テスト駆動開発における実装工程の継続的な支援を目的としたフレームワークの提案

テスト駆動開発における実装工程の継続的な支援を目的としたフレームワークの提案

宮下　丈明 a)・片山　徹郎 b)

Proposal of a Framework for Continuous Support of the Implementation Step in TDD

Takeaki MIYASHITA, Tetsuro KATAYAMA

Abstract

TDD is a development methodology that brings us closer to better implementation and testing by repeating a series of steps: test design, implementation that satisfies the tests, and refactoring. In TDD, tests are executed too often, which is expected to improve software quality and find bugs early. On the other hand, one of the disadvantages of TDD is that too much time is wasted when test failures are repeated. This paper proposes a framework aimed at supporting the implementation steps in TDD. The proposed framework generates source code that passes tests while retaining refactoring by the developer. The prototyped framework reduced the time required for the implementation process by 94.22% and the generation time by 66.17% compared to manual work.

Keywords:　TDD, Boundary Value Analysis, syntax analysis, automatic generation

1.　はじめに

近年、ソフトウェアの活用範囲の拡大と複雑化に伴い、ソフトウェア品質と開発効率の重要性が注目されている[1]。

ソフトウェア品質を高めるためのソフトウェア開発手法の 1 つに、テスト駆動開発（TDD）がある。TDD において開発者は、既存のソースコードではパスできないテストの設計、テストをパスする最低限のソースコードの実装、実装したソースコードのリファクタリング、の一連の工程を繰り返すことで、テストケースと実装をより良いものにしていく。TDD では、テストが頻繁に実行されるため、ソフトウェア品質の向上やバグの早期発見が見込める。一方、TDD のデメリットとして、テストの失敗が繰り返された場合に、時間を浪費しすぎることがある[2]。

近年、ソフトウェアの開発効率の向上を目的として、UML や自然言語からソースコードを自動生成する研究が多く報告されている[3,4]。これらの研究は、実装時間の短縮が期待できる。しかし、これらの研究の多くは一回限りの生成にのみ焦点を当てており、継続的な開発やリファクタリングを対象としていない。

本研究では、TDD における実装工程を継続的に支援することを目的としたフレームワークを提案する。提案するフレームワークは、テストコードによるテストにパスできない元のソースコードを、テストにパスするように修正することによって、新しいソースコードを自動的に生成する。また、生成したソースコードに対するリファクタリングを保持し、次の開発サイクルに反映することによって、継続的な開発に対応する。

a)工学専攻機械・情報系コース大学院生

b)工学教育研究部教授

2.　提案するフレームワーク

本章では、提案するフレームワークの構造と振舞いを紹介する。フレームワークの構造を図 1 に、フレームワークの振舞いを図 2 に、それぞれ示す。図 2 は、各サイクルにおけるフレームワークの振舞いを示しており、赤はテストコード、青は開発者によるリファクタリング、緑はフレームワークの振舞いを表す。

提案するフレームワークは、テストコードと、テストコードによるテストにパスできないソースコード S_{old} を入力とし、テストにパスできるソースコード S_{gen} を生成する。その後、1 つ前のサイクルにおけるリファクタリングを S_{gen} に統合することで、テストにパスし、かつ、1 つ前のサイクルにおけるリファクタリングを保持したソースコード S_{new} を出力する。その後、開発者は、S_{new} のレビューとリファクタリングを行い、次のサイクルに移る。

以降、図 1 に示す 5 つの処理部について、それぞれの振舞いを説明する。

2.1 テストコード解析部

テストコード解析部は、入力されたテストコードを解析し、各テストケースについて、テストケースデータを抽出する。テストケースデータは、テスト対象のクラス名とメンバ関数名、テスト対象のメンバ関数に与える引数、与えられた引数に対するテスト対象の関数の期待出力の 4 つの要素で構成する。テストコード解析部の振舞いを以下に示す。

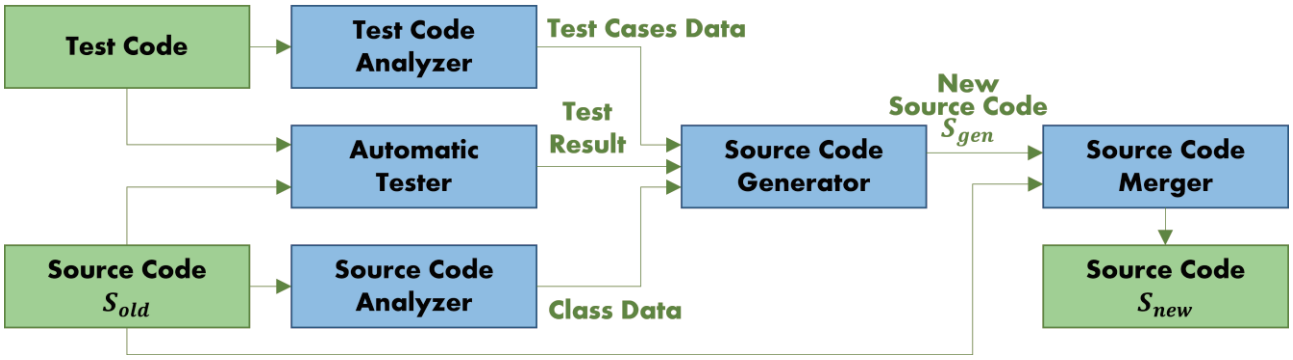

図 1. 提案するフレームワークの構造.

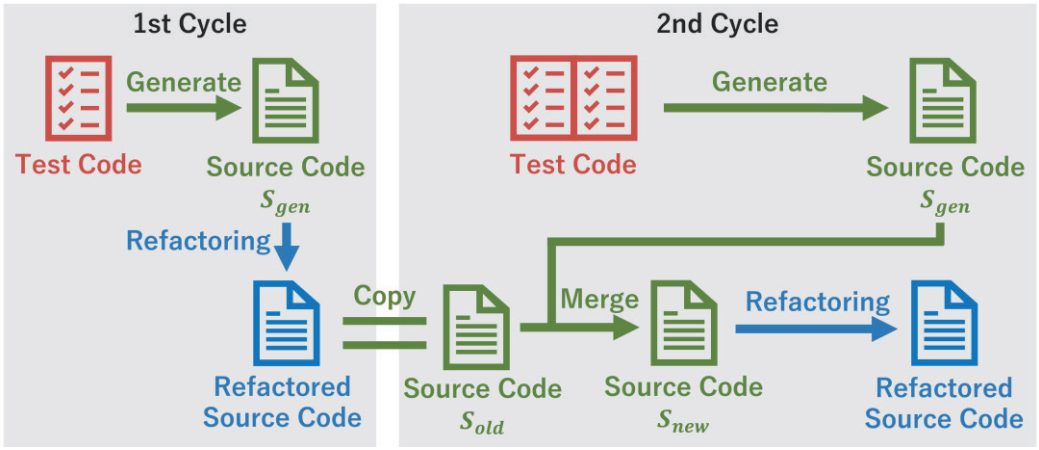

図 2. 提案するフレームワークの振舞い.

1. テストコード解析部は、テストコードを解析し、抽象構文木を生成する。
2. 生成した抽象構文木からアサーションを探索し、アサーションを含む関数を特定する。
3. 関数内の各変数から型と値を抽出し、変数テーブルを作成する。値は、リテラル値か、関数と関数に与えられた引数の値の組を意味する。変数に関数呼び出しの戻り値が代入されている場合は、関数と関数に与えられた引数の値の組を変数の値として変数テーブルに記録する。
4. アサーションで使用される値については、変数テーブルから型と値を取得する。値が関数と引数の組である場合、その関数をテストケースの対象として扱う。
5. 抽出したテストケースデータをソースコード生成部（2.4 節に後述）に渡す。

2.2 ソースコード解析部

ソースコード解析部は、既存のソースコード S_{old} を解析し、クラスデータを抽出する。具体的には、構文解析を行い、メンバ関数の引数の名前と型を抽出する。このクラスデータは、ソースコード生成部（2.4 節に後述）に渡す。ソースコード S_{old} が存在しない場合、ソースコード解析部はデータを渡さない。

2.3 自動テスト部

自動テスト部は、自動的にテストを実行し、テスト結果から失敗したテストケースのデータを抽出する。具体的には、テストケースの名前と、テストケースの対象となるメンバ関数の名前である。また、テスト実行時のエラーやテスト前のコンパイルエラーが発生した場合、エラーメッセージを記録する。抽出したテストケースデータと記録したエラーメッセージは、ソースコード生成部（2.4 節で後述）に渡す。既存のソースコードがすべてのテストをパスした場合は、空のテストケースデータをソースコード生成部に渡す。

2.4 ソースコード生成部

ソースコード生成部は、テストコード解析部、ソースコード解析部、自動テスト部から受け取ったデータをもとに、テストケースを満たすソースコード S_{gen} を生成する。ソースコード生成器は、既存のソースコード S_{old} がテストにパスできなかった場合にのみ動作する。ソースコード S_{old} がテストをパスし、自動テスト部からの失敗したテストケースデータが空だった場合は、処理を終了する。

2.4.1 入力データの統合

ソースコード生成部は、テストコード解析部から受け取ったテストケースデータのうち、自動テスト部から受け取った失敗したテストケースデータの関数名と一致するテストケースを抽出する。また、失敗したテストケースの関数を対象とするテストケースデータも同様に抽出する。ソースコード解析部から受信したクラスデータに、失敗したテストケースの対象となる関数を持つクラスのデータがある場合、このクラスデータを抽出する。

2.4.2 境界値分析に基づく期待出力の推定

ソースコード生成部は、テストケースが作成されていない入力に対する期待出力を、境界値分析をもとに推定する。境界値分析は、境界値を中心としたテストケースを使用するテスト設計手法である [5]。テストケースが境界値分析に基づいて設計されている場合、出力が異なる境界値前後の引数についてのテストケースが存在する。テストケースが作られていない引数やメンバ変数による期待出力は、引数やメンバ変数の値が近いテストケースの期待出力と同じ値である可能性が高いと考えられる。ソースコード生成部は、抽出したテストケースデータの引数やメンバ変数をもとにテストケースをソートする。テストケースがない入力に対する期待出力は、入力の値が近いテストケースの期待出力と同じ値を用いる。

2.4.3 テストケースを満たすソースコード生成

ソースコード生成部は、クラス、メンバ変数、メンバ関数、if ブロックの 4 種の中間データを生成する。そして、それらのデータを再帰的にソースコードに変換する。

クラスの中間データは、クラス名、メンバ変数の中間データ、メンバ関数の中間データを持つ。クラス名は、2.4.1 節で抽出したテストケースデータから参照する。メンバ変数の中間データは、2.4.1 節で抽出したクラスデータから取得した変数名と型を持つ。メンバ関数の中間データは、関数名、戻り値の型、複数の if ブロックの中間データを持つ。関数名と戻り値の型は、2.4.1 節で抽出したテストケースデータから参照する。if ブロックの中間データは、2.4.2 節で求めた境界値を条件式とする if 文と、期待出力を戻り値とする return 文を持つ。

ソースコード生成部は、クラスの中間データを再帰的にソースコードに変換することで、テストケースを満たすソースコード S_{gen} を生成する。生成したソースコード S_{gen} は、ソースコード統合部（2.5 節に後述）に渡す。

2.5 ソースコード統合部

ソースコード統合部は、前サイクルのリファクタリングデータとソースコード生成部で生成したソースコード S_{gen} を統合し、既存のソースコードを上書きする。

2.5.1 リファクタリングデータの統合

ソースコード統合部は、ソースコード生成部で生成したソースコード S_{gen} に既存のソースコード S_{old} を統合し、テストをパスできる、かつ、S_{old} が持つリファクタリングデータを含む新しいソースコード S_{new} を生成する。

ソースコード統合部は、まず S_{gen} と S_{old} を統合したソースコード S_{merge} を生成する。テストケースの追加による ソースコード S_{gen} の変更部分と、リファクタリングにより変更された S_{old} の変更部分とが重なっている場合、ソースコード統合部は、S_{old} を優先して統合したソースコード $S_{ahead\ old}$ と、S_{gen} を優先して統合したソースコード $S_{ahead\ gen}$ を生成する。そして、S_{merge}、$S_{ahead\ old}$、$S_{ahead\ gen}$ を順にテストし、テストにパスできたものを新しいソースコード S_{new} とする。すべてのソースコードがテストにパスできない場合は、正しくソースコードを生成できないことを開発者に通知し、処理を終了する。

2.5.2 レビューとリファクタリングの要求

ソースコード統合部は、開発者に S_{new} のレビューとリファクタリングを要求する。開発者から承認を受けた後、ソースコード統合部は既存のソースコードをリファクタリングされた S_{new} で上書きする。

3. 適用例

提案したフレームワークが正しく機能することを、フレームワークを試作し確認する。フレームワークは Python で試作し、対象とするソースコードは C++とした。構文解析には Antlr（ANother Tool for Language Recognition）[6]を、テストには Google C++ Testing Framework を使用した [7]。ソースコードは、ヘッダーファイルとソースコードファイルの 2 つのファイルを生成する。

3.1 境界値分析に基づくテストケースに対するソースコード生成

提案したフレームワークが正しくソースコードを生成できることを確認するために、試作したフレームワークにテストコードを入力し、ソースコードを生成した。試作したフレームワークに入力したテストコードを、リスト 1 に示す。 リスト 1 に示すテストコードから生成したソースコードのうち、ヘッダーファイルをリスト 2 に、ソースコードファイルをリスト 3 に示す。入力したテストコードは、FeeCalculator クラスのメンバ関数 getFee() に対する境界値テストである。関数 getFee()は、年齢を整数型の引数として受け取り、0〜17 歳は 500、18〜60 歳は 800、61〜120 歳は 500、それ以外の値は-1 を返す。

リスト 1 に示すソースコードを入力として生成したソースコードは、すべてのテストケースをパスした。また、リスト 3 から、生成した条件分岐の範囲が妥当であることが確認できる。

リスト1. テストコードの例

```
#include "gtest/gtest.h"
#include "FeeCalculator.h"
namespace foolish_coder{
	TEST(CalcFeeTest, CalcFeeTestCaseYoungestChild){
		FeeCalculator fee_calculator;
		EXPECT_EQ(fee_calculator.calcFee(0), 500);
	}
	TEST(CalcFeeTest, CalcFeeTestCaseOldestChild){
		FeeCalculator fee_calculator;
		EXPECT_EQ(fee_calculator.calcFee(17), 500);
	}
	TEST(CalcFeeTest, CalcFeeTestCaseYoungestAdult){
		FeeCalculator fee_calculator;
		EXPECT_EQ(fee_calculator.calcFee(18), 800);
	}
	TEST(CalcFeeTest, CalcFeeTestCaseOldestAdult){
		FeeCalculator fee_calculator;
		EXPECT_EQ(fee_calculator.calcFee(60), 800);
	}
	TEST(CalcFeeTest, CalcFeeTestCaseYoungestOld){
		FeeCalculator fee_calculator;
		EXPECT_EQ(fee_calculator.calcFee(61), 500);
	}
	TEST(CalcFeeTest, CalcFeeTestCaseOldestOld){
		FeeCalculator fee_calculator;
		EXPECT_EQ(fee_calculator.calcFee(120), 500);
	}
	TEST(CalcFeeTest, CalcFeeTestCaseTooYoung){
		FeeCalculator fee_calculator;
		EXPECT_EQ(fee_calculator.calcFee(-1), -1);
	}
	TEST(CalcFeeTest, CalcFeeTestCaseTooOld){
		FeeCalculator fee_calculator;
		EXPECT_EQ(fee_calculator.calcFee(121), -1);
	}
}
```

リスト2. リスト1から生成したヘッダーファイル

```
#ifndef FEE_CALCULATOR_H
#define FEE_CALCULATOR_H
class FeeCalculator{
    public:
        int calcFee(int param1);
};
#endif
```

リスト3. リスト1から生成したソースコード

```
#include "FeeCalculator.h"
int FeeCalculator::calcFee(int param1){
   if(param1 <= -1 || 121 <= param1){
      return -1;
   } else if((0 <= param1 && param1 <= 17) || (61 <= param1 && param1 <= 120)){
      return 500;
   } else if((18 <= param1 && param1 <= 60)){
      return 800;
   }
}
```

3.2 次サイクルにおけるリファクタリングの保持

提案したフレームワークが前サイクルにおけるリファクタリングを保持することを確認するために、3.1節で生成したソースコードに対してリファクタリングを行い、次のサイクルにおけるソースコード生成の際にリファクタリングを保持できるかを確認した。リスト3に示したソースコードに対して、以下のリファクタリングを行った。

リスト4. リファクタリングを保持したままリスト3を拡張したソースコード

```
#include "FeeCalculator.h"
int FeeCalculator::calcFee(int age){
   if((0 <= age && age <= 17)){
      return 0;
   } else if((18 <= age && age <= 60)){
      return 800;
   } else if((61 <= age && age <= 120)){
      return 500;
   }
   return -1;
}
```

- -1を返す条件式を削除し、関数の末尾に「return -1;」を追加する。
- 引数名をparam1からageに変更する。

次に、0〜17歳は0を返すようにテストケースを拡張し、再度ソースコードを生成した。生成したソースコードをリスト4に示す。このソースコードは、拡張後のすべてのテストケースをパスした。また、リスト4の緑字で示すように、引数名の変更と-1を返す処理の変更についてのリファクタリングを保持してソースコードを拡張できたことが確認できる。

4. 評価

提案するフレームワークの有用性を評価するために、TDDにおける実装工程に要する時間を、手作業と提案フレームワークを用いた場合とで比較した。

実験に参加した被験者は、大学院生4名と学部4年生2名の計6名である。彼らはTDDを使って2つのタスクを解決する。半数は2つのタスクのうち1つ目のタスクにのみフレームワークを使用し、もう半数は2つのタスクのうち2つ目のタスクにのみフレームワークを使用した。

各タスクの内容を、以下に示す。

1. FeeCalculatorクラスのメンバ関数calcFee()を実装する。関数calcFee()は、ユーザの年齢を整数引数として受け取り、0~17歳は100、18~60歳は500、61~120歳は200、それ以外は-1を返す。
2. DatePickerクラスのメンバ関数getLastDayMonth()を実装する。関数getLastDayMonth()は、引数として月を整数で受け取り、1月、3月、5月、7月、8月、10月、12月は31、4月、6月、9月、11月は30、2月は28、それ以外は0を返す。

実験は、4つのステップで行った。各ステップの内容を、以下に示す。

1. 既存のソースコードでは通過できないテストケースを1つ作る。
2. テストにパスできるソースコードを、手動またはフレームワークを使用して実装する。
3. 生成されたコードをレビューし、必要に応じてリファクタリングする。

表 1. テストケース 1 つあたりの作業時間

タスク	工程	手作業	提案するフレームワーク
T1	実装時間	3m25s	**14s**
	リファクタリング時間	50s	50s
	総時間	5m4s	**1m54s**
T2	実装時間	3m31s	**11s**
	リファクタリング時間	**7s**	40s
	総時間	4m26s	**1m19s**
平均	実装時間	3m28s	**12s**
	リファクタリング時間	**29s**	45s
	総時間	4m45s	**1m36s**

4. 課題が完了したと被験者が判断するまで、ステップ 1〜3 を繰り返す。

1 つのテストケースあたりの平均時間について、ステップ 2 にかかる時間を実装時間、ステップ 3 にかかる時間をリファクタリング時間、ステップ 1〜3 にかかる時間を総時間とし、それぞれ計測した。実験で作業時間を計測した結果を、表 1 に示す。

表 1 より、提案したフレームワークにより、TDD における 1 つのテストケースに対する実装時間を 3m16s（94.22%）短縮したことがわかる。一方で、リファクタリング時間は 16s 長くなった。原因は、フレームワークが自動生成したソースコードのレビューにかかる時間である。しかし、総時間として、3m9s(66.17%)を短縮できた。したがって、提案フレームワークは、TDD における実装工程の効率化に有用である。

また、提案したフレームワークは、被験者による引数名に対するリファクタリングを保持したソースコードを生成した。

以上のことから、本論文で提案したフレームワークは、TDD における実装工程の継続的な支援に有用であるといえる。

5. おわりに

本研究では、TDD における実装工程の継続的な支援を目的としてフレームワークを提案した。提案したフレームワークについて、試作を行い、提案したアルゴリズムでテストコードを入力としたソースコードの生成、および、前サイクルにおけるリファクタリングを保持したソースコードの生成が行えることを確認した。また、評価実験において、試作したフレームワークは、TDD における 1 つのテストケースについて、実装工程に要する時間を 94.22%、開発時間を 66.17%削減した。更に、開発者によるリファクタリングを保持したソースコードを生成した。

以上のことから、提案したフレームワークは TDD における実装工程の継続的な支援に有用であるといえる。

今後の課題を以下に示す。

- ソースコード生成部の拡張による適用範囲の拡大
  本論文で提案したフレームワークのソースコード生成部は、境界値分析を基に設計されたテストケースが入力されることを想定しており、if 文と return 文による実装にしか対応していない。これにより、単純な実装にしか適用できず、有用性が低い。機械学習を用いて、汎用的なソースコード生成を実現することで、有用性を向上できると考える。

- ソースコード統合部の拡張によるリファクタリング保持率の向上
  本論文で試作したフレームワークでは、前サイクルにおけるリファクタリングを統合する際、ソースコードの差分だけに注目しており、その意味については解析していない。そのため、テストをパスすることを最優先したとき、リファクタリングが保持されない可能性がある。この問題に対し、ソースコード統合部によるソースコードの統合時に、ソースコード生成部による変更やリファクタリングによる変更について、それぞれコード中の位置や前後の文との文脈を解析することで、よりリファクタリングを保持できると考える。

- ソースコード解析部によるリファクタリング保持率の向上
  本論文で提案したソースコード解析部は、既存のソースコードから関数の引数名のみを抽出する。既存のソースコードから、頻度の高い構文や特徴的な記述を抽出し、保持できるようソースコード解析部を拡張することで、リファクタリングの保持率をより高くできると考える。

- 自動生成によるソースコードの変更内容の理解支援
  評価実験の結果から、フレームワークによって自動生成したソースコードのレビューに時間がかかることが分かった。自動生成によって生じたソースコードの変更を、開発者にわかりやすく提供し、ソースコードの理解を助けることで、レビューやリファクタリングにかかる時間を短縮し、保守性が高まると考えられる。変更内容の具体的な提供方法として、自動生成するコードにコメントを付加する、ソースコードに適用した変更を要約したメッセージを表示する、変更があった部分をマークし区別しやすくすることがあると考える。

参考文献

1) BBC: What went wrong inside Boeing's cockpit? . https://www.bbc.co.uk/news/extra/IFtb42kkNv/boeing-two-deadly-crashes#group-System-failure-zAFs52mVNd. Accessed: 2023/4/24.
2) F.Anwer, S.Aftab, U.Waheed, Muhammad,S.S., “Agile Software Development Models TDD, FDD, DSDM, and Crystal Methods: A Survey”, International Journal of Multidisciplinary Sciences and Enginnering, Vol. 8, No. 2, pp. 1-10, 2017.
3) Mukhtar,M.I., Galadanci ,B.S., “AUTOMATIC CODE GENERATION FROM UML DIAGRAMS: THE STATE-OF-THE-ART”, Science World Journal, Vol. 13, No. 4, pp. 47-60, 2018.
4) P.Yin, G.Neubig, “A Syntactic Neural Model for General-Purpose Code Generation”, Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, Vol. 1, pp. 440-450, 2017.
5) GIHOZ 境界値分析. https://www.veriserve.co.jp/gihoz/boundary.html. Accessed:2023/04/25.
6) Antlr. https://www.antlr.org/. Accessed:2023/04/23.
7) GitHub google/googletest. https://github.com/google/googletest. Accessed:2023/04/23.