テストカバレッジに基づくテストコードの再構成パターン

坂本 一憲 * **和田 卓人** † **鷲崎 弘宜** ‡ **深澤 良彰** §
Kazunori Sakamoto Takuto Wada Hironori Washizaki Yoshiaki Fukazawa

1. Name

テストカバレッジに基づくテストコードの再構成パターン

2. Context

近年，テストコードを記述するための開発手法が進歩しており，製品コードに対してテストコードが十分に記述されるようになったが，その一方でテストコードの重複が起こりやすくなっている．重複したテストコードの量が増大することによって，以下で説明するテストの効率性の低下とテストコードの保守性の低下の 2 問題が引き起こされる．

- **テストの効率性の低下**
  テストコードの量があまりに増大しすぎると，テストの実行効率性が低下する．この問題は，テストコードの量の増大に比例して，単純にテストの実施項目数が増大することで，テストの実施時間の増大が引き起こされるためである．テストの効率性の低下は，手動によるテストでも，テストフレームワークによる自動テストにおいても問題となり，単体テストや結合テストの実施が困難になってしまう．
  例えば，開発者が単体テストの実施を渋るようになったり，Hudson のような継続的インテグレーションツールを利用して，テストを自動的に実施する場合でも，バージョン管理システムへのコミット，コンパイル，テストの実施のサイクルに遅れが生じ，正常に実施できなくなる場合がある．特に，テスト駆動開発 [1] やリファクタリングを行う際は，コーディングとともに頻繁にテストを実施する必要があるため，テストの実施時間の増大がコーディング作業の能率低下に直結してしまう．また，テストコードの増大によってテストの実施が困難になると，ソフトウェアの品質向上のためにテストコードを追加することが，より一層テストの実施を困難にさせてしまう．そのため，かえって試験性の低下を招いてしまい，ソフトウェア全体の品質問題を引き起こす場合がある．
- **テストコードの保守性の低下**
  テストコードの量があまりに増大しすぎると，テストコードの保守性が低下する．特に，テストコードもソースコードの一部であるため，ソフトウェア全体の保守性の低下につながる．この問題は，テストコードの量の増大によって，人間がソースコードを読んで理解する作業に要する時間が増加したり，ソースコードを変更した際に影響が波及する範囲を推定することが困難になるなど，品質副特性における解析性や変更性の低下，また前節による試験性の低下から，品質特性における保守性の低下が引き起こされるためである．特に，テストコードの重複は DRY 原則に反するため，保守性の低下が顕著である．テストコードの保守性の低下は，テストコードの管理や修正を困難にする．
  例えば，テストが仕様を十分に検証しているかどうかのチェックや，新しいテストコードを追加したり修正することが困難になるため，テストの品質低下を招いてしまい，ソフトウェア全体の品質問題を引き起こす．

3. Problem

ソフトウェア開発においてソフトウェアテスティング，例えば単体テストや結合テストなどにおける重複したテストコードの量が増大しすぎることがある．特に，テスト駆動開発といったアジャイルな手法を用いることでその傾向が強くなる．

テスト駆動開発とは，失敗するテストコードの記述，テストをパスするような最低限の実装，再構成というサイクルを小さく早く繰り返すことによって，ソフトウェアを実装する開発手法である．テスト駆動開発におけるテストコードは，極めて粒度の小さい簡単なものから記述するため，一般的なテストコードよりもテストコードの量が増大しがちである．また，テスト駆動開発におけるテストコードは，ソフトウェアの品質保証のためではなく，ソフトウェアの実装のために記述されるため，別途，品質保証のためのテストコードが必要となる．なお，テスト駆動開発によって記述されたテストコードを，品質保証のためのテストコードに再利用することは可能だが，品質保証のためのテストコードと重複しやすい．このようにして記述されたテストコードの量は一般のプロジェクトと比較して多くなる傾向があり，規模の小さいプロジェクトであっても，一回の単体テストの実施に 5 分以上かかる場合がある．

また，近年のテストフレームワークでは，単体テストのみならず結合テストも実施可能であり，両方のフェーズのテストコードが入り混じる傾向がある．結合テストでは単

* 早稲田大学 大学院 基幹理工学研究科
kazuu@ruri.waseda.jp
† タワーズ・クエスト 株式会社
takuto.wada@towersquest.jp
‡ 早稲田大学 大学院 基幹理工学研究科
washizaki@waseda.jp
§ 早稲田大学 大学院 基幹理工学研究科
fukazawa@waseda.jp

体テストが対象とするコードを複数組み合わせたテストを実施するため，結合テストのテストコードは単体テストのテストコードを包含するケースが多い．そのため，テストコードの重複が発生しやすい．

4. Forces

テストの実行効率の改善，テストコードの保守性の改善，最小限の改善コスト，機械的な判断の 4 つのフォースが挙げられる．

- **テストの実行効率の改善**
  テストの実行効率を改善し，なるべく短い時間でテストを実施する必要がある．
- **テストコードの保守性の改善**
  テストコードを保守するために，可読性や変更容易性の高いテストコードを記述する必要がある．
- **最小限の改善コスト**
  上記二つの改善を行うために，なるべく金銭や時間などのコストを抑える必要がある．
- **機械的な判断**
  上記二つの改善を行うために，極力機械的な判断を元に安全に改善する必要がある．

例えば，テストの実行効率を改善するために，テストフレームワークの改良や分散技術の利用，物理的な PC パフォーマンスの改善などが考えられる．また，テストコードの保守性を改善するために，人手によるテストコードの再構成などが考えられる．この場合，どれも金銭や時間などのコストがかかる上，それぞれが両方の問題を解決することはできない．他に，テストの実施範囲を狭める方法も考えられる．この場合，テストの実行効率のみを改善することができるが，ソースコードのリファクタリングなどの修正を行った際に，どこに変更の影響が生じているか人手で判断してテストの実施範囲を狭めるため，開発者の意思が介入することによってテストが正常に実施されない可能性がある．

したがって，これらの問題を解決するために，不要なテストコードの削除，特に，テストカバレッジに基づいて削除しても安全であると機械的に判断されたテストコードを削除することによって，上述の 4 つのフォースを満たす解決策を利用する．

5. Solution

テストコード網羅している製品コードの部分をテストカバレッジによって算出する．算出結果からテストコードの包含関係を検出する．包含関係にあるテストコードの中から，開発者が重複したテストコードであると判断したものを削除する．

テストカバレッジにはいくつかのレベルが存在する．例えば，すべてのステートメントにおいて，少なくとも一回以上実行されたステートメントの割合を示すステートメントカバレッジ (C0)，すべての条件分岐において，各分岐先が少なくとも一回以上実行された条件分岐の割合を示すデシジョンカバレッジ (C1)，すべての条件分岐の条件式を構成する論理項において，すべての項が少なくとも一回以上はそれぞれ真と偽の両方の値に評価された論理項の割合を示すコンディションカバレッジ (C2), すべての実行パスにおいて，少なくとも一回以上実行されたパスの割合を示すパスカバレッジ (C∞) がある．

テストカバレッジに基づいて判定されたテストコードの包含関係について定義する．テストコード A，B を実行した際に，任意のカバレッジによって製品コードが網羅されたと判定された要素の集合をそれぞれ S_A, S_B と定義する．例えば，ステートメントカバレッジでは網羅されたステートメントの集合が，デシジョンカバレッジでは網羅された条件分岐の集合が，コンディションカバレッジでは網羅された条件式内の論理項の集合が，パスカバレッジでは網羅された実行パスの集合が S_A, S_B に該当する．このとき，カバレッジに基づいて判定されたテストコードの包含関係は，S_A と S_B の包含関係と等価であると定義する．すなわち，S_B が S_A の部分集合である時，S_A は S_B を包含すると呼び，その際，同様にテストコード A は B を包含すると定義する．この包含関係はテストコード A と B といった二要素間の関係のみならず，テストコード A，B と C といったように複数要素間の関係について定義できる．S_A と S_B の和集合が S_C を包含している場合，テストコード A と B の組み合わせがテストコード C を包含していると定義する．同様にして 4 つ以上のテストコードの包含関係も定義する．

図 1 では，テストコードの包含関係のコンセプトを示す．右側の棒がテストカバレッジにおける製品コード中の測定単位の要素，数字は要素に番号付けしたものである．例えば，ステートメントカバレッジに基づいて判断する場合，1 つの棒が 1 つのステートメントに，数字が行番号に該当する．テストコード A によって網羅された部分はテストコード B によって網羅された部分を全て含んでおり，テストコード A はテストコード B を包含している様子を示す．

テストコードの重複はリファクタリングを行う際の匂い [3] として考えられており，重複したテストコードの削除が推奨されている．そこで，先ほど述べた定義に基づいて包含関係にあるテストコードを機械的に検出して，その上で開発者が重複していると判断したテストコードについて削除する．これによって，開発者が重複しているか判断すべきテストコードを機械的に絞り込むことができる．なお，包含関係にあるテストコードを機械的に検出するためには，各テストコードが網羅する要素の集合をそれぞれ算出して，その集合が包含関係にあるか調べればよい．この処理は，

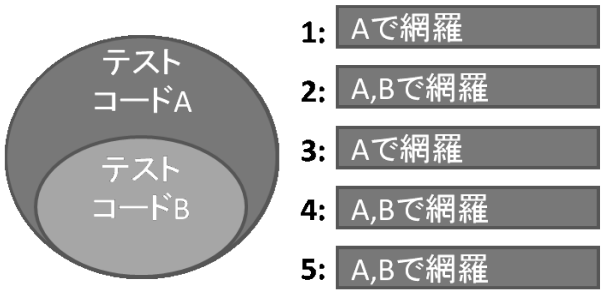

図 1　テストコードの包含

既存のツールやフレームワークで，包含関係を調べるテストコードの単位，例えば各テストケースについて，テストカバレッジと共に網羅する要素の集合を算出することで実現できる．例えば，Open Code Coverage Framework [2] を利用する場合，各テストケースが網羅する要素の集合を出力できるので，その集合について包含関係を調べることで，テストコードの包含関係を検出できる．

以下で，解法の手順を説明する．

1. **網羅する要素集合の算出**
   各テストケースなど，任意のテストコード単位で，網羅する製品コードの要素の集合を算出する．
2. **包含関係にあるテストコードの検出**
   得られた各集合がそれ以外の集合と包含関係にあるかどうか集合論に基づいて計算することで，包含関係にあるテストコードを検出する．
3. **重複テストコードの発見**
   得られた包含関係にあるテストコードの中から，開発者が重複していて不要なテストコードを探す．これは，ブラックボックステストやホワイトボックステストを問わず，片方のテストコードだけで十分テストできると判断した場合，重複していると考える．
4. **重複テストコードの削除**
   重複していると判断されたテストコードを削除する．
5. **テストコード再構成の確認**
   テストを実施して，削除を行う前とテストの結果に変化がないことを確認する．

このように，テストコードの包含関係をテストカバレッジによって判断する仕組みを用いることで，重複したテストコードの候補を機械的に検出して，それに基づいて開発者が不要なテストコードを発見して削除を行える．この解決方法によって，開発者が判断する部分を最小限に抑え，低コストで安全な削除の実施が可能となる．

6. Examples

Java 言語で実装された LRU アルゴリズムを用いたキーと値をマッピングするサンプルプログラムを考える．保持する要素数を LRU アルゴリズムによって 2 個に限定したマップになっており，3 個目の要素追加するタイミングで，使われてから最も長い時間が経過した要素を削除する．

リスト 1 がプログラム本体のソースコードで，リスト 2 がテストコードにあたる．両方のコードはテスト駆動開発の手法にのっとって実装されており，テストには少し冗長なコードも含まれている．例えば，**キー** `abc` **で値** `0` **を設定して_キー** `abc` **で値** `0` **を取得する**テストコードは，設定した値を取得できることを確かめる最低限のテストコードになっているが，**キーと値のペアを** 3 **種類設定すると一つ目のキーのみ値** `null` **を取得する**テストコードも存在する．これは，最低限のテストコードを記述して，そのテストをパスするような最低限の実装を行い，それからさらに厳密なテストコードを記述して，さらにテストをパスするような最低限の実装を行うというテスト駆動開発のサイクルにそってソースコードとテストコードが実装されたためである．この二つのテストコードは明らかに重複関係にあり，後者のテストコードがあれば前者は不要であると考えられる．他にも，**インスタンスを生成する**という極めて簡潔なテストコードも存在する．これは，ソースコードの実装を開始するにあたって，テストフレームワークが正常に動作しているかどうかの確認も込めて記述されている．しかし，明らかにそれ以外のテストコードに包含されているため，このテストコードも不要であると考えられる．

次に，テストカバレッジに基づくテストコードの重複判定の例を示す．**キー** `abc` **で値** `0` **を設定して_キー** `abc` **で値** `0` **を取得する**テストコードに対して，ステートメントカバレッジを測定すると 20,32,33 行目が未実行と出力される．**キーと値のペアを** 3 **種類設定すると一つ目のキーのみ値** `null` **を取得する**テストコードに対して，ステートメントカバレッジを測定すると 20 行目のみが未実行と出力される．したがって，ステートメントカバレッジという測定基準において，後者は前者を包含している．この場合，テストコードを実際に見たうえで，後者のみで十分テストできると判断できる．テストコードの増大によって保守性と効率性の低下が問題になっている場合は，重複している前者のテストコードを削除することで，改善を図ることができる．

このように，テストカバレッジに基づいて機械的に重複関係にあたるテストコードを発見することができ，見つかったテストコードを削除するか否かを開発者が判断をした上で削除を行える．

List 1　LRU を利用したマップの Java ソースコード

```
1 import java.util.HashMap;
2 import java.util.LinkedList;
```

```
3  import java.util.Map;
4  import java.util.Queue;
5
6  public class LruCache<TKey, TValue> {
7
8    private Map<TKey, TValue> _map;
9    private Queue<TKey> _orderedKeys;
10
11   public LruCache() {
12     _map = new HashMap<TKey, TValue>();
13     _orderedKeys = new LinkedList<TKey>();
14   }
15   public TValue get(TKey key) {
16     return _map.get(key);
17   }
18   public void put(TKey key, TValue value) {
19     if (_map.containsKey(key)) {
20       _orderedKeys.remove(key);
21     }
22     else {
23       updateCache();
24     }
25
26     _map.put(key, value);
27     _orderedKeys.add(key);
28   }
29   private void updateCache() {
30     // サイズ2のLRUリスト
31     if (_orderedKeys.size() == 2) {
32       TKey key = _orderedKeys.poll();
33       _map.remove(key);
34     }
35   }
36 }
```

List 2　LRU を利用したマップの Java テストコード

```
1  import static org.junit.Assert.*;
2  import org.junit.Test;
3
4  public class LruCacheTest {
5    @Test
6    public void インスタンスを生成する() {
7      // テストが正常に実施できることを確認
8      assertNotNull(new LruCache<String,Integer>());
9    }
10   @Test
11   public void 空の状態から値nullを取得する() {
12     LruCache<String, Integer> cache =
13       new LruCache<String, Integer>();
14     assertNull(cache.get("a"));
15   }
16   @Test
17   public void キーabcで値0を設定する() {
18     LruCache<String, Integer> cache =
19       new LruCache<String, Integer>();
20     // 例外が発生しないことを確認
21     cache.put("abc", 0);
22   }
23   @Test
24   public void キーabcで値0を設定して_
25       キーabcで値0を取得する() {
26     LruCache<String, Integer> cache =
27       new LruCache<String, Integer>();
28     cache.put("abc", 0);
29     assertEquals(0, (int)cache.get("abc"));
30   }
31   @Test
32   public void キーxyzで値1を設定して_
33       キーabcで値nullを取得する() {
34     LruCache<String, Integer> cache =
35       new LruCache<String, Integer>();
36     cache.put("xyz", 1);
37     assertEquals(null, cache.get("abc"));
38   }
39   @Test
40   public void キーabcで値0_キーxyzで値1を設定して_
41       キーabcで値0_キーxyzで値1を取得する() {
42     LruCache<String, Integer> cache =
43       new LruCache<String, Integer>();
44     cache.put("abc", 0);
45     cache.put("xyz", 1);
46     assertEquals(0, (int)cache.get("abc"));
47     assertEquals(1, (int)cache.get("xyz"));
48   }
49   @Test
50   public void キーと値のペアを3種類設定すると
51       一つ目のキーのみ値nullを取得する() {
52     LruCache<String, Integer> cache =
53       new LruCache<String, Integer>();
54     cache.put("abc", 0);
55     cache.put("xyz", 1);
56     cache.put("def", 2);
57     assertNull(cache.get("abc"));
58     assertEquals(1, (int)cache.get("xyz"));
59     assertEquals(2, (int)cache.get("def"));
60   }
61 }
```

7. Consequences

テストカバレッジに基づいて包含関係にあるテストコードを検出することで，機械的に削除候補である重複テストコードの絞り込みを行うことができる．開発者が包含関係にあるテストコードの中から重複テストコードを発見することで，人手による判断を極力減らして，効率よく安全にテストコードを削除できる．このようにして，重複したテストコードを削除することで，テストコードの増大によるテストの効率性の低下と，ソフトウェアの保守性の低下を改善することができる．さらに，テストコードにおける DRY 原則を遵守できる．

以上から，提案するパターンによって，最小限のコストで重複テストコードの削除を達成する．

8. Known Uses

.NET ライブラリの Paraiba [4] は，本パターンによる重複テストコードの削除により，テスト実施時間の大幅削減に成功している．

Paraiba は GitHub にて管理されているオープンソースプロジェクトで，リビジョン番号が 6c5a74cc832a865969b6c30883b000390753d8de のコードに対して本パターンが適用されて，リビジョン番号が 125da5dacd0d16b4546b5c63052a34869c5d846d のコードが得られている．

Paraiba では MergeSort のテストコードとして，長さ 0 から 11 までの添え字と値が一致しているようなリストに対して，全ての順列を求めてソーティングを行うテストケース 12 個と，特定の値を保持するリストに対するテストケース 14 個が存在していた．ステートメントカバレッジとデシジョンカバレッジに基づいて重複するテストコードを判断して削除することで，本パターンの適用後のテストケースは 14 個にまで削減され，テストの実施時間は 2 分程度かかっていたものが，1 秒程度まで削減されている．

この事例から本パターンの有用性が示されている．

9. Related Patterns

Template Method Pattern は，ある親クラスに対して子クラスの細部の処理は異なっていても，大まかな処理の流れが同じ場合に適用されるパターンである．この時，どの子クラスも処理の大まかな流れは同じであるため，それぞれの子クラスが全ての処理を記述するのではなく，親クラ

スで処理の大まかな流れを共通処理として括り出し，それ以外の部分を抽象メソッドとして定義しておく．子クラスでは抽象メソッドをオーバーライドすることによって，それぞれ異なる細部の処理を記述する．したがって，Template Method Pattern を適用することで，それぞれの子クラスでの重複処理が削除され，DRY の原則が遵守される．この点で，本パターンは Template Method Pattern と関連性が強い．

参考文献

[1] Kent Beck, ”Test-Driven Development: By Example”, The Addison-Wesley Signature Series, 2003.

[2] Kazunori Sakamoto, Open Code Coverage Framework, http://sourceforge.jp/projects/codecoverage/.

[3] A. van Deursen, L. Moonen, A. van den Bergh, and G. Kok, “ Refactoring Test Code ”, Extreme Programming Perspectives, M. Marchesi, ed., pp.92-95, 2001.

[4] Kazunori Sakamoto, Paraiba, http://github.com/KAZUu/Paraiba.