数の割合が高いと分かる．

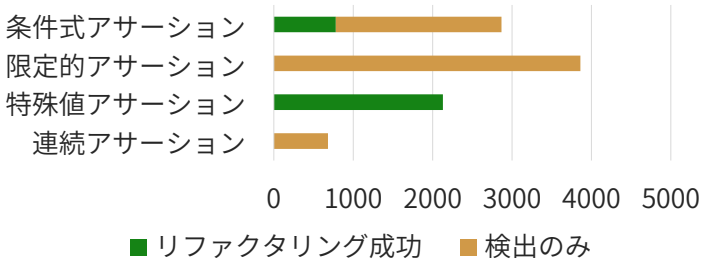

図 5　JUnit に対するリファクタリング成功数と検出数

5.4　汎用アサーション分類別リファクタリング成功数

汎用アサーションの種類ごとの変換数と検出数を各ライブラリごとに説明する．

5.4.1　JUnit のリファクタリング成功数と検出数

図 5 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数が多い．また，リファクタリング成功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と分かる．JUnit のアサーションは機能が限られており，具体的な検証ができる検証用メソッドが存在しないため，全アサーションに対する検出数とリファクタリング成功数の割合も他二つのライブラリより高い．連続アサーションの検出数は条件式アサーションのリファクタリング成功数よりも少なく，他ライブラリよりも，アサーション総数と比較した相対的な数が少ない．

5.4.2　Hamcrest のリファクタリング成功数と検出数

Hamcrest のリファクタリング結果は 図 6 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数は 23 個と少ない．

5.4.3　AssertJ のリファクタリング成功数と検出数

AssertJ のリファクタリング結果を 図 7 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数が多い．一方で条件式アサーションのリファクタリング成功数は 16 個であり，アサーションの合計数が約 1/4 である Hamcrest よりも少ない．

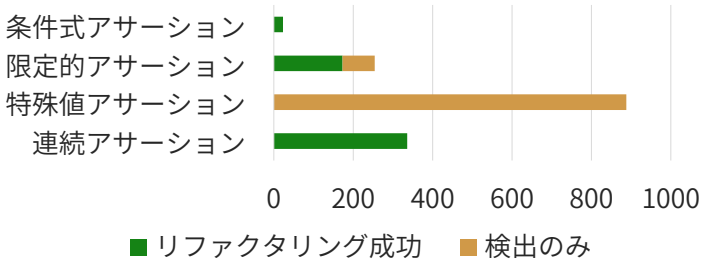

図 6　Hamcrest に対するリファクタリング成功数と検出数

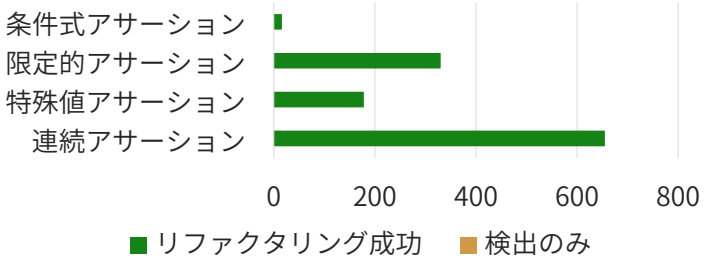

図 7　AssertJ に対するリファクタリング成功数と検出数

5.5　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数が増加すると分かる．

5.6　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 の canal リポジトリ内にある parse プロジェクト[*4]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．

5.7 考察

まず，汎用アサーションの種類ごとの変換数の差について考察する．限定的アサーションの変換数が条件式アサーションや特殊値アサーションに比べて多い理由としては変換先の検証用メソッドが具体性が要因だと考えられる．限定的アサーションの変換先である，要素数を検証する `hasSize()` は java.util.Collection など，要素数の概念を持つ型にのみ使用可能である．それに対し，条件式アサーションの変換先である `isEqualTo()` や特殊値アサーションの変換先である `isTrue()` 等は，その汎用性から使用頻度が高いと考えられる．実際に提案手法を適用した OSS の中で AssertJ のアサーションを使用しているファイルを見ると，`isTrue()` は `hasSize()` の 2 倍以上，`isEqualTo()` に関しては 11 倍以上の数が存在する．使用頻度の高い汎用的なアサーションほど機能が把握されやすいと考えられる．一方，具体的な検証用メソッドは使用できる場面が汎用的な検証用メソッドよりも少なく，把握が困難であると考えられる．そのため，変換先の検証用メソッドがあまり把握されていない限定的アサーションの変換数が多くなると考える．

次に，ライブラリごとの変換数と検出数の差について考察する．JUnit の，アサーションの合計数に対する連続アサーションの検出数は，他のライブラリよりも少ない．これは，JUnit の機能が少なさが原因だと考えられる．同一の実測値に対する複数のアサーションは，主に複数の観点からの検証を行う際に出現する．例えば，AssertJ では同じリストの実測値でも，要素数や個別の要素など複数の観点から検証ができる．それに対し，JUnit でリストに対して複数の観点からの検証を行うためには，`size()` メソッドを呼び出すなど実測値を限定的な値にする必要があるため，リストを実測値にできない．そのため，連続アサーションの対象とならない．

[*4] https://github.com/alibaba/canal/tree/master/parse

6 制約と展望

6.1 条件外とした比較内容のリファクタリング

本稿で分類したアサーションの種類には属しているが，3.4 のリファクタリング条件に当てはまらない比較内容が存在する．このような比較内容に関しても，提案手法と同様の変換を適用し，リファクタリング成功数を計測する必要があると考える．提案手法では OSS の事前調査に基づく一部の比較内容のみをリファクタリング条件としたが，事前調査が十分ではない可能性も考えられる．例えば，`hasSize(0)` は特殊値アサーションとして `isEmpty()` に変換できる．`hasSize()` は，`isEqualTo()` 等の本研究の変換対象となるような検証用メソッドより実験対象中の使用数は少ないが，2,000 回以上と検証用メソッドの種類としては 5 番目の使用数であったため，変換可能な比較内容も他と比べて多いと考える．

6.2 問題箇所限定能力の評価

提案リファクタリングによる問題箇所限定能力の評価には，被験者実験が必要であると考える．汎用アサーションから具体アサーションへの変換は，テスト失敗時の原因特定を容易にするが，変換によってどの程度原因が分かりやすくなるかは本研究では評価していない．そのため，今後はリファクタリング前後のバグ修正にかかる時間差を計測するなどの実験が必要であると考えられる．

6.3 連続アサーションの可読性

連続アサーションに対する変換によって実際に可読性が向上しているかは別途評価が必要であると考える．ユーザーによっては，メソッドチェーンによるアサーションの統合が可読性が低下させると考え，連続アサーションのような記述をしている可能性も考えられる．実際に，Google が提供するアサーションライブラリの Truth は AssertJ と似た記述方法で検証を行えるが，可読性を損なう場合があるとしてメソッドチェーン機能は実装されていない[*5]．メソッドチェーンを用いているテストとそうでないテストを用意し，被験者に可読性を比較させるなどの実験を行い，定性的指標も含めて判断される必要があると考える．

6.4 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

[*5] `https://github.com/google/truth/issues/884`

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．

6.5 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 [17][18]．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[*6]．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[*7]．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．

[*6] `https://hamcrest.org/`

[*7] `https://github.com/assertpy/assertpy`

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，それぞれについて等価かつ具体的なアサーションへの変換を行った．また，実際に OSS のテストコードを対象とした実験を行い，提案手法の効果を確認した．実験の結果，提案手法によって OSS 中の約 1.8% の汎用アサーションを変換できた．

今後の課題として，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．本研究では，実際のプロジェクトに特に多く登場する比較内容に絞って実験を行ったが，条件外となった比較内容の種類も多く，提案手法によって改善できる正確な数が把握できていないのが現状である．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は計測されるべきだと考える．

謝辞

本研究を進める上で，多くの方にご助力を頂きました．

楠本真二教授には，中間報告会で多数の助言を頂きました．客観的な視点からの意見は研究価値の更なる向上につながり，自分にとって非常に貴重なものとなりました．また，煮詰まった時の気分転換など，頂いた差し入れが力になりました．3 回目の中間発表後に頂いた配慮の言葉は，苦しい時期を乗り切る大きな活力となりました．感謝の気持ちと共に，このような気遣いが上手な人間に成長したいという思いが大きくなりました．

柗本真佑准教授には，研究内容や発表の方法，論文の添削に至るまで様々な箇所でご指導を頂きました．至らない点も多く，迷惑をかけてしまうこともありましたが，常に「今後どうすべきか」という未来に向かった議論をしてくださり，研究だけでなく，自分自身の成長にも繋がったと感じています．また，論文の書き方やスライドの作成方法など，今後の仕事などにも活かせるようなスキルも詳細に学ぶことができました．1 年間で吸収できることは限られるかもしれませんが，1 年だけでも柗本准教授からのご指導を受けられたことは非常に貴重な体験だと感じています．心から感謝を申し上げます．

楠本研究室事務補佐員の橋本美砂子様には，事務関連で多大なサポートをしていただきました．出張の手続きや研究室の備品管理等，自分たち学生の負担が最小限になるように気遣いを頂き，研究部分に集中することができました．ハードディスクのデータ削除の際は，削除場所までの送り迎えまでしていただき，サポートの手厚さを特に実感しました．

楠本研究室の先輩方には研究室の習慣や研究内容に至るまで，多くの面でお世話になりました．研究に慣れていない中，研究の経験がある方々の感覚は非常に参考になりました．また，励ましの言葉をもらうことも多くありました．特に，論文執筆で忙しい中，風邪を引いてしまった際に頂いた言葉はとても心に響き，最後まで研究をやり遂げるための大きな力となりました．

研究室の同期の方々には，研究室関連で些細な疑問点でも相談させていただきました．また，研究の進捗を聞くことで自分の研究に対するモチベーションの維持に繋がりました．短い間でしたが，非常に感謝しています．

家族には，金銭面や生活面などで支えていただきました．締め切り直前の日付が変わるような帰宅ができるのは家族の協力あってこそです．

最後に，本研究を支えてくださったすべての方に心より感謝いたします．

参考文献

[1] Dustin, E., Rashka, J. and Paul, J.: *Automated software testing: introduction, management, and performance*, Addison-Wesley Longman Publishing Co., Inc. (1999).

[2] Duvall, P., Matyas, S. and Glover, A.: *Continuous Integration: Improving Software Quality and Reducing Risk*, Pearson Education (2007).

[3] Ebert, C., Gallardo, G., Hernantes, J. and Serrano, N.: DevOps, *IEEE Software*, Vol. 33, No. 3, pp. 94–100 (2016).

[4] Beck, K.: *Test Driven Development: By Example*, Pearson Education (2022).

[5] Koskela, L.: *Effective Unit Testing: A guide for Java developers*, Shelter Island, NY : Manning (2013).

[6] Noback, M.: *The Single Responsibility Principle*, pp. 3–10, Apress (2018).

[7] Aljedaani, W., Peruma, A., Aljohani, A., Alotaibi, M., Mkaouer, M. W., Ouni, A., Newman, C. D., Ghallab, A. and Ludi, S.: Test Smell Detection Tools: A Systematic Mapping Study, in *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] Leotta, M., Cerioli, M., Olianas, D. and Ricca, F.: Fluent vs Basic Assertions in Java: An Empirical Study, in *In Proceedings of International Conference on the Quality of Information and Communications Technology*, pp. 184–192 (2018).

[10] Leotta, M., Cerioli, M., Olianas, D. and Ricca, F.: Hamcrest vs AssertJ: An Empirical Assessment of Tester Productivity, in *In Proceedings of Quality of Information and Communications Technology*, pp. 161–176 (2019).

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

[12] Grano, G., Scalabrino, S., Gall, H. C. and Oliveto, R.: An empirical investigation on the readability of manual and generated test cases, in *In Proceedings of Conference on Program Comprehension*, pp. 348–351 (2018).

[13] Lin, B., Nagy, C., Bavota, G., Marcus, A. and Lanza, M.: On the Quality of Identifiers in Test Code, in *2019 19th International Working Conference on Source Code Analysis and*

Manipulation, pp. 204–215 (2019).

[14] Spadini, D., Palomba, F., Zaidman, A., Bruntink, M. and Bacchelli, A.: On the Relation of Test Smells to Software Code Quality, in *In Proceedings of International Conference on Software Maintenance and Evolution*, pp. 1–12 (2018).

[15] Martin, R. C.: *Clean Code: A Handbook of Agile Software Craftsmanship*, Prentice Hall PTR (2008).

[16] Opdyke, W. F. and Johnson, R. E.: Creating abstract superclasses by refactoring, in *In Proceedings of conference on Computer science*, pp. 66–73 (1993).

[17] JetBrains, : Equality assertion can be simplified, `https://www.jetbrains.com/help/inspectopedia/RsAssertEqual.html` (Accessed at 2025-02-03).

[18] ashleyfrieze, : Lower Level Assertion, `https://codingcraftsman.wordpress.com/2024/08/12/lower-level-assertion/` (Accessed at 2025-02-03).