International Journal of Computer (IJC)

ISSN 2307-4523 (Print & Online)

http://ijcjournal.org/

An Enhanced Software Quality Testing Approach Using Metamorphic Testing Technique

Mohammed Abdalla Osman Mukhtar[a], Nisreen Beshir Osman[b*], Sulieman Ibrahim S. Bahar[c]

[a]Faculty of Computer Science and Information Technology, Alzaiem Alazhari University, Sudan

[b]Faculty of Computer Science and Information Technology, Bayan University. P.O. Box 210, Khartoum, Sudan

[c]Faculty of Computer Science and Information Technology, Geneina University, Sudan

[a]Email: mohammedabdalla@aau.edu.sd

[b]Email: nisreenbeshir@gmail.com, [b]Email: nisrenbeshir@bayan.edu.sd, [c]Email: sul_bahar@gu.edu.sd

Abstract

The software testing process plays an important role in improving the quality of the software product. The product or program which is free from errors greatly contributes to assuring the quality of the software. An oracle in software testing is a person (tester) who performs the testing process. The oracle problem is the difficulty of determining the expected outcomes of selected test cases. A tester (oracle) may not always be available, or might be available but the process is too expensive and difficult to apply. The research presented in this paper proposes an approach for reducing the effect of the oracle problem during testing software and hence enhancing the quality of testing. Metamorphic Testing (MT) approach has been introduced and applied to generate a follow-up test case for multiple executions of program under test and verify the result automatically. An experimental method has been used to explain the mechanism of work for (MT). JUNIT tool which supports MT has been used to apply selected case studies (trigonometric function, geometric shapes classification, booking web service). The obtained results showed a good enhancement in the testing process. The importance of this research lies in overcoming oracle problem or alleviates it and thus, the research contributes to knowledge the domain by guiding researchers to use the metamorphic method because of its great advantages, as well as evaluating the effect of metamorphic method through empirical studies.

Keywords: Software Quality; Metamorphic Testing Technique.

1. Introduction

Software testing is a mainstream approach to software quality assurance and verification. Metamorphic testing is an approach to both test case generation and test result verification [1]. A metamorphic testing method has been proposed to test programs without the involvement of an oracle.

----

* Corresponding author.

It employs properties of the target function, known as metamorphic relations, to generate follow-up test cases and verify the outputs automatically. An “oracle” in software testing is a procedure by which testers can decide whether the output of the program under testing is correct. In some situations, however, the oracle is not available or too difficult to apply. This is known as the “oracle problem”. In other situations, the oracle is often the human tester who checks the testing result manually. The manual prediction and verification of program output greatly decreases the efficiency and increases the cost of testing [2].

2. Background and Related Work

Let p be a program implementing a specification f. Let D represent the input domain. Usually, it is impossible to do exhaustive testing to check whether $p(t) = f(t)\ \forall t \in D$. As a result, a great amount of research in the literature of software testing has been devoted to the development of test case selection strategies, aiming at selecting those test cases that have a higher chance of detecting a failure. Let $T = \{t_1, t_2, \ldots, t_n\} \subset D$ be the set of test cases generated according to some test case selection strategy, where $n \geq 1$. Running the program on these test cases, the tester will check the outputs $\mathbf{p}(t_1), \mathbf{p}(t_2), \ldots, \mathbf{p}(t_n)$ against the expected results $\mathbf{f}(t_1), \mathbf{f}(t_2), \ldots, \mathbf{f}(t_n)$, respectively. If it is found that $\mathbf{p}(t_i) = \mathbf{f}(t_i)$ for some i, where $1 \leq i \leq n$, then we say a “failure” is revealed and t_i is a failure-causing input. Otherwise t_i is a successful test case. The procedure through which the tester can decide whether $\mathbf{p}(t_i) = \mathbf{f}(t_i)$ is called an oracle. For instance, let $f(x, y) = x \times y$, the test case t_i be $\{x = 3.2, y = 4.5\}$, and $\mathbf{p}(t_i) = 14.4$. The tester can verify this output either by manually calculating the product of 3.2×4.5 or using the inverse function to check whether $14.4/4.5 = 3.2$, where the inverse can be done either manually or using a correct division program if available [3].

Software quality assessments is divided into two categories, the first one is static analysis it examines the code and reasons over all behaviors that might arise during run time[4]. The second is dynamic analysis which means actual program execution to expose possible program failure. Figure 1 explains static and dynamic test in V-Model.

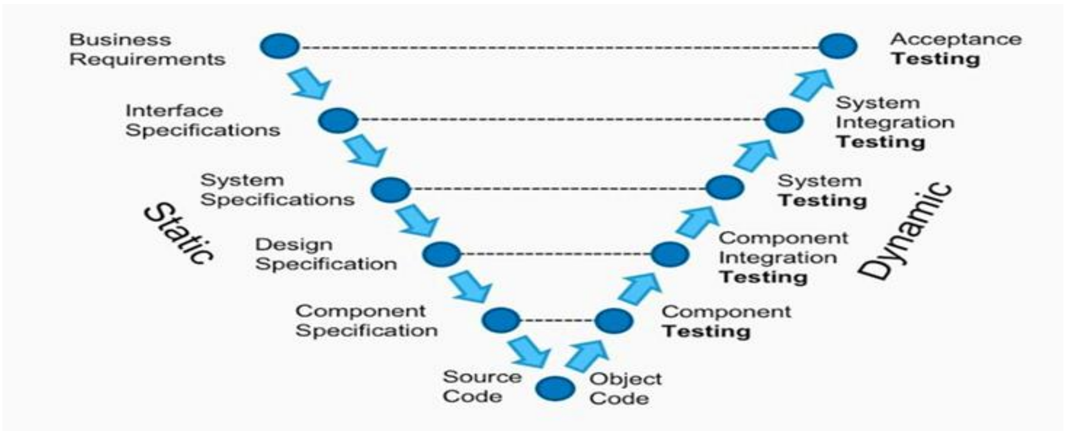


Figure 1: Static and Dynamic Test in V-Model [5].

Metamorphic testing has been suggested by Chen and his colleagues [6] as a way of testing applications that do not have test oracles ensuring that the software under test exhibits its expected metamorphic properties.

MT has been developed to alleviate the oracle problem. Instead of focusing on the correctness of each individual output, MT checks the program against selected metamorphic relations (MRs). MR is a necessary property of the target function, and is a relation that involves multiple executions of the target function. MRs is identified based on knowledge of the problem domain, such as known properties of the target function/system or knowledge about the algorithms to be implemented. Each metamorphic test involves two or more executions of the program under test [7]. In this part of the research, summarization of related works should be provided in Table 1. Accordingly, some properties and advantages of this research will be discussed in compare with the following related works to explain the goodness of the current work.

Table 1: Summarization of Using Metamorphic Approach to Solve Oracle Problems.

References	Oracle problem	Solution	Metamorphic testing
J. Chen and his colleagues [8]	Yes	Yes	Yes
S. K. Yong [9]	Yes	Yes	Yes
Z. Q. Zhou and his colleagues [10]	Yes	Alleviated	Yes
W. K. Chan and his colleagues [11]	Yes	Alleviated	Yes
C. Aruna and R. S. R. Prasad [12]	Yes	Yes	Yes
S. Segura [13]	Test case generation	Survey paper	Yes
Murphy, G. Kaiser, and L. Hu [14]	Yes	Yes	Yes
In [15] and F. Kuo and his colleagues [16]	Yes	Yes	Yes
W. K. Chan and his colleagues [17]	Yes	Yes	Yes
L. Xu, and his colleagues [18]	Test case generation	Yes	Yes
A. Goffi [19]	Yes	Yes	Yes
L. Xu and D. Towey, and his colleagues [20]	Yes	Yes	Yes
D. Peters and D. L. Parnas [21]	Test case generation	Yes	Yes
J. Ding, and his colleagues [22]	Yes	Yes	Yes

3. The Proposed Method

For ease of presentation, unless stated otherwise, we assume that each metamorphic test involves only two executions; the first execution, which is called the source execution, with its input called the source test case; and the next execution, called the follow-up execution with its input called the follow-up test case. If the outcomes of a source and its follow-up executions are found to violate an MR, the software under test must contain a fault. Figure 2 explains relationships between several metamorphic testing conceptions.

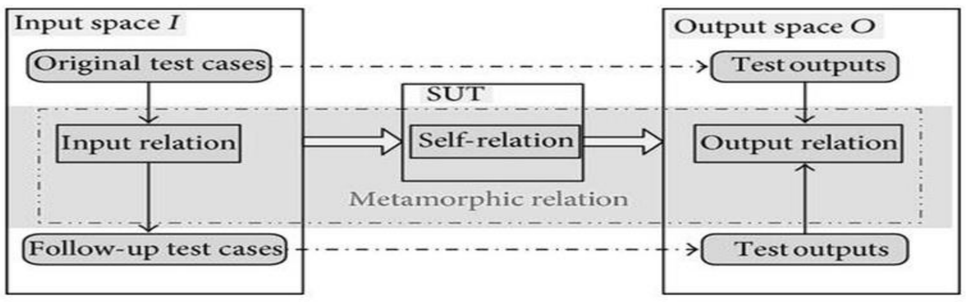


Figure 2: Relationships between Several Metamorphic Testing Conceptions.

For each case study, metamorphic file has been created (java file) to be passed to JUNIT tool for testing. Each test case should be either passed or failed.

3.1 Description (The Steps)

This section explored the steps of proposed method in details and how the proposed method enables the software developers (testers) to confirm the correctness of (MT).

3.1.1 Step One: Metamorphic Relation Identification

In order to identifying metamorphic relation for software under test, there is the need for user specification. So metamorphic relation includes test suit inputs and their corresponding outputs.

3.1.2 Step Two: Test Cases Generation

The test case is divided into two types, source test cases which are generated using traditional test case selection strategies and follow-up test cases it is constructed from the source test cases according to the metamorphic relations.

3.1.3 Step Three: Test Case Execution

Both source and follow-up test cases are applied to the software under test to know if test pass or fail.

3.1.4 Step Four: Outputs Verification

The test case outputs are checked against the relevant metamorphic relations to confirm whether the relations are satisfied, or have been violated. Therefor if expected outputs equal to actual value the test is pass otherwise the test fail.

3.2 Implementation (3 Case Studies)

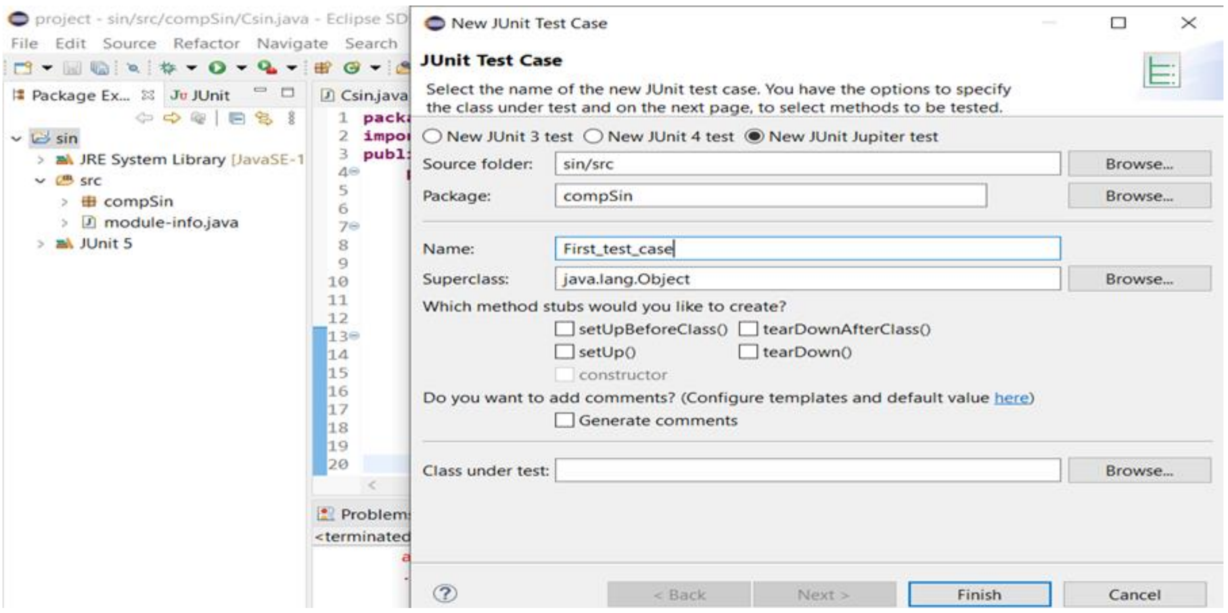
This section explored in details how to implement proposed method steps on 3 case studies to know the effectiveness of (MT).

3.2.1 Trigonometric Function

In order to find the value of the function sin (x), array created to store elements which generated randomly. These elements passed to the function as inputs to get the outputs each time and the validation of those outputs through the mechanism of the metamorphic testing method, and this method does not deal with the single execution but with the multiple execution of the program under test. These parameters which passed to the function are a set of test cases.

3.2.1.1 Metamorphic Relation Identification

When step1 of proposed method has been implemented on case study 1 the metamorphic relation identified according to problem domain and user specification. Do not need to remind, this steps extracted manually.



JUnit Test Case

Select the name of the new JUnit test case. You have the options to specify the class under test and on the next page, to select methods to be tested.

New JUnit 3 test   New JUnit 4 test   [✓ New JUnit Jupiter test]

Source folder: sin/src

Package: compSin

Name: First_test_case

Superclass: java.lang.Object

Which method stubs would you like to create?

☐ setUpBeforeClass() ☐ tearDownAfterClass()
☐ setUp() ☐ tearDown()
☐ constructor

Do you want to add comments? (Configure templates and default value here)

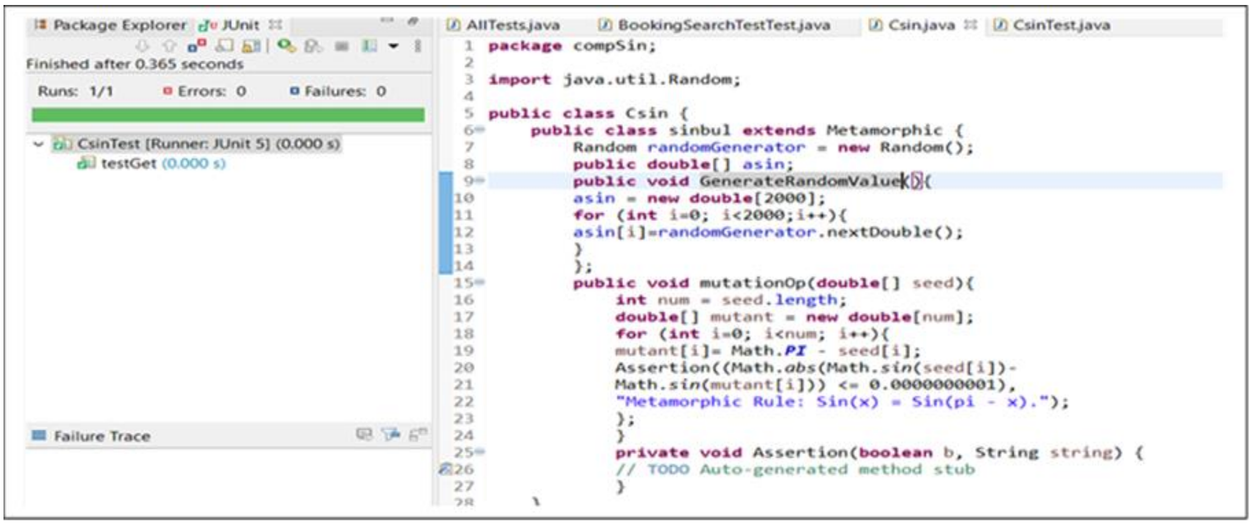
☐ Generate comments

Class under test:

Figure 3: Test Case Generation Process.

3.2.1.2 Test Cases Generation

When step2 has been implemented on case study1 the test cases generated according to test case selection strategies. Figure 3 explains test case generation for case study 1.



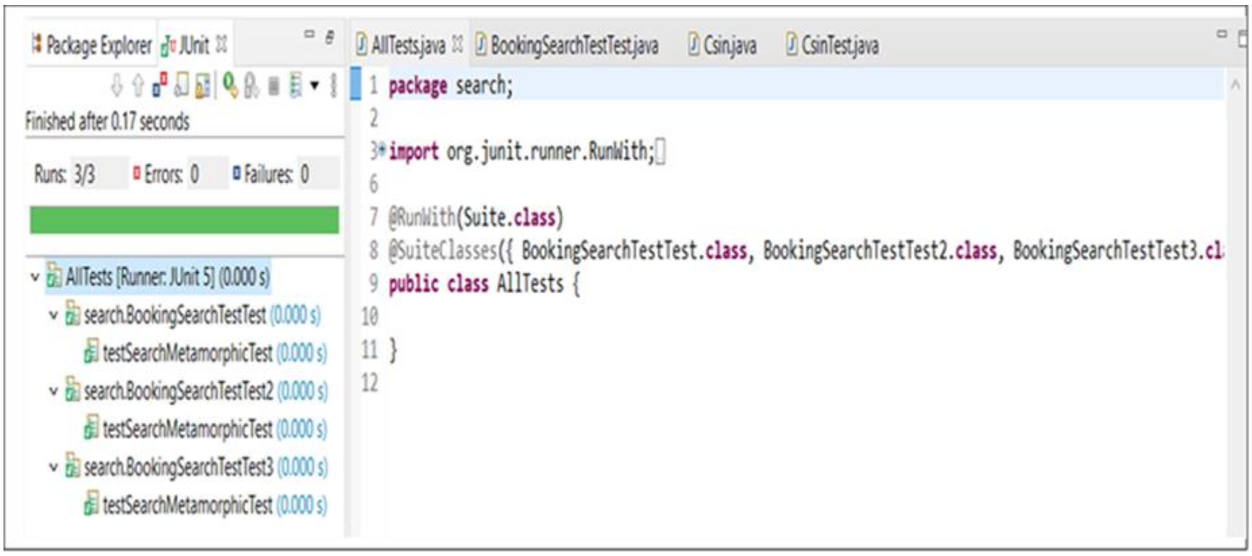
Finished after 0.365 seconds

Runs: 1/1   Errors: 0   Failures: 0

CsinTest [Runner: JUnit 5] (0.000 s)
testGet (0.000 s)

```java
package compSin;

import java.util.Random;

public class Csin {
    public class sinbul extends Metamorphic {
        Random randomGenerator = new Random();
        public double[] asin;
        public void GenerateRandomValue(){
        asin = new double[2000];
        for (int i=0; i<2000;i++){
        asin[i]=randomGenerator.nextDouble();
        }
        };
        public void mutationOp(double[] seed){
            int num = seed.length;
            double[] mutant = new double[num];
            for (int i=0; i<num; i++){
            mutant[i]= Math.PI - seed[i];
            Assertion((Math.abs(Math.sin(seed[i])-
            Math.sin(mutant[i])) <= 0.0000000001),
            "Metamorphic Rule: Sin(x) = Sin(pi - x).");
            };
            }
            private void Assertion(boolean b, String string) {
            // TODO Auto-generated method stub
            }
    }
```

Figure 4: sin(x) Test Case Implementation.



```
Finished after 0.17 seconds

Runs: 3/3    Errors: 0    Failures: 0

AllTests [Runner: JUnit 5] (0.000 s)
  search.BookingSearchTestTest (0.000 s)
    testSearchMetamorphicTest (0.000 s)
  search.BookingSearchTestTest2 (0.000 s)
    testSearchMetamorphicTest (0.000 s)
  search.BookingSearchTestTest3 (0.000 s)
    testSearchMetamorphicTest (0.000 s)
```

```java
package search;

import org.junit.runner.RunWith;

@RunWith(Suite.class)
@SuiteClasses({ BookingSearchTestTest.class, BookingSearchTestTest2.class, BookingSearchTestTest3.cl
public class AllTests {

}
```

Figure 5: Test Suite sin(x) Implementation.

3.2.1.3 Test Case Execution

When step3 of proposed method has been implemented on case study 1, the results already shown in Figure 4, which obviously displayed that no errors or failures have been detected, as well as the implementation of test suite for sin(x) has been shown in Figure 5.

3.2.1.4 Outputs Verification

When step4 has been implemented on case study1 the results of testing checked according to metamorphic rules.

3.2.2 Geometric Shapes Classification

In the second case study, the idea of triangle classification program is it has consisted of input as three natural numbers X, Y, and Z as the length of the side of a triangle. Its function is to classify triangle into equilateral (all sides the same length), or isosceles (two the same), or scalene (none the same), or to determine that the input does not represent an actual triangle when the summary of two parameters is not greater than the third. Table 2 contains these seed test cases.

Table 2: Four Test Cases Possibilities for Second Case Study.

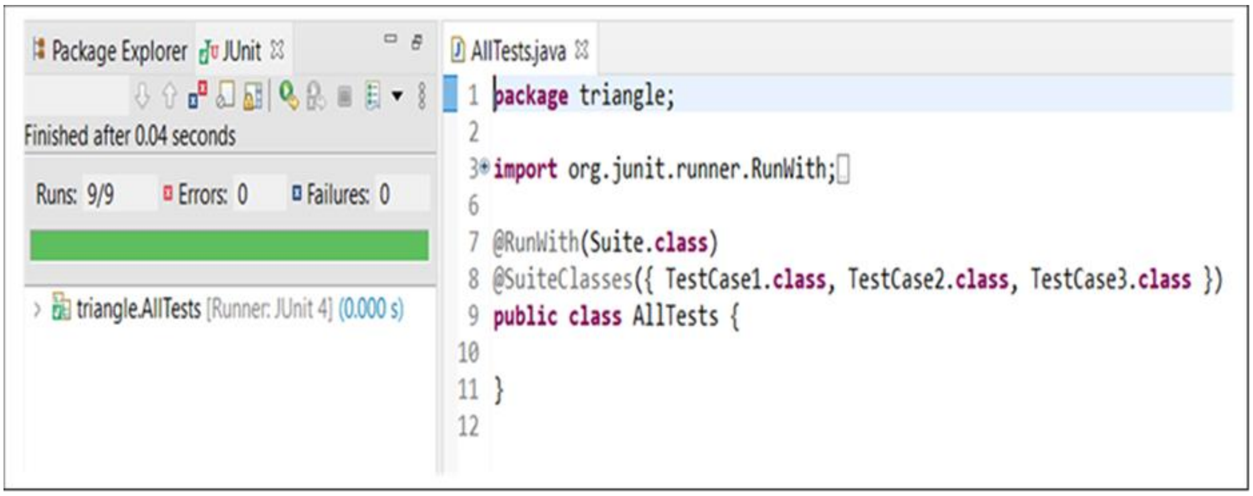
Test Case	**Input**	**Expected Output**
t1	X = 4, Y = 4, Z = 4	Equilateral
t2	X = 4, Y = 4, Z = 6	Isosceles
t3	X = 4, Y = 6, Z = 8	Scalene
t4	X = 2, Y = 5, Z = 8	Not a triangle

3.2.2.1 Metamorphic Relation Identification

When step1 has been implemented on case study1 the metamorphic relation identified according to problem domain and user specification.

3.2.2.2 Test Cases Generation

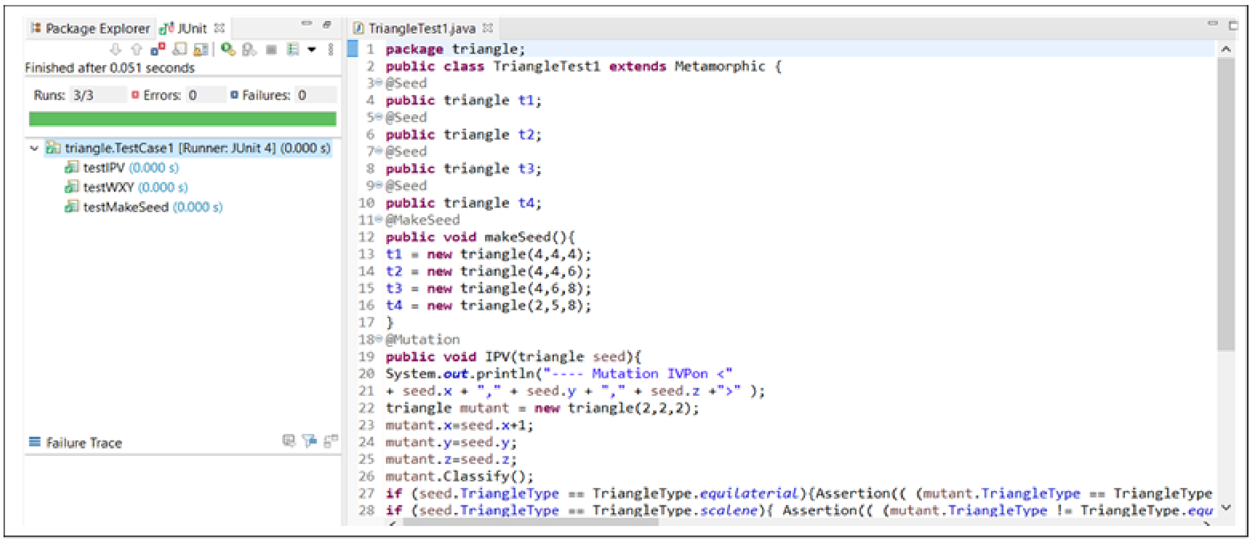
When step2 of proposed method has been implemented on case study 2, test cases have been generated according to test case selection strategies, and there are two types of test cases the first one is source test case and the second is follow up test case. Figure 3 explained test case generation.



```
Finished after 0.04 seconds
Runs: 9/9    Errors: 0    Failures: 0
triangle.AllTests [Runner: JUnit 4] (0.000 s)
```

```java
package triangle;

import org.junit.runner.RunWith;

@RunWith(Suite.class)
@SuiteClasses({ TestCase1.class, TestCase2.class, TestCase3.class })
public class AllTests {

}
```

Figure 6: Triangle Shape Test Case Implementation.



```
Finished after 0.051 seconds
Runs: 3/3    Errors: 0    Failures: 0
triangle.TestCase1 [Runner: JUnit 4] (0.000 s)
    testIPV (0.000 s)
    testWXY (0.000 s)
    testMakeSeed (0.000 s)
```

```java
package triangle;
public class TriangleTest1 extends Metamorphic {
@Seed
public triangle t1;
@Seed
public triangle t2;
@Seed
public triangle t3;
@Seed
public triangle t4;
@MakeSeed
public void makeSeed(){
t1 = new triangle(4,4,4);
t2 = new triangle(4,4,6);
t3 = new triangle(4,6,8);
t4 = new triangle(2,5,8);
}
@Mutation
public void IPV(triangle seed){
System.out.println("---- Mutation IVPon <"
+ seed.x + "," + seed.y + "," + seed.z +">" );
triangle mutant = new triangle(2,2,2);
mutant.x=seed.x+1;
mutant.y=seed.y;
mutant.z=seed.z;
mutant.Classify();
if (seed.TriangleType == TriangleType.equilaterial){Assertion(( (mutant.TriangleType == TriangleType
if (seed.TriangleType == TriangleType.scalene){ Assertion(( (mutant.TriangleType != TriangleType.equ
```

Figure 7: Triangle Shape Test Suite Implementation.

3.2.2.3 Test Case Execution

When step3 of proposed method has been implemented on case study 2, the test case inputs implemented and results showed. No errors or failures occurred and this is strong evident that the proposed method more effective to detect or reveal different types of errors for enhancing the quality of software under test. Figure 6 explains triangle test case implementation. In addition, the implementation of test suite has been shown in Figure 7.

3.2.2.4 Outputs Verification

When step4 of proposed method has been implemented on case study 2, the outputs of software under test verified against metamorphic relation. The verification step checks whether expected outputs has passed the test by equaling to actual value, or fail to pass this test.

3.2.3 Booking Web Service

In this case study the idea of the program is enabling users to find potential lodgings according to their preferences. Firstly, the program creates booking query object which includes destination room and setting beginning and ending of date then setting adult's number. Secondly the program creates follow up test case through setting budget and currency. Finally, the program specifies metamorphic relations assertion. Metamorphic testing method in this case generate test case, execute them verifies the output automatically.

3.2.3.1 Metamorphic Relation Identification

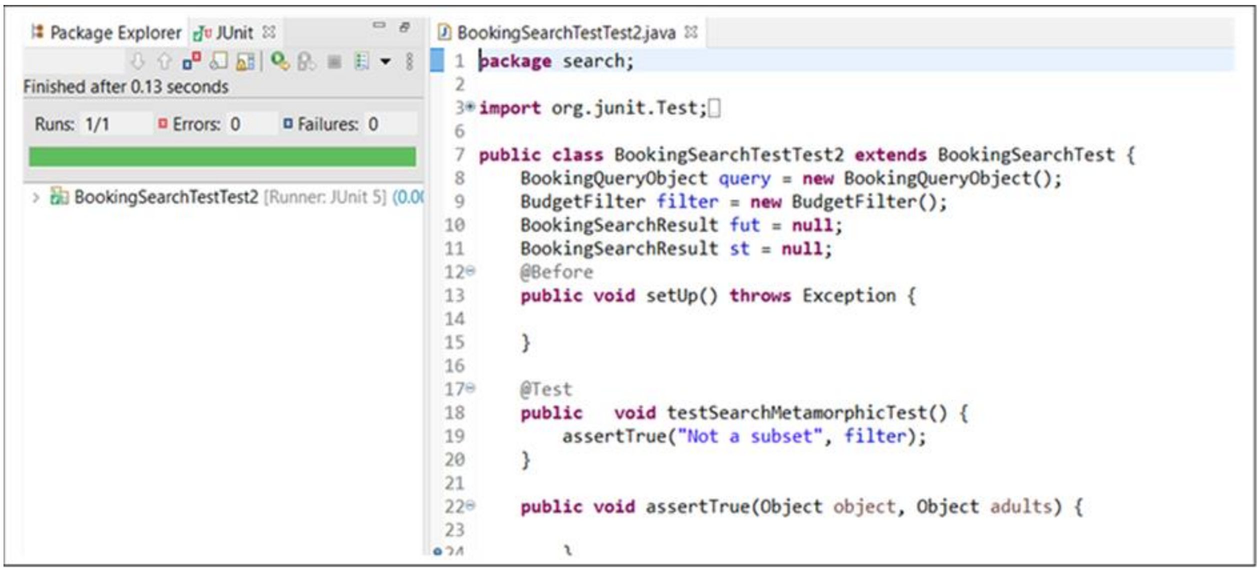
When step1 of proposed method has been implemented on case study 3 the metamorphic relation identified according to problem domain or from user specification. As mentioned before, this steps are usually extracted manually.

3.2.3.2 Test Cases Generation

When step2 of proposed method has been implemented on case study 3, test cases have been generated according to test case selection strategies. There are two types of test cases the first one is source test case and the second is follow up test case which is extracted from source test case. Refer to Figure 3 to see the general test case generation process.

3.2.3.3 Test Case Execution

When step3 of proposed method has been implemented on case study 3, the test case inputs implemented and appeared results, that no errors or failures appeared and this is strong evident that the proposed method more effective to detect or reveal different types of errors for enhancing the quality of software under test. Figure 8 explains booking web service test case implementation. In addition, the implementation of test suite has been shown in Figure 9 for the same case study.



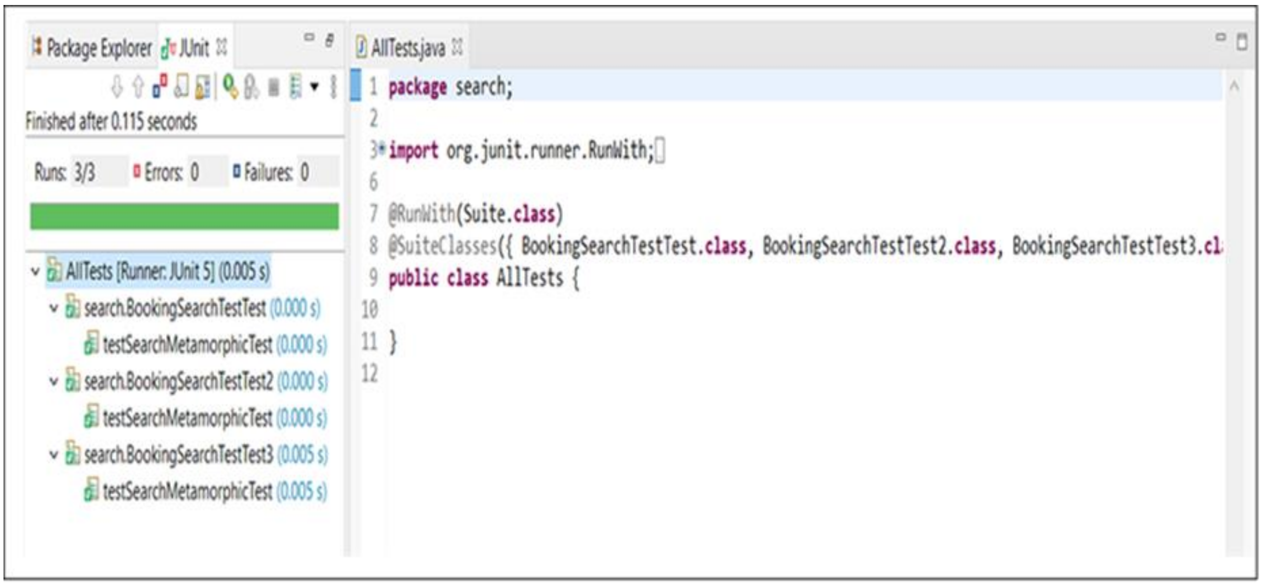
```
Finished after 0.13 seconds
Runs: 1/1    Errors: 0    Failures: 0
> BookingSearchTestTest2 [Runner: JUnit 5] (0.0
```

```java
package search;

import org.junit.Test;

public class BookingSearchTestTest2 extends BookingSearchTest {
    BookingQueryObject query = new BookingQueryObject();
    BudgetFilter filter = new BudgetFilter();
    BookingSearchResult fut = null;
    BookingSearchResult st = null;
    @Before
    public void setUp() throws Exception {

    }

    @Test
    public   void testSearchMetamorphicTest() {
        assertTrue("Not a subset", filter);
    }

    public void assertTrue(Object object, Object adults) {

        }
```

Figure 8: Booking Web Service Test Case Implementation.

3.2.3.4 Outputs Verification

When step4 of proposed method has been implemented on case study 3, the outputs of software under test verified and checked against metamorphic relation, so the verification step checks if expected outputs equal to actual value then the test is pass otherwise the test is fail.



```
Finished after 0.115 seconds
Runs: 3/3    Errors: 0    Failures: 0
v AllTests [Runner: JUnit 5] (0.005 s)
  v search.BookingSearchTestTest (0.000 s)
      testSearchMetamorphicTest (0.000 s)
  v search.BookingSearchTestTest2 (0.000 s)
      testSearchMetamorphicTest (0.000 s)
  v search.BookingSearchTestTest3 (0.005 s)
      testSearchMetamorphicTest (0.005 s)
```

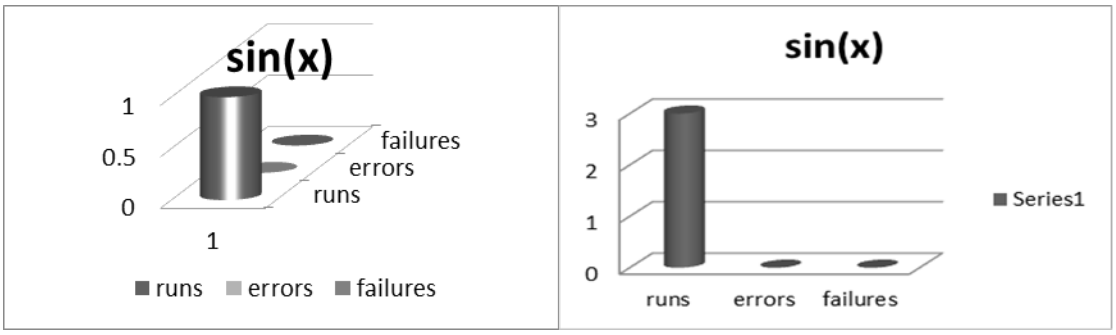
```java
package search;

import org.junit.runner.RunWith;

@RunWith(Suite.class)
@SuiteClasses({ BookingSearchTestTest.class, BookingSearchTestTest2.class, BookingSearchTestTest3.cl
public class AllTests {

}
```

Figure 9: Booking Web Service Test Suit Implementation.

4. Results and Discussions

In terms of case study, empirical execution of program in JUNIT tool showed that total runs are 1, failures are 0 and errors are 0 for one test case. On the hand, for test suits, runs are 3, failures are 0 and errors are 0. Figure

10(a) and figure 10(b) explain implementation of sin(x) as well as explain all aspects related to implementation using JUNIT.

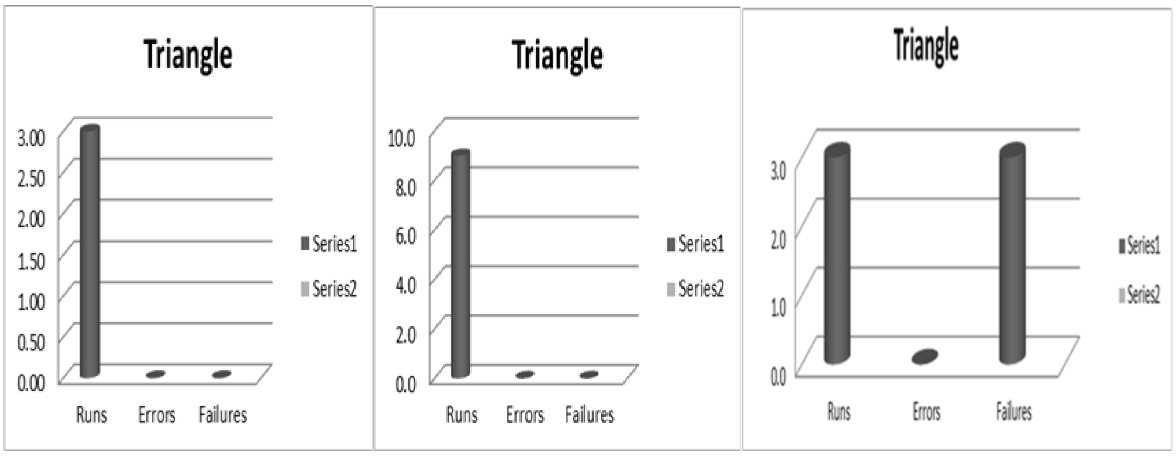


(a) **(b)**

Figure 10: Test Case Execution for sin(x).

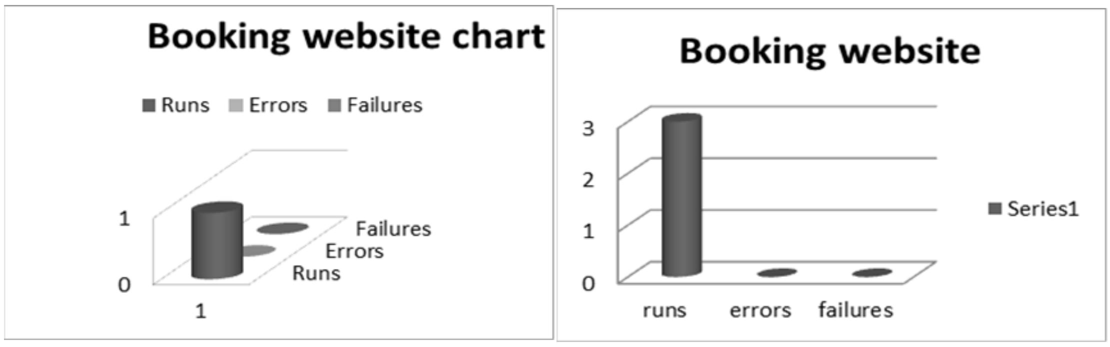
The implementation process was seen free of errors, thus ensuring the smoothness and integrity of the code. On the other hand, the implementation result was devoid of failure, meaning there are no problems in the testing process, and this increases reliability, and thus it showed that the metamorphic method has a high efficiency to alleviate the problem of oracle. The absence of errors and failures in the results of the testing process means the effectiveness of metamorphic method in ensuring the correctness of the outputs and thus contributes (MT) to improving the quality of the program.

Triangle has been taken as one of geometric shape classification. A set of test cases can be created and executed based on a set of pre-existing test cases, for example in the triangle classification program, there are three test cases: test case 1, test case 2 and test case 3. When the test case has been started as shown in Figure 11(a), the number of parameters passed are 9, the errors are 0, and failure are 0, while only one test case executed in Figure 11(b). So that the total run becomes 3 with 0 errors and 0 failures. In Figure 11(c), the chart explains that total runs are 3, errors are 0 and failures are 3. From the above charts, the fewer errors in the implementation, the more reliable and efficient the metamorphic method in software testing, and the success of all test cases means there is a strong logic code that can be relied upon. The absence of errors and failures in the results of the testing process means the efficacy of the metamorphic method in ensuring the correctness of the outputs and thus (MT) contributes to improving the quality of the program. Although there is a high implementation failure in Figure 11(c), it can be fixed by rewriting the test case. After case study3 has been implemented, a test suit can be created and executed based on a set of pre-existing test cases. For example, in the booking searching program, there are three test cases: test case1, test case 2 and test case 3. When the test case has been started as shown in Figure 12(b), the number of parameters passed are 3, the errors are 0, and failure are 0, while only one test case executed in Figure 12(a), error are 0 and failure are 0.



(a) **(b)** **(c)**

Figure 11: Test Case Execution for Triangle Geometric Shape.



(a) **(b)**

Figure 12: Test Case Execution for Booking Web Site.

4.1 General Discussions

Through the empirical study and the results, it became clear that the testing process became smooth, as the successful implementation of each case study separately.

In the first case study, geometric shapes - one test case was executed and no error or failure was shown in the implementation. This is considered to be that the metamorphic method can be rely upon to overcome the oracle problem, as well as that the code is free of logical errors, but in return it can appear failures in execution, and this is normal due to the tool used, as some problems appear when creating a test case for the first time, and this requires rewriting the test case before implementation. In the second case study - classification of geometric shapes, there is also a similarity in the results of the implementation, when executing one test case a very high failure occurred and it was fixed by rewriting the test case and this is normal as mentioned above, while a number of test cases were executed and no errors or failures appeared in execution. In third case study- booking web service - one test case was executed, and the result of the implementation did not show any error or failure mentioned, which is evidence of the strength of the metamorphic testing in improving the quality of software testing. On the other hand, when implementing a number of test cases also did not appear any error or failure in the result of the implementation. Through the discussion and analysis above, we conclude that there is similarity in implementation between the three case studies, and appearance of failure in implementation is a natural thing due to the deficiency of the tool used in generating test cases, and despite some limitations of the metamorphic method, it is the most appropriate in software testing because of its characteristics which can link inputs and outputs to multiple relationships and validate results, thus contributing to improving the quality of the program under test.

5. Conclusions

The use of metamorphic test (MT) method to overcome the problem of test case oracle, which is expected to contribute greatly to improving the efficiency of the testing process and the accuracy of the results. In addition, the experimental results showed that the MT is very effective way to test program without involving an oracle.

6. Future Work

This research recommends using MT approach to cover other domains in machine learning, modeling and simulation, deep learning and computer graph etc.

References

[1] Z. H. I. Q. Zhou, “Metamorphic Testing : A Review of Challenges and Opportunities,” vol. 000, no. 000, 2017.

[2] Z. Q. Zhou, D. Huang, T. Tse, Z. Yang, H. Huang, and T. Chen, “Metamorphic testing and its applications,” *Proceedings of the 8th International Symposium on Future Software Technology (ISFST 2004)*, 2004.

[3] “Case Studies on the Selection of Useful Relations in Metamorphic Testing ⋆.pdf.” .

[4] E. Fuchs, *Quality: Theory and Practice*, vol. 65, no. 2. 1986.

[5] “V-model – Software Testing Watch This Video.”

[6] C. Murphy, “Metamorphic Testing Techniques to Detect Defects in Applications without Test Oracles,” 2010.

[7] T. Y. Chen, F. C. Kuo, D. Towey, and Z. Q. Zhou, “Metamorphic testing: Applications and integration with other methods: Tutorial synopsis,” in *Proceedings - International Conference on Quality Software*, 2012, pp. 285–288.

[8] J. Chen, Y. Wang, Y. Guo, and M. Jiang, *A metamorphic testing approach for event sequences*, vol. 14,

no. 2. 2019.

[9] S. K. Yong, "Cost-effective Metamorphic Testing Techniques for Failure Detection in Software with Oracle Problem," 2015.

[10] Z. Q. Zhou, S. Xiang, and T. Y. Chen, "Metamorphic Testing for Software Quality Assessment : A Study of Search Engines," no. January, 2015.

[11] W. K. Chan, T. Y. Chen, H. Lu, and S. S. Yau, "A Metamorphic Approach to Integration Testing of Context-Sensitive Middleware-Based Applications ∗," 2005.

[12] C. Aruna and R. S. R. Prasad, "MTAF : A Testing Framework for Metamorphic Testing Automation MTAF : A Testing Framework for Metamorphic Testing Automation," no. September 2015, 2017.

[13] S. Segura, "Metamorphic Testing 20 Years Later : A Hands-on Introduction," pp. 3–6, 2018.

[14] C. Murphy, G. Kaiser, and L. Hu, "Properties of Machine Learning Applications for Use in Metamorphic Testing."

[15] T. Y. Chen, F. Kuo, R. Merkel, and W. K. Tam, "Testing an Open Source Suite for Open Queuing Network Modelling Using Metamorphic Testing Technique Testing an Open Source Suite for Open Queuing Network Modelling using Metamorphic Testing Technique," no. April 2014, 2009.

[16] F. Kuo, T. Y. Chen, and W. K. Tam, "Testing Embedded Software by Metamorphic Testing : a Wireless Metering System Case Study," pp. 291–294, 2011.

[17] S. C. C. and K. R. P. H. L. W. K. Chan †‡, "Towards a Metamorphic Testing Methodology for Service-Oriented Software Applications ∗.pdf." .

[18] "Automated Metamorphic Testing on the Analyses of Feature Models☆.pdf." .

[19] A. Goffi, "Automatic Generation of Cost-Effective Test Oracles Categories and Subject Descriptors."

[20] L. Xu, D. Towey, A. P. French, S. Benford, and T. Y. Chen, "Enhancing Supervised Classifications with Metamorphic Relations," 2018.

[21] D. Peters and D. L. Parnas, "Generating a Test Oracle from Program Documentation work in progress."

[22] J. Ding, T. Wu, J. Q. Lu, and X. Hu, "Self-Checked Metamorphic Testing of an Image Processing Program Self-Checked Metamorphic Testing of an Image Processing Program," no. August, 2017.