ABSTRACT

Software Testing plays an important role in Software Development because it can minimize the development cost. State-based testing is known as deriving test cases from state machines and examining the dynamic behavior of the model. It helps in identifying different faults present within a model under the test. Various test cases are being generated from state chart diagrams. Unified modeling Language diagrams are used to generate test cases. Generating test cases for a machine using UML diagrams comes under the Model-based testing which helps in detecting more faster than the code based testing.

Contents

1. CHAPTER- 1

2. CHAPTER- 2

3. CHAPTER- 3

4. CHAPTER- 4

5. CHAPTER- 5

CHAPTER – 1

INTRODUCTION

Testing at the early phase of software development life cycle can able to find the ambiguities and inconsistencies in the design and hence, design should be enhanced before the program is written. Software testing is a process, to evaluate the functionality of a software application with an intent to find whether the developed software met the specified requirements or not and to identify the defects to ensure that the product is defect free in order to produce the quality product. Software testing is an investigation conducted to provide stakeholders with information about the quality of the software product or service under test. Software testing can also provide an objective, independent view of the software to allow the business to appreciate and understand the risks of software implementation. Test techniques include the process of executing a program or application with the intent of finding software bugs (errors or other defects), and verifying that the software product is fit for use.

Software testing involves the execution of a software component or system component to evaluate one or more properties of interest. In general, these properties indicate the extent to which the component or system under test:

- meets the requirements that guided its design and development,
- responds correctly to all kinds of inputs,
- performs its functions within an acceptable time,
- it is sufficiently usable,
- can be installed and run in its intended environments, and
- achieves the general result its stakeholders desire.

As the number of possible tests for even simple software components is practically infinite, all software testing uses some strategy to select tests that are feasible for the available time and resources. As a result, software testing typically (but not exclusively) attempts to execute a program or application with the intent of finding software bugs (errors or other defects). The job of testing is an iterative process as when one bug is fixed, it can illuminate other, deeper bugs, or can even create new ones.

Software testing can provide objective, independent information about the quality of software and risk of its failure to users or sponsors.

Research is going on state-based testing (SBT) to find effective test cases and to minimize cost of the test suite. In Software Development Life Cycle (SDLC), software testing has a significant role in the software development process .The intention is to find out bugs in the program. Earlier tester went for manual testing and that was a prolonged and tedious process which required more time to execute. Then the tester went for automated testing in the software development process. Till today it has been a challenge to have fully automated software .There are two types of software testing named as Black box testing and White box testing. Black box testing is also known as functional testing. In white box testing or code-based testing, the requirement of the source code is necessary and early fault detection in the specification or design phase is not possible. For this, Unified Modeling Language (UML) diagrams are used to generate test cases. Initially testers were going for traditional testing, which is also known as code

coverage testing. But, state-based coverage cannot be achieved in code-based testing (Binder, 2000). To achieve this, tester generates test scenarios from the state chart diagrams and then test cases are generated from these scenarios. Test cases are generated at design level and coverage analysis is performed from the source code. The diagrams are generated at the design stage of development life cycle. Generating test cases based on UML diagrams come under Model Based Testing (MBT). It is a better testing approach than code based testing as it detects the error at the early phase which requires less cost to fix it .MBT are conducted for the following reasons:

- To get an abstract model of the system;
- Validate the model;
- Generate and execute test cases;
- Assigning pass/fail verdict;
- Analyzing the execution result;
- To prevent fault;
- To reduce cost with updating test cases.

Early testing activities make early fault detection (Binder, 2000; Broy et al., 2005) and more and more articles are referred to as MBT using state-based testing. In a very recent article, we find in MBT, where it elaborates several findings from MBT users in industry, security testing and various MBT have proposed a SBT technique to design black box testing. State-based testing is primarily considered as a black box testing to generate test cases .

A test case is a document, which has a set of test data, expected results with preconditions and post conditions. Test case is a particular test scenario in order to verify action against a specific requirement. There are different types of software faults that can be found in different ways (y . A set of test cases is called a test suite. To examine the effectiveness of the test suit, tester goes for the mutation testing and sneak-path testing. Mutation testing technique is applied on the generated test suite to measure its efficiency. In mutation testing, a faulty version of a software system is generated by introducing some mutant in the software, which is known as mutant operators. Some mutants still undetected after conformance testing.

So, another way of testing where remaining mutants are killed is known as sneak-path testing Conformance testing is seeking to test specified behavior of the model and also it is equally important to test unspecified behavior of the model through sneak path testing.

CHAPTER – 2

LITERATURE SURVEY

We have done a survey on the State-based based testing by going through various journal papers.

- A survey on model based testing tools for test case generation (Author- Wen Bin Li,Franck Le Gall ,Naum Spasseski Year(2017).

- State based testing is a functional testing(Author-Florentum Ipate, Raluca Lefticana Year(2015))

- Test Case generation Based on state and activity models(Author-Santosh kumar Swain Durgaprasad Mohapatra,Rajib mall. Year-(2010)).

- Automatic Generation of test cases using effect graph(Author-Hyun Seung Son, Keunsang Yi , Beyung Kook Jeon. Year(2018))

CHAPTER – 3

PROBLEM FORMULATION

Initially testers were going for traditional testing, which is also known as code coverage testing. But, state-based coverage cannot be achieved in code-based testing .To achieve this, tester generates test scenarios from the state chart diagrams and then test cases are generated from these scenarios. Test cases are generated at design level and coverage analysis is performed from the source code. The complexity in code-based testing is that it can achieve 100 percent code coverage but it does not say about the behaviors of the system under test as it may not fulfil the state based coverage. In MBT, abstractions are applied when modelling, fault is prevented and cost can be reduced with updating testcases. The diagrams are generated at the design stage of development life cycle. Generating test cases based on UML diagrams come under Model Based Testing (MBT). It is a better testing approach than code based testing as it detects the error at the early phase which requires less cost to fix it.

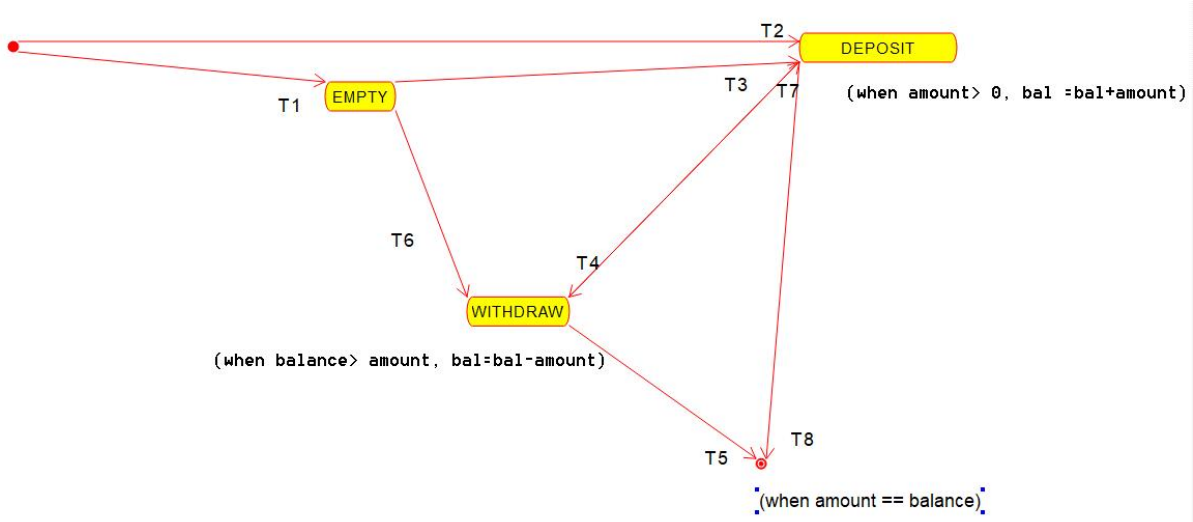
We discuss on generation of test cases based on different coverage criteria from two state chart diagrams. We consider the state chart diagrams of one case studies, Bank Account processing. To examine the efficiency of the test cases mutation testing and sneak-path testing are used. Model Based Testing is a state-based testing, in general term, test case generation and test result evaluation are done on the model of the system under test. SBT is done to derive test cases and to examine the dynamic behavior of the system .States are explained by their property, a constraint that must always be true when the SUT is in that state. State-based testing thus validates the states that are achieved, adaptable and the transitions that are evoked according to the requirement. State transition testing is used, in a finite state machine having finite number of different states and the transitions are from one state to another state. The system and the tests are based on the model. A finite state machine can handle the dynamic behavior of the system where you get a different output for the same input. The advantage of the state transition technique is that according to the tester's need, the model can be abstract or can be detailed. A part of the system is more important, if it requires more testing and a greater depth of detail can be modelled. The system is less important that requires less testing, the model can use a single state otherwise system goes for series of different states. State- based testing is a technique to validate software systems by generating test cases from models. State machine diagrams are similar to activity diagrams with little changes in its notations and usage. It describes the behavior of objects that act differently according to state with their state of moment Test cases are derived from state machines in the state-based testing that checks the expected system behavior. State-based testing validates the transitions and the states reached in the system under test. Using the generated tests cases, the tester checks whether the test cases are sufficiently cover the requirements or not. This can be verified through mutation testing. Mutation testing is also known as mutation analysis. machine. Turner (1993) proposed state based testing and investigates some new coverage criteria. Most studied coverage criteria are Full Predicate (FP), Round Trip Path (RTP), and All Transitions (AT), All Transition Pairs (ATP), ATP paths of length 2 (LN2), ATP paths of length 3 (LN3), and ATP paths of length 4 (LN4) . The FP criterion tends to obtain when model has guard conditions and FP has higher or similar mutation score as ATP, although at a higher cost. From the experimental results , it is shown that the

coverage criteria AT, RTP, ATP, and LN4 generate high quality test suites and they are sufficient powerful to detect the seeded faults that is the mutation testing.

Test Case Generation From State Chart Diagram

A TEST CASE is a set of conditions or variables under which a tester will determine whether a system under test satisfies requirements or works correctly.
The steps for generating test cases are as follows:
1. Construct the state chart diagram of the system using the UML Modeller. tool
2. Convert the state chart diagram into an intermediate graph.
3. Apply various algorithms on the intermediate graph to generate all possible test paths.
4. Generate test cases from the testing paths of the coverage criteria

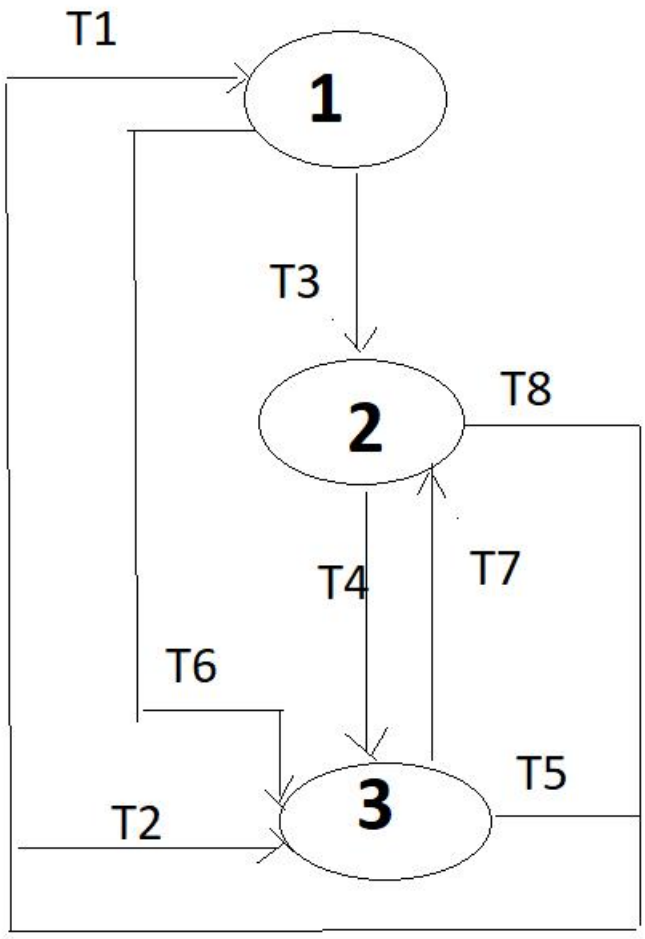
A state chart diagram of Bank Account processing is shown below. To create the state models, we have used Umbrello UML Modeller. Umbrello UML Modeler is used to create diagrams for designing and documenting the systems. For example, in Bank Account processing, the object Bank Account class has different states like Empty, Deposit, Withdraw and the transitions between them are created by the tool as a complete state model. We have use Junit tool with Eclipse to execute the test cases.



(fig1)

The sequences of transitions represented from the example model of Bank Account processing and t*i* represents states of transitions. AT: TC1=(T1, T3, T4,T5); TC2= (T2, T3,T4,T5); TC3= (T6,T7,T8) Test suite 1 [TC (I)] =3 (3 number of test cases are generated for AT). All Transition Pair criteria covers all pairs of adjacent transitions of the state machine and it also give less faults so we have use this coverage criteria.

The intermediate graph which is generated from the state diagram is given below-:



(Fig 2)

The basic details of the testcases are-:

T1- Instance is created using the default constructor.

T2- Instance is created using add balance constructor.

T3- Balance is added by deposit method.

T4 – Balance is withdrawn.

T5- Addition of negative balance.

T6- Withdraw when balance is 0.

T7 – Withdraw greater than balance.

T8-Withdraw negative balance.

In state-based testing, tester observes the state change in the SUT. Instate-based testing, tester also observes for state coverage, transition coverage and action coverage. The strategy which deals with the verification of control structures at the design level is called automata theoretic testing. That automata theoretic testing could find operation errors, extra states, missing states, and transition errors in a finite state machine. But, this method can only be used when the control structures are represented by finite state machine.

Now the work will be further extended by including the cost effectiveness analysis for test case generation of a critical system in SBT using the event dependency graph and we will add few more complex testcases by performing mutation testing.

CHAPTER - 4

RESULT AND ANALYSIS

The results of the testcases which we have taken are-:

T1:- In transition t1 the transition moves from initial to empty by create instance using default constructor method.

T2:- In transition t2 the transition moves from initial to deposit by create instance using add balance constructor.

T3:- In transition t3 the transition moves from empty to deposit to add balance(balance=balance+ amount) by deposit method.

T4:- In transition t4 the transition moves from deposit to withdraw balance (balance=balance-withdraw Amount) by withdraw method.

T5:- In transition t5 the transition moves from deposit to exit state because if (amount<0) it will show "Invalid deposit amount".

T6:- In transition t6 the transition moves from empty to withdraw because if we try to withdraw when balance=0 then it will show "balance is 0".

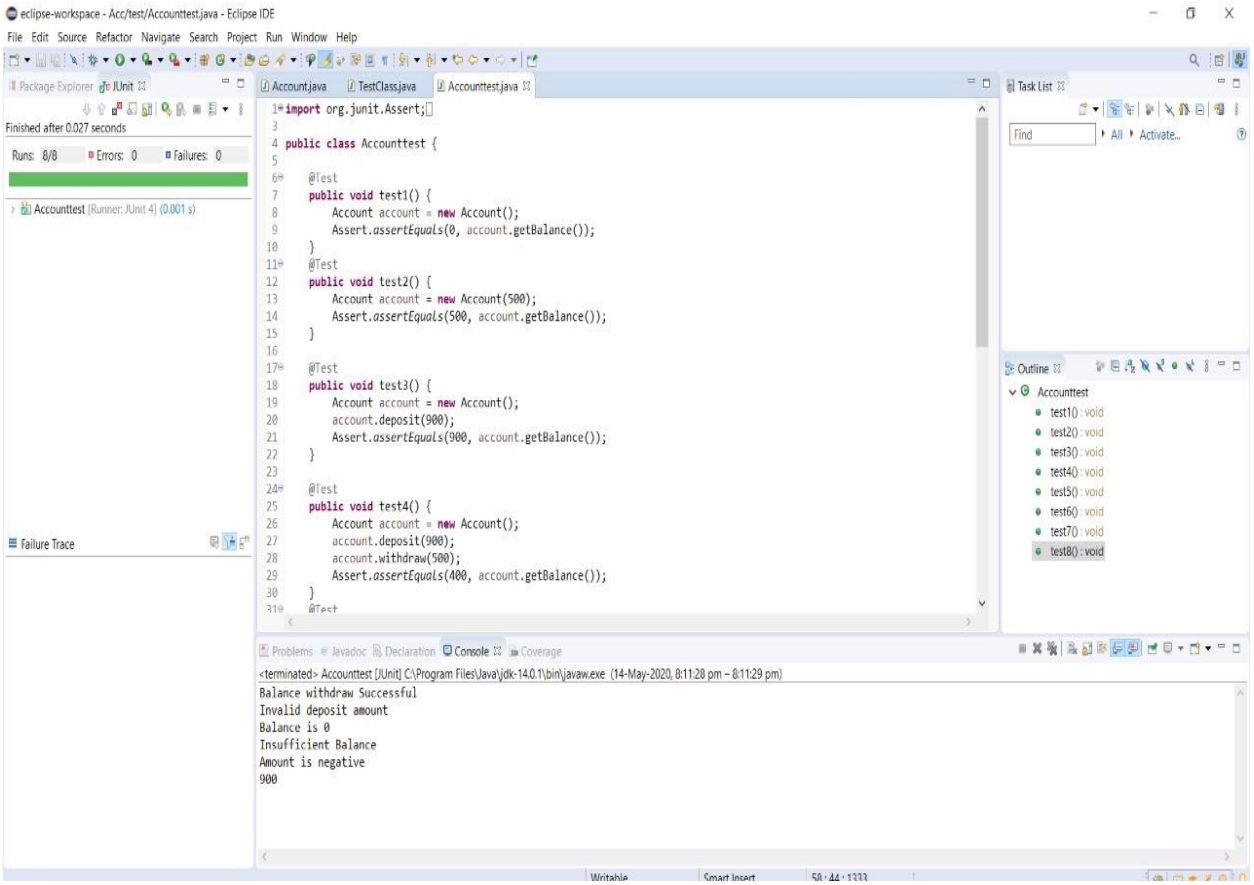
T7:- In transition t7 the transition moves from withdraw to deposit because if we try to withdraw greater than balance it will show "Insufficient Balance".

T8:- In transition t8 the transition moves from deposit to exit state, if(Withdraw Amount<0) then it will show "Amount is Negative" which is not possible.

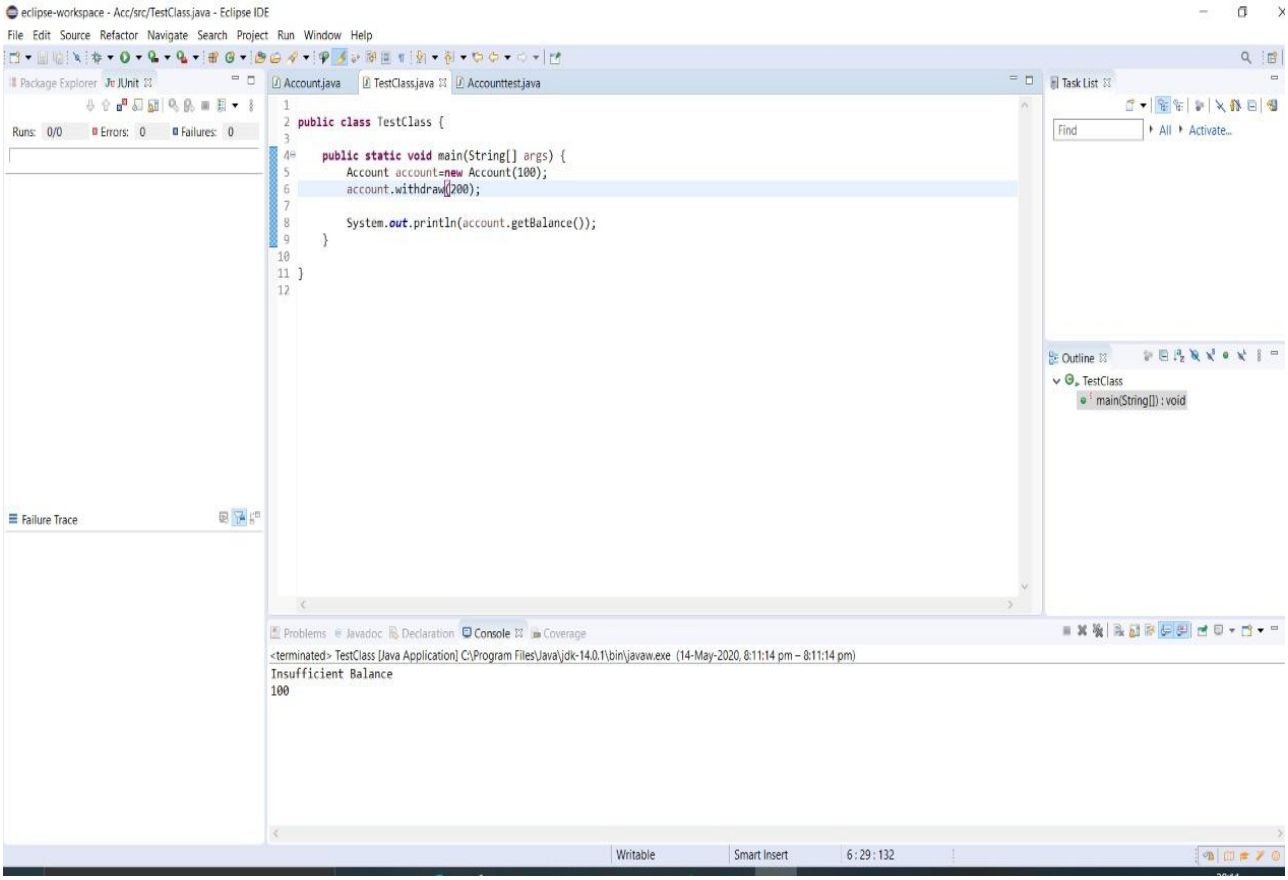
In the case study (Bank Account processing), we found that All Transition Pair with LN4 is the best for having more genuine test cases covering all the required transitions. All Transition, has been found not to be an adequate level of fault detection with more execution time compared to other coverage criteria and with larger test suite.

THE SNAPSHOTS OF OUR CODE ARE AS FOLLOWS:

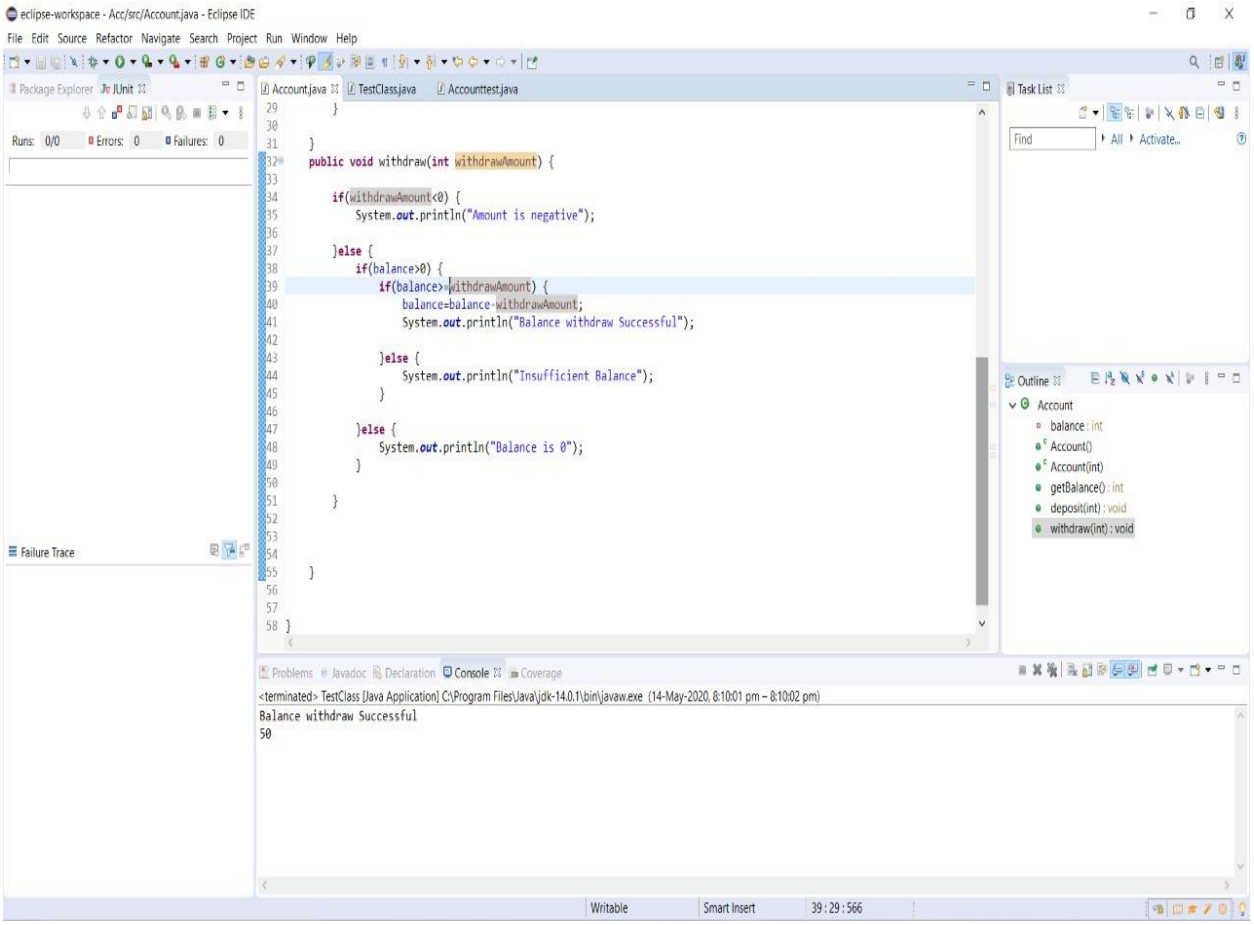
1.



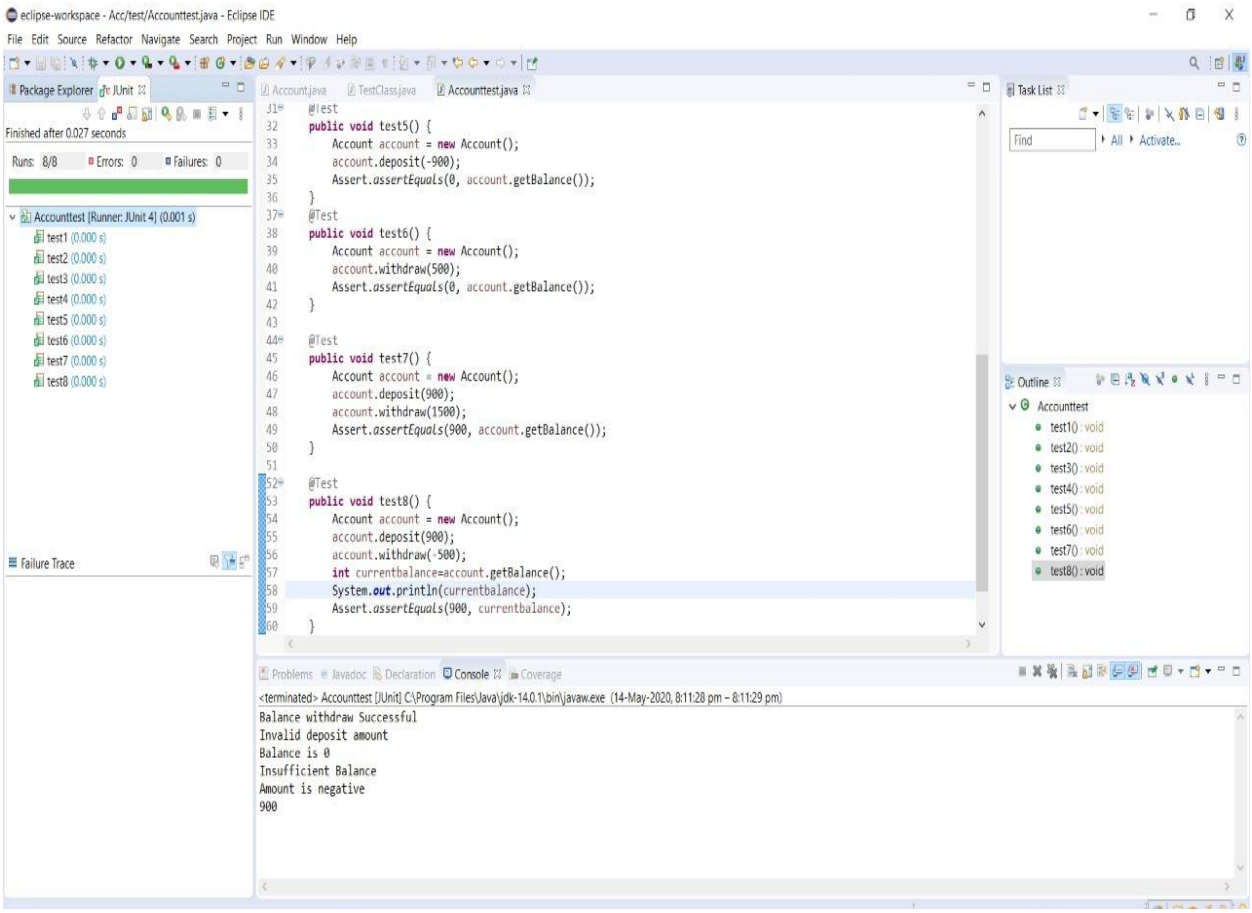
2.


eclipse-workspace - Acc/src/TestClass.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer
JUnit
Runs: 0/0
Errors: 0
Failures: 0
Failure Trace
Account.java
TestClass.java
Accounttest.java
1
2 public class TestClass {
3
4 public static void main(String[] args) {
5 Account account=new Account(100);
6 account.withdraw(200);
7
8 System.out.println(account.getBalance());
9 }
10
11 }
12
Task List
Find
All
Activate...
Outline
TestClass
main(String[]) : void
Problems
Javadoc
Declaration
Console
Coverage
<terminated> TestClass [Java Application] C:\Program Files\Java\jdk-14.0.1\bin\javaw.exe (14-May-2020, 8:11:14 pm – 8:11:14 pm)
Insufficient Balance
100
Writable
Smart Insert
6 : 29 : 132

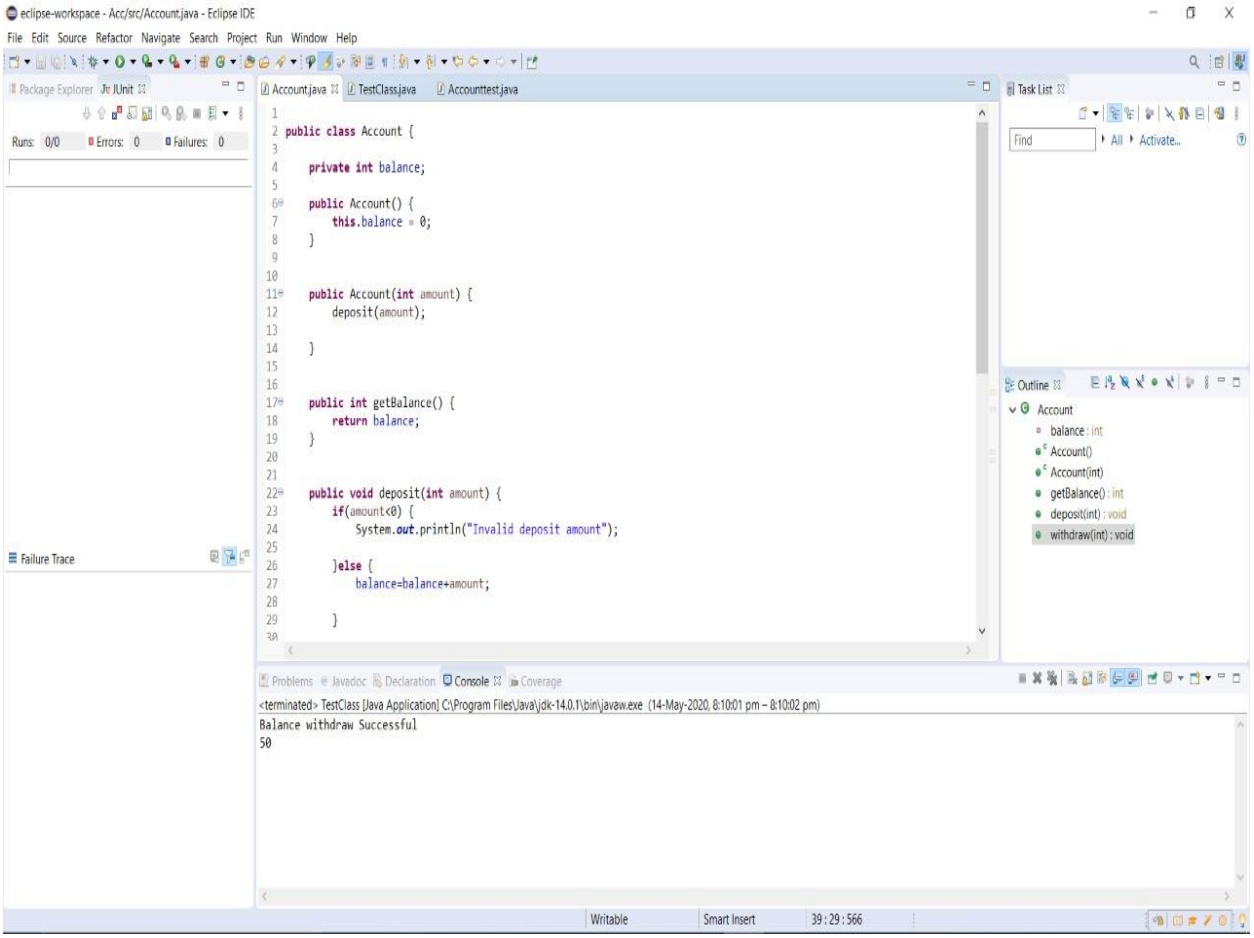
3.


eclipse-workspace - Acc/src/Account.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer JUnit
Runs: 0/0 Errors: 0 Failures: 0
Failure Trace
Account.java TestClass.java Accounttest.java
29 }
30
31 }
32 public void withdraw(int withdrawAmount) {
33
34 if(withdrawAmount<0) {
35 System.out.println("Amount is negative");
36
37 }else {
38 if(balance>0) {
39 if(balance>=withdrawAmount) {
40 balance=balance-withdrawAmount;
41 System.out.println("Balance withdraw Successful");
42
43 }else {
44 System.out.println("Insufficient Balance");
45 }
46
47 }else {
48 System.out.println("Balance is 0");
49 }
50
51 }
52
53
54
55 }
56
57
58 }
Task List
Find All Activate...
Outline
Account
balance : int
Account()
Account(int)
getBalance() : int
deposit(int) : void
withdraw(int) : void
Problems Javadoc Declaration Console Coverage
<terminated> TestClass [Java Application] C:\Program Files\Java\jdk-14.0.1\bin\javaw.exe (14-May-2020, 8:10:01 pm – 8:10:02 pm)
Balance withdraw Successful
50
Writable
Smart Insert
39 : 29 : 566

4.

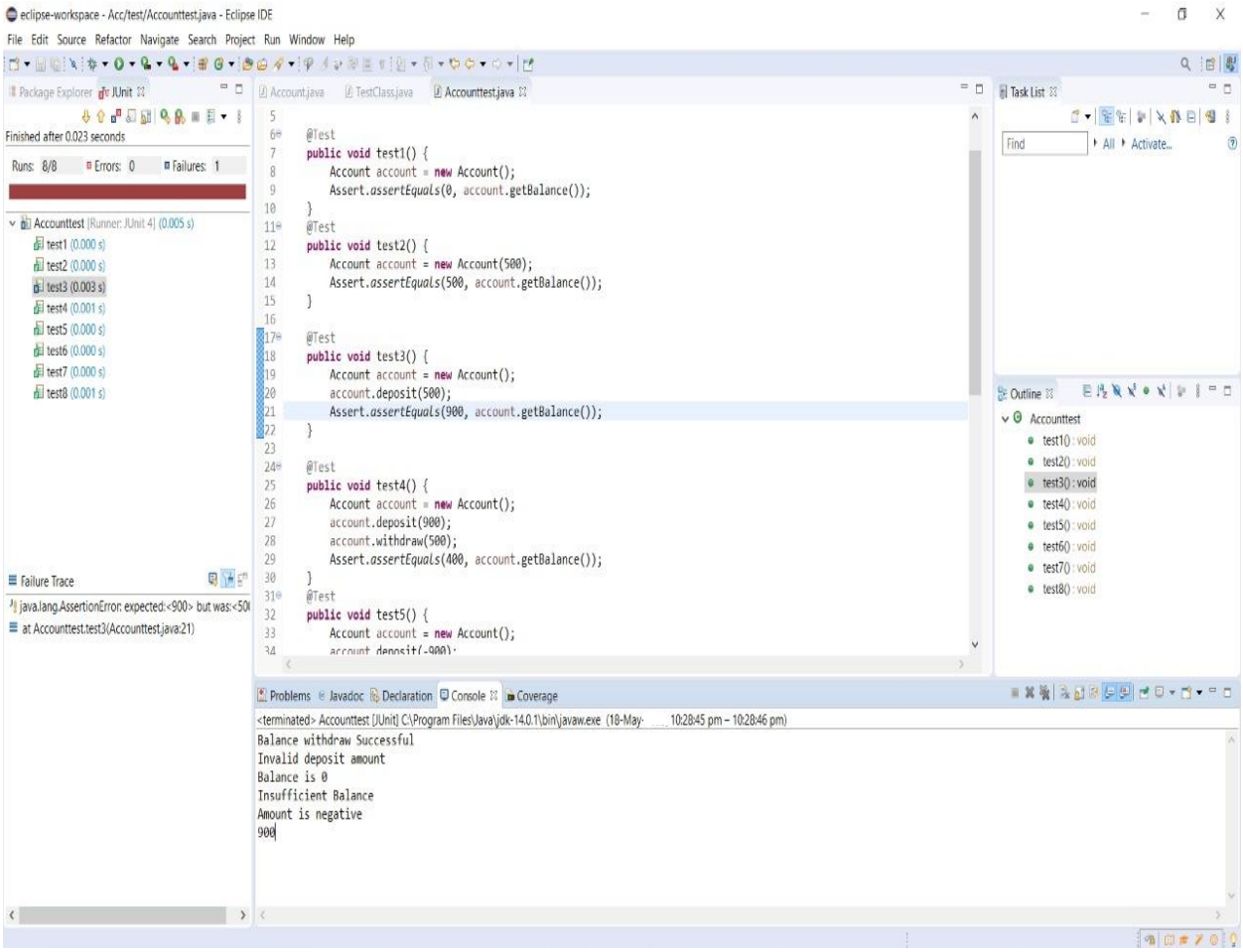

eclipse-workspace - Acc/test/Accounttest.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer JUnit
Finished after 0.027 seconds
Runs: 8/8 Errors: 0 Failures: 0
Accounttest [Runner: JUnit 4] (0.001 s)
test1 (0.000 s)
test2 (0.000 s)
test3 (0.000 s)
test4 (0.000 s)
test5 (0.000 s)
test6 (0.000 s)
test7 (0.000 s)
test8 (0.000 s)
Failure Trace
Account.java TestClass.java Accounttest.java
@Test
public void test5() {
Account account = new Account();
account.deposit(-900);
Assert.assertEquals(0, account.getBalance());
}
@Test
public void test6() {
Account account = new Account();
account.withdraw(500);
Assert.assertEquals(0, account.getBalance());
}
@Test
public void test7() {
Account account = new Account();
account.deposit(900);
account.withdraw(1500);
Assert.assertEquals(900, account.getBalance());
}
@Test
public void test8() {
Account account = new Account();
account.deposit(900);
account.withdraw(-500);
int currentbalance=account.getBalance();
System.out.println(currentbalance);
Assert.assertEquals(900, currentbalance);
}
Task List
Find All Activate...
Outline
Accounttest
test1() : void
test2() : void
test3() : void
test4() : void
test5() : void
test6() : void
test7() : void
test8() : void
Problems Javadoc Declaration Console Coverage
<terminated> Accounttest [JUnit] C:\Program Files\Java\jdk-14.0.1\bin\javaw.exe (14-May-2020, 8:11:28 pm – 8:11:29 pm)
Balance withdraw Successful
Invalid deposit amount
Balance is 0
Insufficient Balance
Amount is negative
900

5.

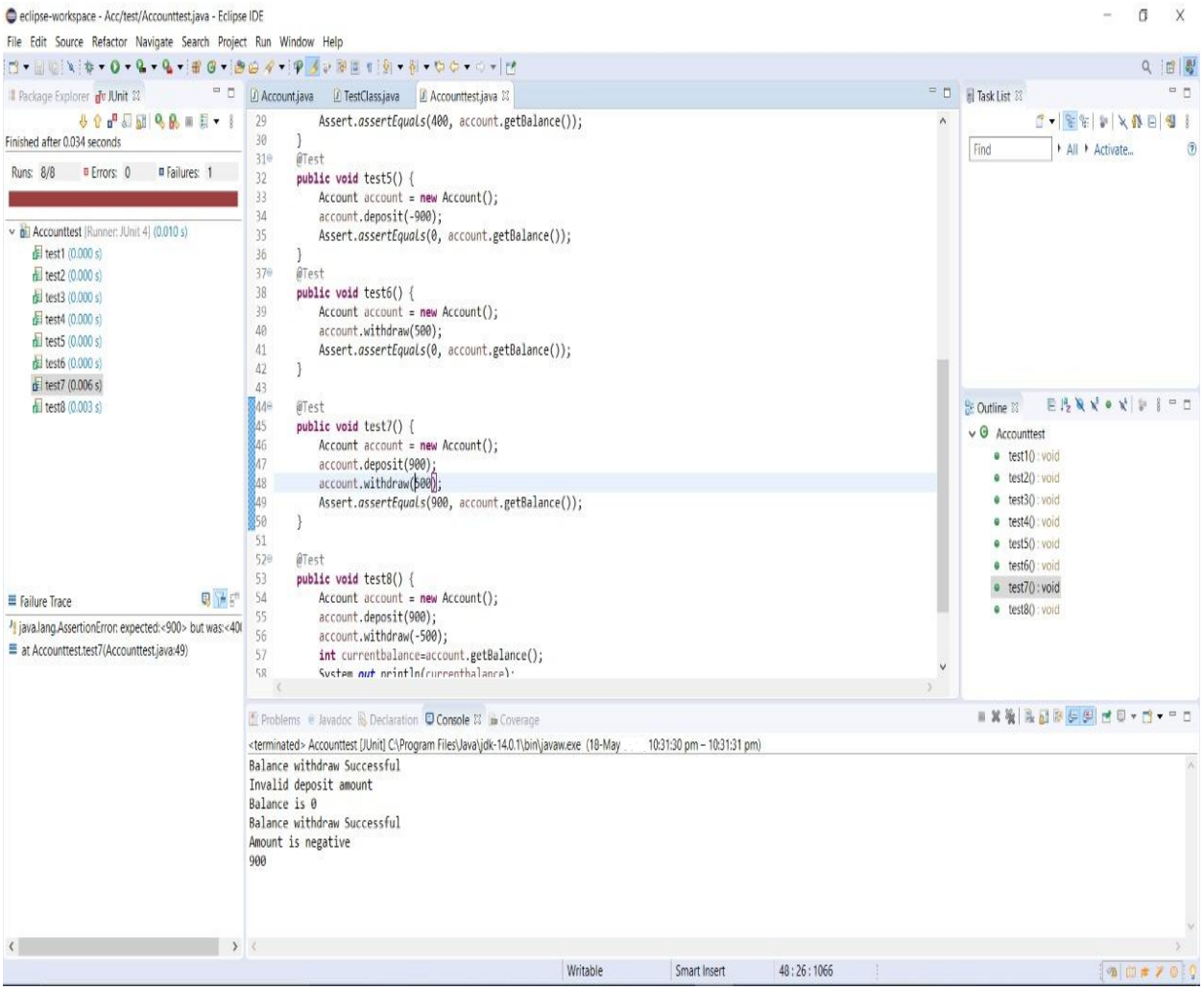

eclipse-workspace - Acc/src/Account.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer JUnit
Runs: 0/0 Errors: 0 Failures: 0
Failure Trace
Account.java TestClass.java Accounttest.java
1
2 public class Account {
3
4 private int balance;
5
6 public Account() {
7 this.balance = 0;
8 }
9
10
11 public Account(int amount) {
12 deposit(amount);
13
14 }
15
16
17 public int getBalance() {
18 return balance;
19 }
20
21
22 public void deposit(int amount) {
23 if(amount<0) {
24 System.out.println("Invalid deposit amount");
25
26 }else {
27 balance=balance+amount;
28
29 }
30
Task List
Find All Activate...
Outline
Account
balance : int
Account()
Account(int)
getBalance() : int
deposit(int) : void
withdraw(int) : void
Problems Javadoc Declaration Console Coverage
<terminated> TestClass [Java Application] C:\Program Files\Java\jdk-14.0.1\bin\javaw.exe (14-May-2020, 8:10:01 pm – 8:10:02 pm)
Balance withdraw Successful
50
Writable
Smart Insert
39 : 29 : 566

SNAPSHOT OF SOME TEST CASES WHICH ARE NOT PASSED

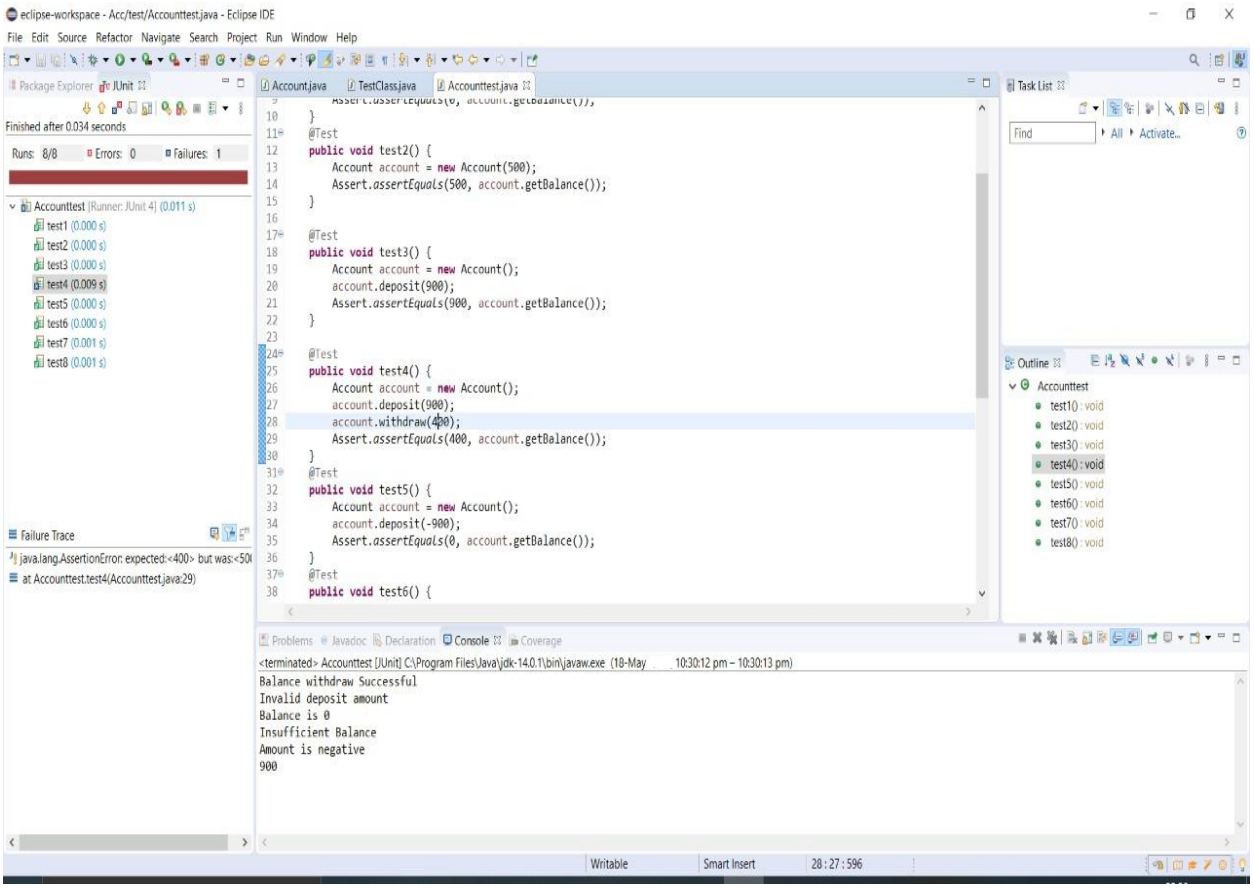
1.



2.


eclipse-workspace - Acc/test/Accounttest.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer JUnit
Finished after 0.034 seconds
Runs: 8/8 Errors: 0 Failures: 1
Accounttest [Runner: JUnit 4] (0.010 s)
test1 (0.000 s)
test2 (0.000 s)
test3 (0.000 s)
test4 (0.000 s)
test5 (0.000 s)
test6 (0.000 s)
test7 (0.006 s)
test8 (0.003 s)
Failure Trace
java.lang.AssertionError: expected:<900> but was:<40
at Accounttest.test7(Accounttest.java:49)
Account.java
TestClass.java
Accounttest.java
29 Assert.assertEquals(400, account.getBalance());
30 }
31 @Test
32 public void test5() {
33 Account account = new Account();
34 account.deposit(-900);
35 Assert.assertEquals(0, account.getBalance());
36 }
37 @Test
38 public void test6() {
39 Account account = new Account();
40 account.withdraw(500);
41 Assert.assertEquals(0, account.getBalance());
42 }
43
44 @Test
45 public void test7() {
46 Account account = new Account();
47 account.deposit(900);
48 account.withdraw(500);
49 Assert.assertEquals(900, account.getBalance());
50 }
51
52 @Test
53 public void test8() {
54 Account account = new Account();
55 account.deposit(900);
56 account.withdraw(-500);
57 int currentbalance=account.getBalance();
58 System.out.println(currentbalance);
Task List
Find All Activate...
Outline
Accounttest
test1() : void
test2() : void
test3() : void
test4() : void
test5() : void
test6() : void
test7() : void
test8() : void
Problems Javadoc Declaration Console Coverage
<terminated> Accounttest [JUnit] C:\Program Files\Java\jdk-14.0.1\bin\javaw.exe (18-May 10:31:30 pm – 10:31:31 pm)
Balance withdraw Successful
Invalid deposit amount
Balance is 0
Balance withdraw Successful
Amount is negative
900
Writable
Smart Insert
48 : 26 : 1066

3.



The test cases are generated to minimize cost as well as time. Unlike our approach, they concentrate on only one coverage criteria, all state-transition coverage, which requires more testing resources. We propose algorithms based on various coverage criteria. Out of those, RTP can detect various state-based faults with less number of test cases. They give more emphasis on state coverage rather than transitions and also don't pay attention to redundancy. We emphasize on avoiding redundant test cases with the generation of bound and simple test paths which is not considered in their approach.

CHAPTER - 5

CONCLUSION AND FUTURE WORK

Model based testing is not a new area of research, evaluating state-based testing approach is a challenging area in software testing. For SBT, the emphasis is to ensure efficient state-based fault detection within. We focus on generating test cases from the state chart diagram by implementing different coverage criteria to detect various state-based faults. We analyzed various state-based coverage criteria and observed that RTP is the best coverage criteria among them when the state change of the state machine is non-deterministic. Fault prediction is more in the case of coverage criteria, RTP, as compared to other coverage criteria. It is experimentally observed that the coverage criteria, AT, is most likely and not adequately reliable for an indicator of fault detection. The test suite preparation and execution time for ATP is more compared to other coverage criteria. The limitation is that we have not considered the event dependency graph. We only considered the state chart diagram. The event dependency graph is beneficial for a critical system and it is used to identify critical components, interfaces and subsystems.

In future,. The work can be extended to detect the faults related to unspecified behavior (sneak-paths) of the state machine and how other UML diagrams can be combined with state chart diagram to achieve more number of state-based faults. Evaluation of the state-based testing is the most challenging field in the software Engineering field . We have chosen the model based testing as it ensure efficient state based fault detection within a limited time.

REFERENCES

Abdurazik, A., & Offutt, J., 1999. Generating test cases from UML specifications (Master's thesis, George Mason University).

Alhir, S. S., 2002. Introduction to the Unified Modeling Language (UML). In Guide to Applying the UML, Springer New York, 1-11. Aljawarneh, S. A., Alawneh, A., & Jaradat, R., 2017. Cloud security engineering: Early stages of SDLC. Future Generation Computer Systems, 74, 385-392.

Antoniol, G., Briand, L. C., Di Penta, M., & Labiche, Y., 2002. A case study using the round-trip strategy for statebased class testing. In 13th International Symposium on Software Reliability Engineering, 2002. Proceedings, IEEE, 269-279. Bauer, B., & Odell, J., 2005. UML 2.0 and agents: how to build agent-based systems with the new UML standard, Engineering applications of artificial intelligence, 18(2), 141-157.

Bohme, M., Pham, V. T., & Roychoudhury, A., 2017. Coverage-based greybox fuzzing as Markov chain, IEEE Transactions on Software Engineering.

Briand, L. C., Labiche, Y., & Wang, Y., 2004. Using simulation to empirically investigate test coverage criteria based on state chart. In Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on IEEE, 86-95.

Burgueno, L., Vallecillo, A., & Gogolla, M., 2018. Teaching UML and OCL models and their validation to software engineering students: an experience report. Computer Science Education, 1-19.

Da Silva, A. R., 2015. Model-driven engineering: A survey supported by the unified conceptual model. Computer Languages, Systems & Structures, 43, 139-155.

El-Fakih, K., Simao, A., Jadoon, N., & Maldonado, J. C., 2017. An assessment of extended finite state machine test selection criteria. Journal of Systems and Software, 123, 106-118. Fitzgerald, B., & Stol, K. J., 2017. Continuous software engineering: A roadmap and agenda. Journal of Systems and Software, 123, 176-189.

H Hashim, N. L., Ibrahim, H. R., Rejab, M. M., Romli, R., & Mohd, H., 2018. An Empirical Evaluation of Behavioral UML Diagrams Based on the Comprehension of Test Case Generation. Advanced Science Letters, 24(10), 7257-7262. H Holt, N. E., Briand, L. C., & Torkar, R., 2014. Empirical evaluations on the cost-effectiveness of state-based testing: An industrial case study. Information and Software Technology, 56(8), 890-910.

Kabir, S., 2017. An overview of fault tree analysis and its application in model-based dependability analysis. Expert Systems with Applications, 77, 114-135.