Vol.2015-SE-187 No.34
2015/3/13

テストカバレッジを用いた新しいテスト駆動開発プロセスの提案

坂本 一憲[1,a)]　土肥 拓生[1]

概要：テスト駆動開発は製品コードを記述する前にテストコードを記述することで，品質の高いコードの記述を実現する．従来のテスト駆動開発のプロセスには，テストコードの要件を満たすように製品コードを記述するというステップが含まれているが，意図せずテストが実行しない箇所を含む製品コードを記述するケースが起こりうる．
本論文では，記述した製品コードのテストカバレッジを 100%に維持するという制約を加えることで，より厳格なテスト駆動開発のプロセスを提案する．被験者実験を実施して，意図せずテストが実行しない箇所を含む製品コードを記述することがあること，また，製品コードのテストカバレッジを 100%に維持しながら製品コードを記述できることを確認した．

キーワード：テスト駆動開発，テストファースト，テストカバレッジ

1. はじめに

一般的なコーディングでは，開発するソフトウェアの実装となるコード（製品コード）を記述してから，製品コードを自動テストで検証するためのコード（テストコード）を記述するという順序を取る．製品コードの記述は，ソフトウェアを実装する直接的な活動であるが，一方で，テストコードの記述は，それ自体がソフトウェアを実装する活動ではない．そのため，開発者はテストコードの記述を退屈な活動だと捉えがちである [2]．その結果，開発者は製品コードを記述しても，それに対するテストコードを記述しないケースが多い．

テスト駆動開発はエクストリーム・プログラミング（XP）において推奨されている開発手法である [1]．一般的なコーティングの手順とは異なり，テスト駆動開発はテストファースト，つまり，テストコードを記述してから，製品コードを記述するという順序を取る．テスト駆動開発では，ソフトウェアが満たすべき要件をテストコードで記述してから，記述したテストコードのテストが成功するように製品コードを記述する．その結果，テストコードを記述しないケースを避けることができる上，製品コードの記述において達成すべき目標が明確化されるというメリットがある．

[1] 国立情報学研究所
National Institute of Informatics, Chiyoda, Tokyo 101–8430, Japan
[a)] exkazuu@nii.ac.jp

既に多くの研究でテスト駆動開発のメリットについて議論および評価されており，特に，テストの記述により製品コードの品質が向上することが確認されている．[3–7]．しかし，テスト駆動開発は従来のコーディング手順とは大きく異なるため，テスト駆動開発を導入するにあたり，手順の違いを意識する必要がある．本論文では，開発者が意図せずテストファーストの手順を取らない問題，つまり，テストコードが検証できない箇所を含む製品コードを記述してしまう問題を扱う．テストファーストの手順を取らなければ，テスト駆動開発のプロセスに逸脱するため，その結果，テスト駆動開発のメリットを享受できないと著者らは考える．

本論文では，開発者が意図せずテストファーストの手順を取らない問題に対処するため，製品コードを記述してからテストコードを実行する際に，記述した製品コードのテストカバレッジが 100%であることを確認するステップを追加した，新しいテスト駆動開発のプロセスを提案する．提案プロセスを利用することで，記述した製品コードがテストで実行されることを確認するため，テストファーストを完全に遵守しながら開発を進めることができる．被験者実験を通して，本論文が取り扱う問題が実際におこるかどうかを確認して，また，テストカバレッジを 100%に維持した状態でコーディングできることを確認した．

本論文の研究課題（Research Questions, RQs）は以下のとおりである．

RQ1 開発者は意図せずテストで実行されない箇所を有する製品コードを記述することがあるか？

RQ2 提案プロセスを用いて製品コードのテストカバレッジを 100%に維持した状態でコーディングできるか？

本論文の貢献点は以下のとおりである．

- 開発者が意図せずテストで実行されない箇所を有する製品コードを記述することを防ぐために，記述した製品コードのテストカバレッジを 100%に維持する新しいテスト駆動開発のプロセスを提案した点．
- 被験者実験を通して，開発者が意図せずテストで実行されない箇所を有する製品コードを記述するケースを確認した点．
- 被験者実験を通して，提案プロセスを用いてテストカバレッジを 100%に維持した状態でコーディングできることを確認した点．

2. 意図せずテスト駆動開発のプロセスから逸脱する例

本節では FizzBuzz 問題を取り上げて，意図せずテスト駆動開発のプロセスから逸脱する例を説明する．

本論文では，従来のテスト駆動開発のプロセスを以下のステップで定義する．

(1) 仕様を反映したテストケースをテストコードとして記述する．
(2) テストを実施して，テストが失敗することを確認する．
(3) テストの成功に必要かつ最もシンプルな製品コードを記述する．
(4) テスト実施して，テストが成功することを確認する．
(5) 必要に応じて製品コードおよびテストコードをリファクタリングする．
(6) テスト実施して，テストが成功することを確認する．
(7) 実装が完了していなければ，ステップ 1 に戻る．

FizzBuzz 問題とは，ユーザが入力した 1 以上の整数に対して，特定のルールに従って文字列を出力するプログラムの仕様を指す．入力に対する出力の関係を定めるルールは以下のとおりである．なお，出力は以下のいずれかであり，上の項目が優先的に適用される．

15 の倍数の入力に対して，FizzBuzz と出力する．
3 の倍数の入力に対して，Fizz と出力する．
5 の倍数の入力に対して，Buzz と出力する．
上記以外の数に対して，その数値の文字列表現を出力する．

図 1 で，整数 1 と 15 を入力するテストコードを示す．図 1 のテストコードに対して，従来のテスト駆動開発のプロセスで，テストが成功するように記述した製品コードを図 2 で示す．

図 2 に記載されている if 文の条件式は，図 1 のテストコードがテストで実行しない箇所を含む．図 2 の fizzBuzz メソッドに整数 15 を入力すると条件式を構成する各項の評価値はどちらも真になるが，整数 1 を入力すると最初の項の評価値が偽となり，2 つ目の項が評価されない．したがって，図 2 の製品コードのコンディションカバレッジ

```
1  class FizzBuzzTest {
2    @Test void pass_1() {
3      assertThat(FizzBuzz.fizzBuzz(1), is("1"));
4    }
5
6    @Test void pass_15() {
7      assertThat(FizzBuzz.fizzBuzz(15),
8          is("FizzBuzz"));
9    }
10 }
```

図 1 整数 1 と 15 を入力するテストコード

```
1  class FizzBuzz {
2    String fizzBuzz(int i) {
3      if (i % 3 == 0 && i % 5 == 0) {
4        return "FizzBuzz"
5      }
6      return "" + i;
7    }
8  }
```

図 2 従来のテスト駆動開発で記述した整数 1 と 15 を正しく処理する製品コード

```
1  class FizzBuzz {
2    String fizzBuzz(int i) {
3      if (i % 15 == 0) {
4        return "FizzBuzz"
5      }
6      return "" + i;
7    }
8  }
```

図 3 テストカバレッジが 100%である整数 1 と 15 を正しく処理する製品コード

```
1  class FizzBuzz {
2    String fizzBuzz(int i) {
3      if (i % 3 == 0) {
4        if (i % 5 == 0) {
5          return "FizzBuzz"
6        }
7        return "Fizz"
8      } else {
9        return "Buzz"
10     }
11     return "" + i;
12   }
13 }
```

図 4 完成した FizzBuzz 問題の製品コード

（条件網羅）およびコンディションデシジョンカバレッジ（条件分岐網羅）は 100%を満たさず，製品コード中にテストによって実行されない箇所が存在する．

図 3 で，図 1 のテストコードが全ての箇所を実行する製品コードを示す．図 3 の `fizzBuzz` メソッドでは，入力した整数が 15 の倍数かどうかを最もシンプルに判定している．整数 15 と 1 を入力することで，条件式を構成する各項は真と偽の両方で評価されるため，コンディションカバレッジおよびコンディションデシジョンカバレッジは 100%を満たす．したがって，図 3 の製品コード中に，テストによって実行されない箇所が存在しないことが分かる．

図 4 で完成した製品コードを示す．開発者は FizzBuzz 問題が 3 と 5 の倍数に対して特殊な処理を行うという要件を意識してしまったり，図 4 の完成形が早い段階で思いついたりするケースがあると著者らは考える．その結果，図 1 のテストコードが検証可能な範囲を超えて，意図せず図 2 のような製品コードを記述してしまうケースがあると考える．本論文では，以上のようなテストコードが検証できる範囲を超えるように，意図せず製品コードを記述してしまう問題を取り扱う．

3. テストカバレッジを用いたテスト駆動開発プロセスの改善

前節の例を通して，テストカバレッジを測定することで，テストによって実行されない箇所が存在するかどうかを判定できる例を示した．テストカバレッジは製品コード全体のうち，テストコードによって実行された製品コードの割合を示すため，記述した製品コードのテストカバレッジが 100%でない場合は，必ずテストされていない箇所が存在する．なお，テストカバレッジにはいくつかの測定基準があり，多くの基準では測定値が 100%であっても，テストされていない箇所が存在するケースがある．

以上をふまえて，テストを実行する際にテストカバレッジを測定して，テストされていない箇所が存在した場合は，当該箇所を削除する新しいテスト駆動開発のプロセスを提案する．提案するテスト駆動開発のプロセスを以下のステップで定義する．

(1) 仕様を反映したテストケースをテストコードとして記述する．
(2) テストを実施して，テストが失敗することを確認する．
(3) テストの成功に必要かつ最もシンプルな製品コードを記述する．
(4) テスト実施して，テストが成功することを確認する．
(5) テストが成功した状態で，記述した製品コードのテストカバレッジが 100%になるように，製品コードを修正する．
(6) 必要に応じて製品コードおよびテストコードをリファクタリングする．
(7) テスト実施して，テストが成功することを確認する．
(8) テストが成功した状態で，記述した製品コードのテストカバレッジが 100%になるように，製品コードを修正する．
(9) 実装が完了していなければ，ステップ 1 に戻る．

5 と 9 ステップ目にテストカバレッジが 100%になるような製品コードの修正作業を追加した．本作業では，意図せずテストの範囲を超えて記述してしまった製品コードを削除するために，コメントアウトをしたり，もしくは，条件分岐の条件式を書き換える作業を行う．

提案プロセスを利用することで，製品コードの記述およびリファクタリングを行った際に，テストカバレッジが 100%になることを保証できる．したがって，記述した製品コードは常にテストで実行されるため，テストファーストを完全に遵守しながら開発を進めることができる．

4. 評価

本節では，テスト駆動開発で製品コードを記述する際に，テストカバレッジが 100%にならないケースが起こりうるかどうか（RQ1）と，提案した新しいテスト駆動開発のプロセスを用いて，テストカバレッジが 100%である状態を維持できるかどうか（RQ2）を評価する．

4.1 実験設定

RQ1 および 2 に回答するために，コンピュータサイエンスを専攻する修士学生 2 名を被験者として，従来および提案するテスト駆動開発を用いて，Java 言語でソフトウェアを開発する実験を行った．被験者はソフトウェアテストおよびテスト駆動開発の知識および経験を有している．

開発対象のソフトウェアとしてライフゲームを選んだ [8]．ライフゲームは古典的なシミュレーションゲームである．2 次元の格子状マップにセルが敷き詰められており，各セルは生存もしくは死亡の 2 種類の状態のうちどちらかの状態にある．離散的な単位時間が経過するにつれ，誕生，生存，過疎，過密という 4 つのルールによって，各セルの状態が変化する様子をシミュレーションする．

被検者がライフゲームの本質的な実装に集中できるように，著者らがライフゲームの GUI ビューアや，セルの状態を管理するクラスを含んだ半完成のソースコードを提供した．[*1]．

4.2 実験手順

被験者 2 名を A と B として，それぞれ従来および提案するテスト駆動開発のプロセスを用いて開発した．なお，開発の経過を記録するために，被検者に Git リポジトリを提供して，以下の手順で示すタイミングでコミットをして

[*1] 提供したコードは `https://github.com/exKAZUu/GameOfLifeForTDD` で公開している．

もらった．

以下で従来のテスト駆動開発のプロセスを用いる被験者 A の手順を示す．

(1) 仕様を反映したテストケースをテストコードとして記述する．
(2) テストを実施して，テストが失敗することを確認する．
(3) テストの成功に必要かつ最もシンプルな製品コードを記述する．
(4) テスト実施して，テストが成功することを確認する．
(5) impl というメッセージでコミットする（実験固有の手順）．
(6) 必要に応じて製品コードおよびテストコードをリファクタリングする．
(7) テスト実施して，テストが成功することを確認する．
(8) refactor というメッセージでコミットする（実験固有の手順）．
(9) 実装が完了していなければ，ステップ 1 に戻る．

以下で提案するテスト駆動開発のプロセスを用いる被験者 B の手順を示す．なお，テストカバレッジの測定基準として，ステートメントカバレッジおよびコンディションデシジョンカバレッジを用いた．

(1) 仕様を反映したテストケースをテストコードとして記述する．
(2) テストを実施して，テストが失敗することを確認する．
(3) テストの成功に必要かつ最もシンプルな製品コードを記述する．
(4) テスト実施して，テストが成功することを確認する．
(5) テストが成功した状態で，記述した製品コードのテストカバレッジが 100%になるように，製品コードを修正する．
(6) impl というメッセージでコミットする（実験固有の手順）．
(7) 必要に応じて製品コードおよびテストコードをリファクタリングする．
(8) テスト実施して，テストが成功することを確認する．
(9) テストが成功した状態で，記述した製品コードのテストカバレッジが 100%になるように，製品コードを修正する．
(10) refactor というメッセージでコミットする（実験固有の手順）．
(11) 実装が完了していなければ，ステップ 1 に戻る．

4.3 実験結果

表 1 と 2 で被験者 A と B のコミットログをそれぞれ示す．コミットの回数は何回目のコミットかを，コミットの種類は実装（コミットメッセージが impl）かリファクタリングか（コミットメッセージが refactor）を，テスト未実行箇所はステートメントカバレッジおよびコンディションデシジョンカバレッジの基準において，テストが実行しない製品コードの箇所数をそれぞれ示す．

表 1 と 2 から読み取れるように，既存のテスト駆動開発ではテストされない箇所が存在したのに対して，提案するテスト駆動開発ではテストされない箇所が存在しなかった．

4.4 考察

本節では実験結果をふまえて，RQ1 および 2 への回答を説明する．

既存のテスト駆動開発を用いた被験者 A の実験結果では，テストが実行されない箇所を含む製品コードが記述されていた．ただし，開発途中にのみそのような箇所が存在しており，開発終了時点では全ての箇所がテストで実行されていた．

```
1  int w = _field.getWidth();
2  int h = _field.getHeight();
3  for (int y = 0; y < h; y++) {
4    for (int x = 0; x < w; x++) {
5      boolean isLiving = _field.isLiving(x, y);
6      int c = countLivingNeighborCells(x, y);
7      if (!isLiving && c == 3)
8        birth(x, y);
9      if (isLiving && c >= 2 && c <= 3)
10       living(x, y);
11   }
12 }
```

図 5 被験者 A が記述したテストで実行されない箇所を有する製品コードの抜粋

図 5 で被験者 A が記述したテストで実行されない箇所を有する製品コードの抜粋を示す．2 種類のテストケースが存在しており，テストでは図 5 の 6 行目の変数 c の値が 3 もしくは 4 となる．その結果，9 行目の条件式において，必ず $C >= 2$ の式が真と評価され，偽となるケースがテストで実行されない．以上から，RQ1 に対して，開発途中においてテストが実行されない箇所を含む製品コードが記述されたと回答する．

一方，提案するテスト駆動開発を用いた被験者 B の実験結果では，テストが実行されない箇所を含む製品コードが記述されなかった．つまり，コミットした時点のソースコードのテストカバレッジは常に 100%であった．以上から，RQ2 に対して，提案プロセスを用いて製品コードのテストカバレッジを 100%に維持した状態でコーディングできたと回答する．

5. おわりに

本論文では，従来のテスト駆動開発のプロセスにおいて，テストで実行されない箇所を有する製品コードが記述されるケースがあることについて，FizzBuzz 問題を例に挙げ

表 1 被験者 A のコミットログ

コミットの回数	1	2	3	4	5	6
コミットの種類	実装	実装	実装	リファクタリング	実装	リファクタリング
テスト未実行箇所	0	1	1	1	0	0

表 2 被験者 B のコミットログ

コミットの回数	1	2	3	4	5	6	7	8
コミットの種類	実装	リファクタリング	実装	実装	実装	リファクタリング	リファクタリング	リファクタリング
テスト未実行箇所	0	0	0	0	0	0	0	0

て問題提起した．記述した製品コードのテストカバレッジを 100%に維持するようなステップを加えることで，問題を解決できるような新しいテスト駆動開発のプロセスを提案した．テストで実行されない箇所を有する製品コードが記述されるケースがあるかどうか（RQ1），記述した製品コードのテストカバレッジを 100%に維持できるかどうか（RQ2）を明らかにするために被験者実験を実施した．その結果，テストで実行されない箇所を有する製品コードが記述されるケース，および，記述した製品コードのテストカバレッジを 100%に維持できるケースを確認した．

本論文では，テストで実行されない箇所を有する製品コードが記述されるケースが引き起こす問題について，評価を行っていない．今後は，問題の重大さを検証するための被験者実験を実施する予定である．また，提案したテスト駆動開発プロセスの遵守を支援するために，機械的にテストカバレッジを測定して，100%を維持しているか確認するツールを開発する予定である．

謝辞 本実験にご協力頂いた時武 佑太氏，矢藤 康祐氏に深く感謝する．

参考文献

[1] Beck, Kent：Test-driven development: by example, *Addison-Wesley Professional*, (2003).
[2] Shah, Hina and Harrold, Mary Jean：Studying Human and Social Aspects of Testing in a Service-based Software Company: Case Study, *Proceedings of the 2010 ICSE Workshop on Cooperative and Human Aspects of Software Engineering*, pp. 102–108 (2010).
[3] Williams, Laurie and Maximilien, E Michael and Vouk, Mladen：Test-driven development as a defect-reduction practice, *Software Reliability Engineering, 2003. ISSRE 2003. 14th International Symposium on*, pp. 34–45 (2003).
[4] Desai, Chetan and Janzen, David and Savage, Kyle：A Survey of Evidence for Test-driven Development in Academia, *SIGCSE Bull*, Volume 40, Issue 2, pp. 97–101 (2008).
[5] Maximilien, E.M. and Williams, L.：A structured experiment of test-driven development, *Information and Software Technology*, Volume 46, Issue 5, pp. 337–342 (2004).
[6] Maximilien, E.M. and Williams, L.：Assessing test-driven development at IBM, *Software Engineering, 2003. Proceedings. 25th International Conference on*, pp. 564–569 (2003).
[7] Hussan Munir and Misagh Moayyed and Kai Petersen：Considering rigor and relevance when evaluating test driven development: A systematic review, *Information and Software Technology*, Volume 56, Issue 4, pp. 375–394 (2014).
[8] Conway, John：The game of life, *Scientific American*, Volume 223, Issue 4, pp. 4 (1970).