Proposal of a Framework to Improve the Efficiency of the Implementation Step in Test Driven Development (TDD)

Takeaki Miyashita[*], Tetsuro Katayama[*], Yoshihiro Kita†,
Hisaaki Yamaba[*], Kentaro Aburada[*], and Naonobu Okazaki[*]
[*] *Department of Computer Science and Systems Engineering, Faculty of Engineering, University of Miyazaki*
1-1 Gakuen-kibanadai nishi, Miyazaki, 889-2192 Japan
[†]*Department of Information Security, Faculty of Information Systems, Siebold Campus, University of Nagasaki*
1-1-1 Manabino, Nagayo-cho, Nishi-Sonogi-gun, Nagasaki, 851-2195 Japan
E-mail: miyashita@earth.cs.miyazaki-u.ac.jp, kat@cs.miyazaki-u.ac.jp, kita@sun.ac.jp,
yamaba@cs.miyazaki-u.ac.jp, aburada@cs.miyazaki-u.ac.jp, oka@cs.miyazaki-u.ac.jp

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. 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. Introduction

Test Driven Development (TDD) is a software development methodology. In TDD, developers repeat a series of steps to get closer to better test cases and implementation: creating tests that fail the existing source code, implementing the minimum source code that will pass the tests, and refactoring the implemented source code. One of the disadvantages of TDD is that it wastes more time in some cases due to repeated test failure[1]. In recent years, many researches on automatic source code generation from UML and natural language have been reported[2][3]. However, most of them focus only on one-time generation and not on maintaining continuous refactoring.

This study proposes a framework aimed at supporting the implementation process in TDD. The proposed framework automatically generates new source code by modifying the original source code, which fails to pass tests by test code, to pass tests. In addition, refactoring to the generated source code is retained and reflected in the next development cycle.

2. Proposed Framework

In this chapter, we present the structure and behavior of the proposed framework. The framework takes as input the test code and the source code S_{old} that cannot pass the test by the test code, and outputs the source code S_{new} that can pass the test. The structure of the framework is shown in Fig. 1. The behaviors of the five processing parts shown in Fig. 1 are described as below.

2.1. *Test Code Analyzer*

Test Code Analyzer parses the given test code and extracts test case data from each test case. The test case data is defined as four elements: the names of the class and member function to be tested, the arguments given to the member functions to be tested, and the expected output of the function under test for the given arguments.

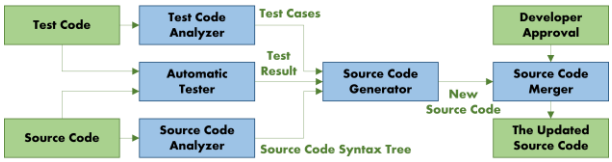

Fig. 1 The Structure of the Proposed Framework

The following is a flow of Test Code Analyzer's behavior.

(i) Test Code Analyzer parses the test code and generates an abstract syntax tree.
(ii) It searches for assertions in the generated abstract syntax tree and identifies functions that contain assertions.
(iii) For each variable in the function, create the variable table by extracting type and value. The value means either a literal value or a pair of the function and the value of the arguments given to the function. If the variable is assigned the return value of a function call, record the pair of the function and the argument values given to the function as the value of the variable in the variable table.
(iv) For values used in assertion, get the type and value from the variable table. If the value is a pair of the function and the arguments, the function shall be the function to be tested.
(v) It sends the extracted test case data to Source Code Generator (described in section 2.4).

2.2. *Source Code Analyzer*

Source Code Analyzer parses source code S_{old} and extracts existing class data. Specifically, the name and type of the argument of a member function are extracted. The class data is sent to the Source Code Generator (described in section 2.4). If source code S_{old} does not exist, Source Code Analyzer does not send data.

2.3. *Automatic Tester*

Automatic Tester automatically runs tests and extracts failed test case data from the test results. Specifically, it is the names of the test case and the member function that the test case target. The failed test case data is sent to the Source Code Generator (described in section 2.4).

2.4. *Source Code Generator*

Source Code Generator generates source code S_{gen} that satisfies the test case based on data received from Test Code Analyzer, Source Code Analyzer, and Automatic Tester. The Source Code Generator operates only if the existing source code S_{old} fails to pass the test. Otherwise, it terminates the process.

2.4.1. Integration of Input Data

Source Code Generator first extracts the test cases in the test case data received from the Test Code Analyzer that match the names of the failed test case data received from the Automatic Tester. In addition, the Source Code Generator also extracts test case data for the function that is the target of the failed test case in the same way. If the class data received from the Source Code Analyzer has data for a class that has a function targeted by the failed test case, extract this class data.

2.4.2. Estimation of Expected Output based on Boundary Value Analysis

Source Code Generator sorts test cases based on the arguments and member variables of the extracted test case data. Then, assume that the expected outputs from the arguments and member variables located between adjacent test cases from sorted test cases with the same expected outputs are equivalent. This assumption is based on the concept of boundary value analysis. Boundary value analysis is a test design method that uses test cases around boundary values. If the test cases are well designed, outputs by parameters between test cases that are adjacent on the sorted test cases and that expect the same value of output have a high probability of expecting the same value as well.

2.4.3.Source Code Generation to Satisfy Test Cases

conditional expression and a return statement with the expected output as the return value. The function intermediate data has a function name, a return type, and multiple blocks. The class intermediate data has a class name, member functions, and member variables.
Member intermediate data are generated based on the class data extracted in section 2.4.1 and the generated function. The Source Code Generator generates source code S_{gen} that satisfies the test case by recursively transforming the class intermediate data and sending it to the Source Code Merger (described in Section 2.5).

List. 1 Example of Test Code

```
#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);}}
```

2.5. *Source Code Merger*

Source Code Merger merges the refactoring data from the previous cycle with the S_{gen} generated by Source Code Generator and overwrites the existing source code.

2.5.1. Integration of Refactoring Data

Source Code Merger merges the existing source code S_{old} into the source code S_{gen} generated by Source Code Generator and generates a new source code S_{new} with the refactoring data included in S_{old}.

Source Code Merger first generates the source code S_{merge}, which merges S_{gen} and S_{old}. If conflicts exist between the parts in S_{gen} changed due to the addition of test cases and the parts in S_{old} changed due to refactoring, Source Code Merger generates source code $S_{ahead\ old}$ of S_{old} and source code $S_{ahead\ gen}$ merged ahead of S_{gen}. Then, S_{merge}, $S_{ahead\ old}$, and $S_{ahead\ gen}$ are tested in this order, and the one that can pass the test is the new source code S_{new}. If all source code cannot pass the test, notify the developer that it cannot be generated source code and terminate the process.

List.2 The Header File for Test Code is shown in List. 1

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

List. 3 The Source Code File for Test Code is shown in List. 1

```
#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;
  }}
```

2.5.2. Review and Refactoring Requests

Source Code Merger requires a developer to review and refactor S_{new}. After receiving approval from the developer, Source Code Merger overwrites the existing source code with the refactored S_{new}.

3. Application Example

We have prototyped the framework and generated C++ source code from the test code. It has been implemented in Python, with Antlr (ANother Tool for Language Recognition) used for parsing and Google C++ Testing Framework used for testing. The output source code is generated in two files: a header file and a source code file.

An example of the test code is shown in List. 1. The generated source code for the test code shown in List. 1 is shown in List.2 and List. 3. The header file is shown in List.2. The source code file is shown in List. 3. The test code is for the member function getFee() of the FeeCalculator class. The function getFee() takes the user's age as an integer argument and returns 500 for ages 0-17, 800 for ages 18-59, 500 for ages 60-120, and -1 for all other values.

The generated source code passed all test cases. Also, List. 3 shows that the range of conditionals generated for the example is reasonable. In addition, the following refactoring was done to the source code shown in List. 1.

- Delete the conditional expression that returns -1 and add "return -1;" at the end of the function.
- Change argument name to age.

List. 4 Extension with Retained Refactoring

```
#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;
}
```

Table 1 Time per Test Case

Method	Implementation Time	Refactoring Time	Total Time
Manual	3m28s	29s	4m45s
The Proposed Framework	12s	45s	1m36s

Then, the test case was extended to return 0 for 0-17 years old, and the source code was generated again. The generated source code is shown in List. 4. This source code passed all test cases after the extension. List. 4 shows that the refactoring could be retained and the program extended.

4. Discussion

The time required for the implementation process in TDD was compared between manual and using the proposed framework. The results are shown in Table 1.

A total of six subjects participated in the experiment: four graduate students and two fourth-year undergraduate students. They will use TDD to solve two tasks. Half of them used the framework only for the first of two tasks, while the others used the framework only for the second of two tasks.

The following is a description of the tasks.

(i) Member function calcFee() of the FeeCalculator class: Member function calcFee() of the FeeCalculator class: Function that takes the user's age as an integer argument and returns 100 for ages 0~17, 500 for ages 18~60, 200 for ages 61~120, and -1 for all other values

(ii) Member function getLastDayMonth() of the DatePicker class: Receives the month as an integer argument and returns 31 for January, March, May, July, August, October, and December, 30 for April, June, September, and November, 28 for February, and 0 for all other values.

The experimental flow is shown below.

(i) Make one test case that cannot be passed by the existing source code.
(ii) Implement source code that can pass testing, either manually or using the framework.
(iii) Review generated code and refactor as needed.
(iv) Repeat (i)-(iii) until the subject determines that task is complete.

For the average time per test case, the time taken for (ii) is the implementation time, the time taken for (iii) is the refactoring time and the time taken for (i)-(iii) is the total time. Table 1 shows that the proposed framework reduced the implementation time in TDD by 3m16s (94.22%). Refactoring time was 16s longer. The cause is the time taken to review the generated source code. Although, the total time was reduced by 3m9s (66.17%). In addition, the proposed framework retained the refactoring for the argument names. Therefore, the proposed framework is useful to improve the efficiency of the TDD implementation step.

5. Conclusion

We have proposed a framework to improve the efficiency of the implementation process in TDD. In the applied example, the prototype framework reduced especially the time required for the implementation process by 94.22% and the development time in TDD by 66.17%. It also retains refactoring by the developer.

Hence, the proposed framework is useful to improve the efficiency of the implementation process in TDD.

Future issues are as follows.

- Addition of supported syntax
- Improving refactoring retention

References

1. 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.
2. 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.
3. 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.

Authors Introduction

Takeaki Miyashita

Takeaki Miyashita received the Bachelor's degree in engineering (computer science and systems engineering) from the University of Miyazaki, Japan in 2022. He is currently a Master's student in Graduate School of Engineering at the University of Miyazaki, Japan. His research interests software testing and software quality.

Tetsuro Katayama

Tetsuro Katayama received a Ph.D. degree in engineering from Kyushu University, Fukuoka, Japan, in 1996. From 1996 to 2000, he has been a Research Associate at the Graduate School of Information Science, Nara Institute of Science and Technology, Japan. Since 2000 he has been an Associate Professor at the Faculty of Engineering, Miyazaki University, Japan. He is currently a Professor with the Faculty of Engineering, University of Miyazaki, Japan. His research interests include software testing and quality. He is a member of the IPSJ, IEICE, and JSSST.

Yoshihiro Kita

Yoshihiro Kita received a Ph.D. degree in systems engineering from the University of Miyazaki, Japan, in 2011. He is currently an Associate Professor with the Faculty of Information Systems, University of Nagasaki, Japan. His research interests include software testing and biometrics authentication.

Hisaaki Yamaba

Hisaaki Yamaba received the B.S. and M.S. degrees in chemical engineering from the Tokyo Institute of Technology, Japan, in 1988 and 1990, respectively, and the Ph D. degree in systems engineering from the University of Miyazaki, Japan in 2011. He is currently an Assistant Professor with the Faculty of Engineering, University of Miyazaki, Japan. His research interests include network security and user authentication. He is a member of SICE and SCEJ.

Kentaro Aburada

Kentaro Aburada received the B.S., M.S, and Ph.D. degrees in computer science and system engineering from the University of Miyazaki, Japan, in 2003, 2005, and 2009, respectively. He is currently an Associate Professor with the Faculty of Engineering, University of Miyazaki, Japan. His research interests include computer networks and security. He is a member of IPSJ and IEICE.

Naonobu Okazaki

Naonobu Okazaki received his B.S, M.S., and Ph.D. degrees in electrical and communication engineering from Tohoku University, Japan, in 1986, 1988 and 1992, respectively. He joined the Information Technology Research and Development Center, Mitsubishi Electric Corporation in 1991. He is currently a Professor with the Faculty of Engineering, University of Miyazaki since 2002. His research interests include mobile network and network security. He is a member of IPSJ, IEICE and IEEE.