Software Testing

Bank System

Computer Engineering and Software Systems

Junior level

Khaled Mohamed Kord 14p6032

Mina Wagdi Fikri 14p8053

Kyrillos Nagy 14p6018

Introduction :

Our project is about testing a bank software system.

This Bank System Software allows to Add Accounts , Withdraw money , Deposit Money , Transfer Money between two accounts and to Change money currency . (see Figure 1)

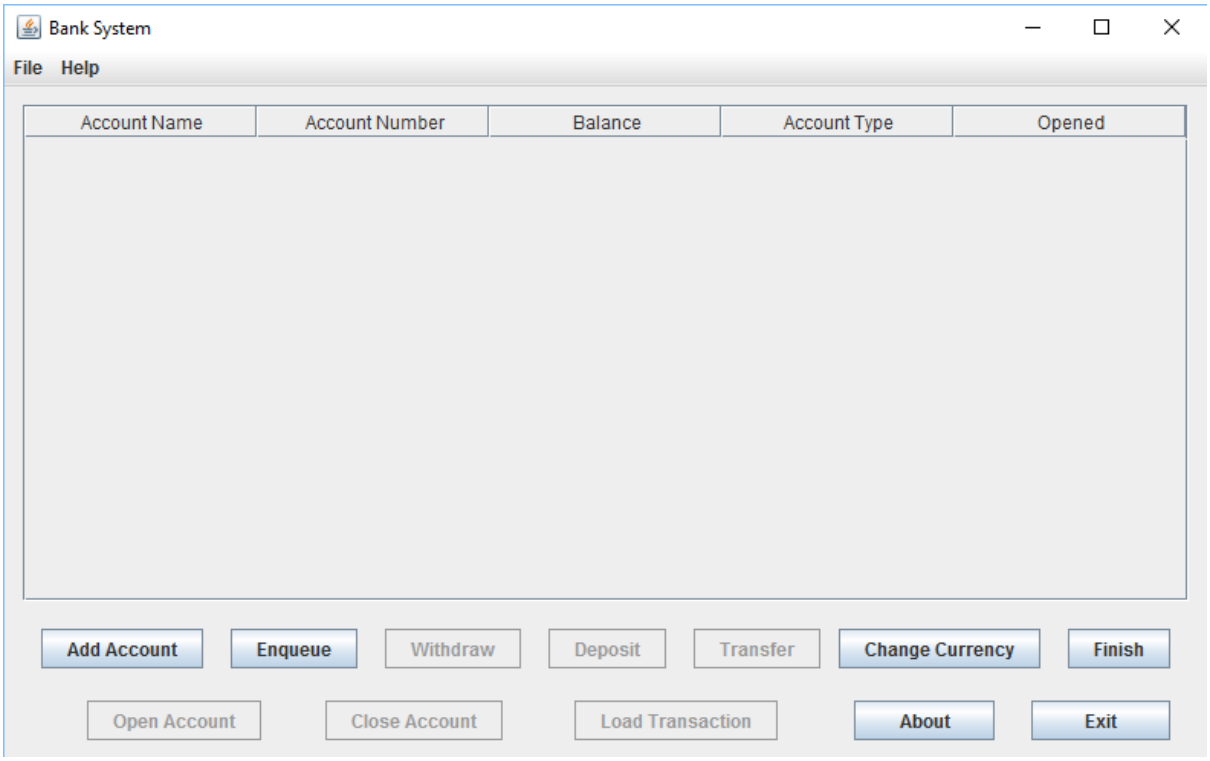


Figure 1

The Bank System consists of many classes and Frames that make the above functions able to happen. These classes are :

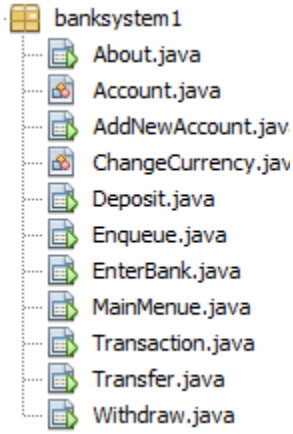


Figure 2

Bank Requirements

-User can withdraw, depose, change currency.

-user can transfer money into another users

- System is able to save data on text file with pre specified formats

- system can load data for already existed users.

Test requirements:

-test the functionalities of the user (the defined test cases)

-design test to assure better quality and efficiency

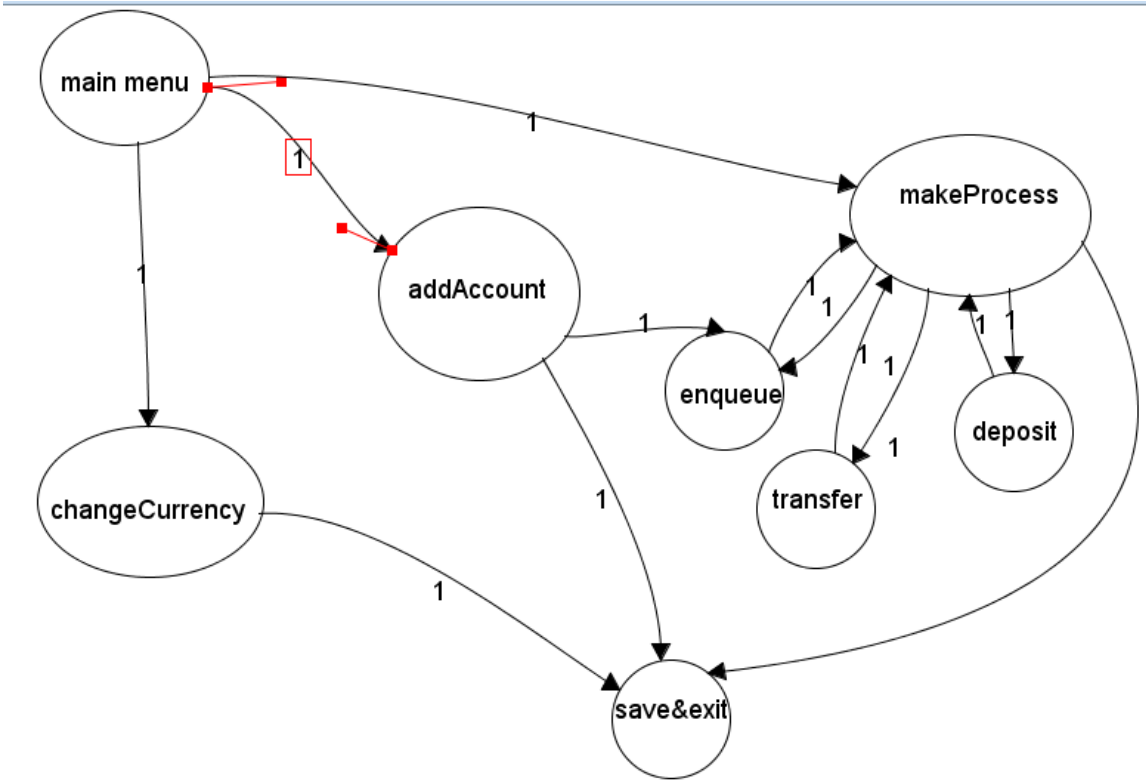
-test GUI.

Test Strategy:

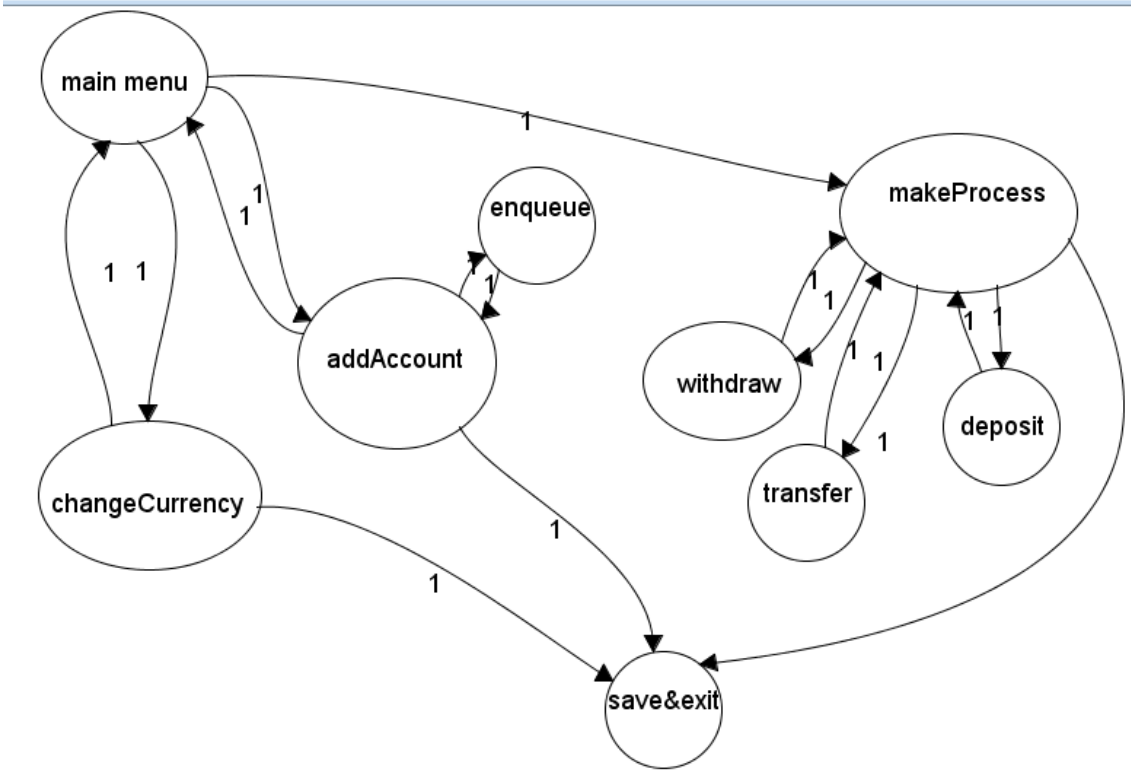
1- Unit Testing for AccountsSaver an AccountsLoader using drivers for each one of them (JUNIT)
2- After testing the AccountsSaver and the AccountsLoader , We tested the AccountAdder , Withdrawer , Depositer , CurrencyChanger and Transferer. (JUNIT)
3- Full System testing + GUI testing (using SIKULI)

Design test

We make a finite state machine representing the system



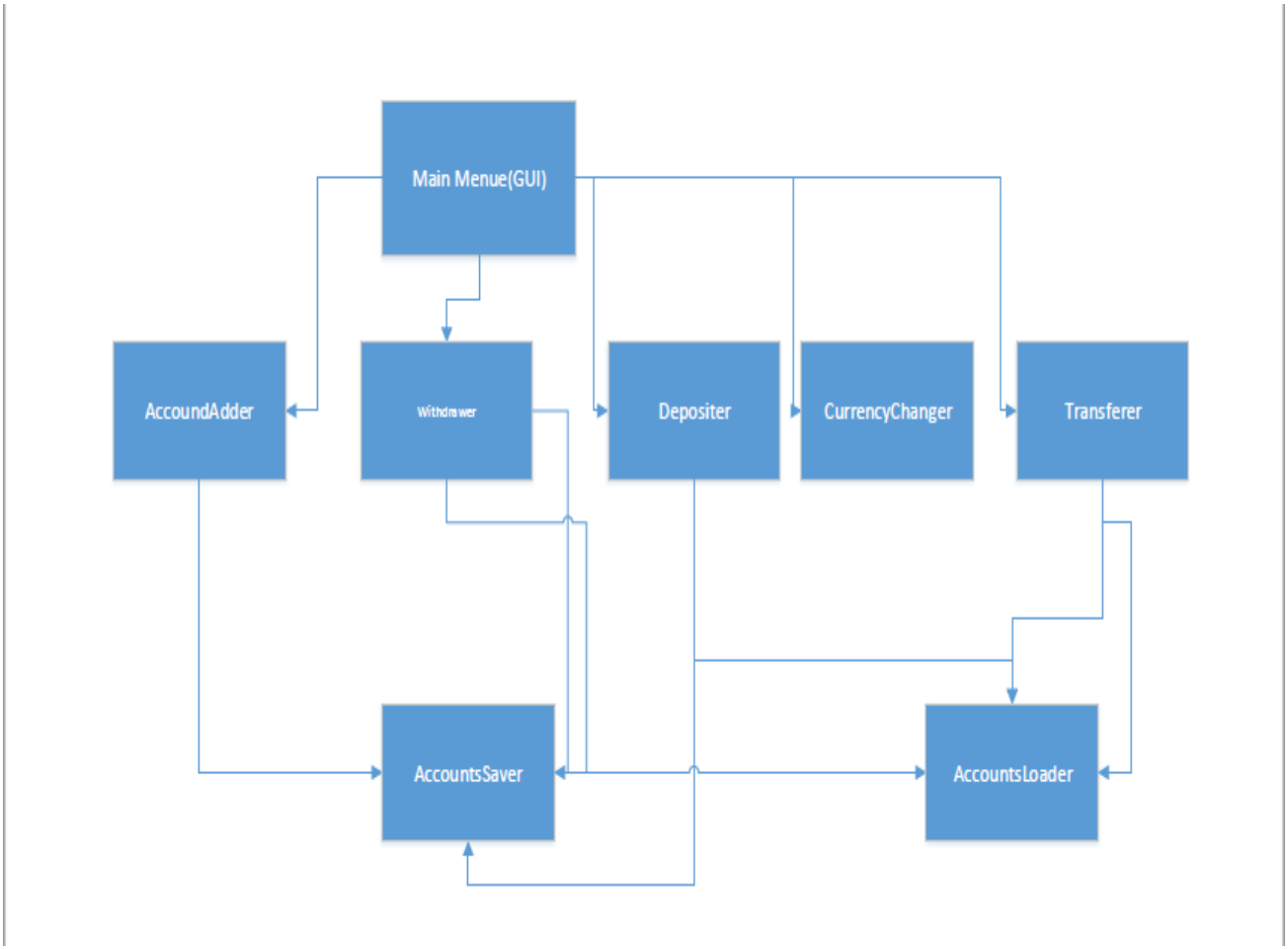
Applying state test we note that the minimum required test cases are 2 to test all states, this clearly wrong as from the requirements user should be able to do all he needed once then close, we solved this by adding to edges from addAccount and change currency back to main menu



Integration test

This course is a Software Testing course. So , our main concern is to test all the functionalities of the system.

To do that , we used Bottom-up strategy .(see Figure 3)



J Unit

LoadTest : we created 2 accounts instances from the Account class.

One with the information provided by us. (What is meant by information is the attributes : Account name , Account Number , Account passkey , account balance and Account type.) and the other with no information.

We write these information in a separate text file and the idea is to load the information from the file to the second account instance and assert that the two accounts have the same attributes values.

GOAL :

1- Test that that there will be no exceptions
2- Test that the information will be loaded correctly.
3- Test that No Exceptions found when loading an existing file
4- Test that there will be FileNotFoundException when loading a non-existing file.

```
@Before
public void setUp() throws FileNotFoundException, UnsupportedEncodingException
{
    acc1=new Account("j" , "s" , 2558 , "sad");
    acc2=new Account("j" , "s" , 2558 , "sad");
    acc2.name = "MINAWAGDIFIKRI";
    acc2.accNum = 2500;
    acc2.passKey = "123";
    acc2.balance = 5000.0;
    acc2.type= "Standard";
    acc2.opened = true;
}
```

```
@Test
public void TestLoad()
{
    acc1.accNum=2500;
    try {
        acc1.load();
        assertTrue(true);
    } catch (FileNotFoundException ex) {

    }
    assertEquals(acc2,acc1);
    acc1.accNum=66666;
    try {
        acc1.load();
    } catch (FileNotFoundException ex) {
        assertTrue(true);
    }
```

Save Test :

We created an account instance and saved it.
In the previous step , we tested the AccountLoader. So we will try to make a few changes to the account instance and then load it another time. We should find that the correct informations are loaded to proof that the AccountSaver is working fine.

GOALS :

1- Test the save functionality.
2- Test that there will be no exceptions found.

```
Account acc;
@Before
public void setUp ()
{
    try {
        acc = new Account("Mina" , "0" , 1000.0 , "TEST");
    } catch (Exception ex) {
        assertTrue(true);
    }
```

```
@Test
public void TestSave()
{
    try {
        try {
            acc.save();
        } catch (FileNotFoundException ex) {
            Logger.getLogger(SaveTest.class.getName()).log(Level.SEVERE, null, ex);
        } catch (UnsupportedEncodingException ex) {
            Logger.getLogger(SaveTest.class.getName()).log(Level.SEVERE, null, ex);
        }
        acc.name="askdjahskdj";
        acc.load();
        assertTrue(true);
        assertEquals("Mina" , acc.name);

    } catch (FileNotFoundException ex) {
        Logger.getLogger(SaveTest.class.getName()).log(Level.SEVERE, null, ex);
    }

}
```

Deposit Test :

At this test step , we test the deposit functionality. So , when we make a deposit operation , the account balance should increase.

Test Cases :

1- Deposit a normal value

2- Deposit a negative amount value
3- Test that no Exceptions occur.

```
public class DepositTest {

    Account acc1;

    @Before
    public void setUp() throws Exception
    {
        acc1=new Account("khaled" , "0" , 1000.0,"TestDeposit");
    }
    @Test
    public void TestDeposit() {
        try {
            acc1.deposit(1500.0);
            assertTrue(true);
        } catch (Exception ex) {
        }
        assertEquals(2500.0, acc1.getBalance());
        try {
            acc1.deposit(-6);
            assertTrue(true);
        } catch (Exception ex) {

        }
        assertEquals(2500.0, acc1.getBalance());
    }

}
```

Transferer Test :

Test the transfer function which will transfer an amount of money between two accounts.
Two account instances are created.

The following test cases are done for each of the two accounts.

Test cases :

1- Transfer a normal amount of money
2- Transfer a very big amount of money that doesn't exist in the sender account balance
3- Transfer a negative value of money

```
@Test
public void TestTransfer() throws Exception {
    acc1.transfer(acc2 , 100.0);
    acc2.transfer(acc1, 922);
    assertEquals(1822.0,acc1.getBalance());
    assertEquals(178.0,acc2.getBalance());

    acc1.transfer(acc2,1000000.0);
    assertEquals(1822.0,acc1.getBalance());
    assertEquals(178.0,acc2.getBalance());

    acc2.transfer(acc1, 100000000);
    assertEquals(1822.0,acc1.getBalance());
    assertEquals(178.0,acc2.getBalance());

    acc1.transfer(acc2, -666);
    assertEquals(1822.0,acc1.getBalance());
    assertEquals(178.0,acc2.getBalance());

    acc2.transfer(acc1,-9999);
    assertEquals(1822.0,acc1.getBalance());
    assertEquals(178.0,acc2.getBalance());

    acc1.transfer(acc3 , acc1.getBalance());
    assertEquals(1822.0,acc3.getBalance());
    acc2.transfer(acc3 , acc2.getBalance());
    assertEquals(2000.0,acc3.getBalance());

}
```

Withdrawer test :

Test that when the withdraw operation is done the account balance decreases by that certain amount.

Test Cases :

1- Withdraw a normal amount of money
2- Withdraw a Very big amount of money that doesn't exist in the account balance
3- Withdraw a negative amount value

```java
@Test
public void TestWithdraw() throws Exception
{
    acc.withdraw(100);
    assertEquals(900.0, acc.getBalance());
    acc.withdraw(150);
    assertEquals(750.0 , acc.getBalance());
    acc.withdraw(111111);
    assertEquals(750.0 , acc.getBalance());
    acc.withdraw(-111111);
    assertEquals(750.0 , acc.getBalance());
}
```

ChangeCurrencyTester :

Test that the money amount is changed from a specific currency to another correctly :

TestCases :

1- Change from all currencies to all currencies
2- Test changing negative amount value of currency

GUI Testing, sikuli:

from the Junit test we made sure that the functionalities required are met, now we need to make sure that GUI works fine, using sikul we test all the buttons make sure there are no exceptions and that these buttons do their requirments.

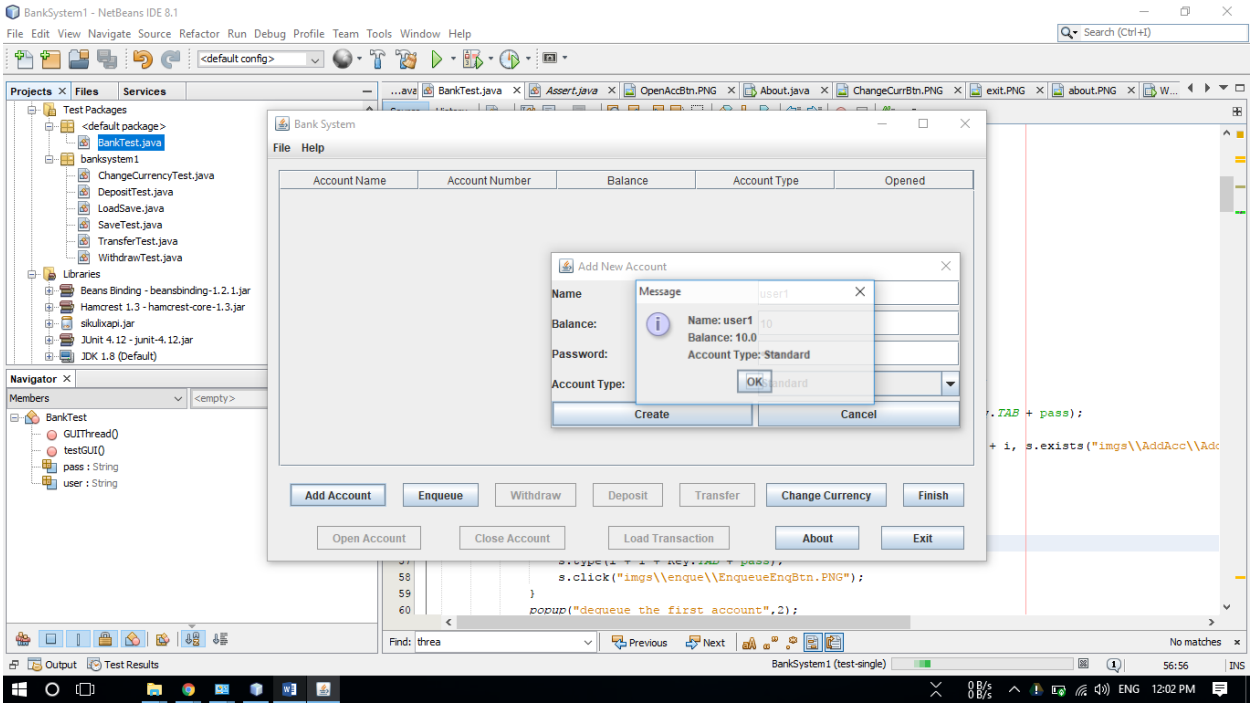
In the happy scenario we enter right data that tests all the functions and state

Screenshots of code

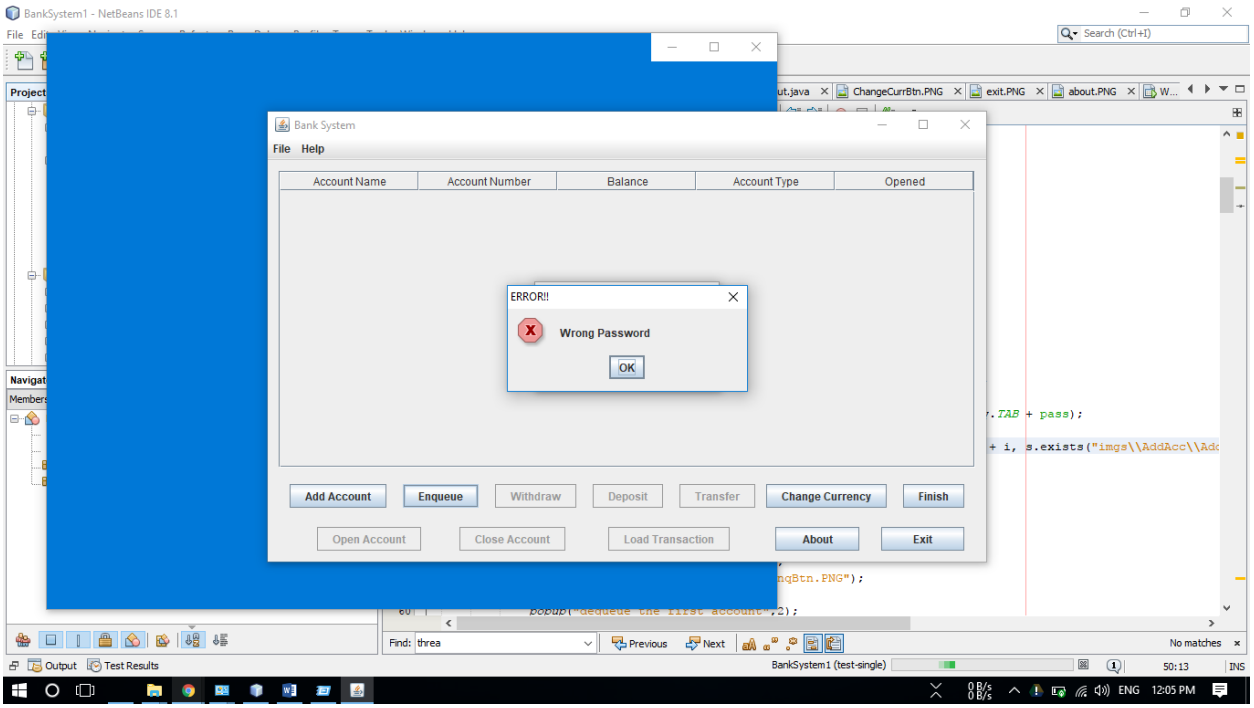
```
31      @Test
32      public void testGUI() {
33          try {
34
35              Screen s = new Screen();
36              MainMenue w = new MainMenue();
37              w.setVisible(true);
38              popup("begin testing\nI hope you enjoy :D ",3);
39              popup("The Happy scenario :D ");
40              // happy scenario :D
41              // create 5 accounts
42              popup("adding 5 accounts ");
43              for (int i = 0; i < 5; i++) {
44                  s.click("imgs\\AddAcc\\AddAccountBtn.PNG");
45
46                  s.type((user + (i + 1)) + Key.TAB + ("1" + (i * 200)) + Key.TAB + pass);
47                  s.click("imgs\\AddAcc\\AddAccCreateBtn.PNG");
48                  Assert.assertNotNull("error in creating Account when i = " + i, s.exists("imgs\\AddAcc\\A
49                  s.click("imgs\\okBtn.PNG");
50              }
51              //enqueue all 5 accounts
52              popup("enqueue 5 accounts ");
53              for (int i = 0; i < 5; i++) {
54                  s.click("imgs\\enque\\EnqueueBtn.PNG");
55                  s.type(i + 1 + Key.TAB + pass);
56                  s.click("imgs\\enque\\EnqueueEnqBtn.PNG");
57              }
58              popup("dequeue the first account");
59              //dequeue first account by clicking finish
60              s.click("imgs\\finish\\FinishBtn.PNG");
```

```
104             popup("begin Add account tests");
105             //----- begin AddAccount Jframe Test name, balance, pass empty or wrong --------------
106             s.click("imgs\\AddAcc\\AddAccountBtn.PNG");
107             s.type((user+"6") + Key.TAB + Key.TAB + pass);
108             s.click("imgs\\AddAcc\\AddAccCreateBtn.PNG");
109             //balance empty
110             Assert.assertNotNull("balance empty ", s.exists("imgs\\AddAcc\\balnceempty.PNG"));
111             s.click("imgs\\okBtn.PNG");
112             s.click("imgs\\AddAcc\\balance.PNG");
113             s.type("kero");
114             s.click("imgs\\AddAcc\\AddAccCreateBtn.PNG");
115             // not a number
116             Assert.assertNotNull("not a number ", s.exists("imgs\\AddAcc\\balancemustbenumber.PNG"));
117             s.click("imgs\\okBtn.PNG");
118             s.doubleClick("imgs\\AddAcc\\balance.PNG");
119             s.type(Key.BACKSPACE+"5000");
120             s.doubleClick("imgs\\AddAcc\\name.PNG");
121             s.type(Key.BACKSPACE);
122             s.click("imgs\\AddAcc\\AddAccCreateBtn.PNG");
123             // name empty
124             Assert.assertNotNull("name empty ", s.exists("imgs\\AddAcc\\nameempty.PNG"));
125             s.click("imgs\\okBtn.PNG");
126             s.type("imgs\\AddAcc\\name.PNG","user6");
127             s.doubleClick("imgs\\AddAcc\\pass.PNG");
128             s.type(Key.BACKSPACE);
129             s.click("imgs\\AddAcc\\AddAccCreateBtn.PNG");
130             // pass empty
131             Assert.assertNotNull("pass empty ", s.exists("imgs\\AddAcc\\passempty.PNG"));
132             s.click("imgs\\okBtn.PNG");
133             s.click("imgs\\AddAcc\\pass.PNG");
134             s.type(pass);
```

```
205
206             //-------------- begin deposite test ----------------------
207             popup("begin deposite tests");
208             s.click("imgs\\deposite\\DepositBtn.PNG");
209             s.type("kk");
210             s.click("imgs\\deposite\\depositedepoBtn.PNG");
211             // not number
212             Assert.assertNotNull("not number ", s.exists("imgs\\deposite\\number.PNG"));
213             s.click("imgs\\okBtn.PNG");
214             s.doubleClick("imgs\\deposite\\value.PNG");
215             s.type(Key.BACKSPACE +Key.BACKSPACE+ "0");
216             s.click("imgs\\deposite\\depositedepoBtn.PNG");
217             // zero
218             Assert.assertNotNull("zero ", s.exists("imgs\\deposite\\zero.PNG"));
219             s.click("imgs\\okBtn.PNG");
220             s.doubleClick("imgs\\deposite\\value.PNG");
221             s.type(Key.BACKSPACE +Key.BACKSPACE+ "-1000");
222             s.click("imgs\\deposite\\depositedepoBtn.PNG");
223             // -ve
224             Assert.assertNotNull("-ve ", s.exists("imgs\\deposite\\zero.PNG"));
225             s.click("imgs\\okBtn.PNG");
226             s.doubleClick("imgs\\deposite\\value.PNG");
227             s.type(Key.BACKSPACE +Key.BACKSPACE+ "5000");
228             s.click("imgs\\deposite\\depositedepoBtn.PNG");
229             //success
230             Assert.assertNotNull("success ", s.exists("imgs\\deposite\\success.PNG"));
231             s.click("imgs\\okBtn.PNG");
232             popup("finish deposite tests");
233             //----------------- end deposite test ---------------------
234
```



Then we test exceptions and errors(run to see the full cases)



Conclusion :

Finally , in this project , automated tools (Junit and Sikuli) are used to automate the testing operation and boost its effectiveness.

Bottom-up strategy is used :

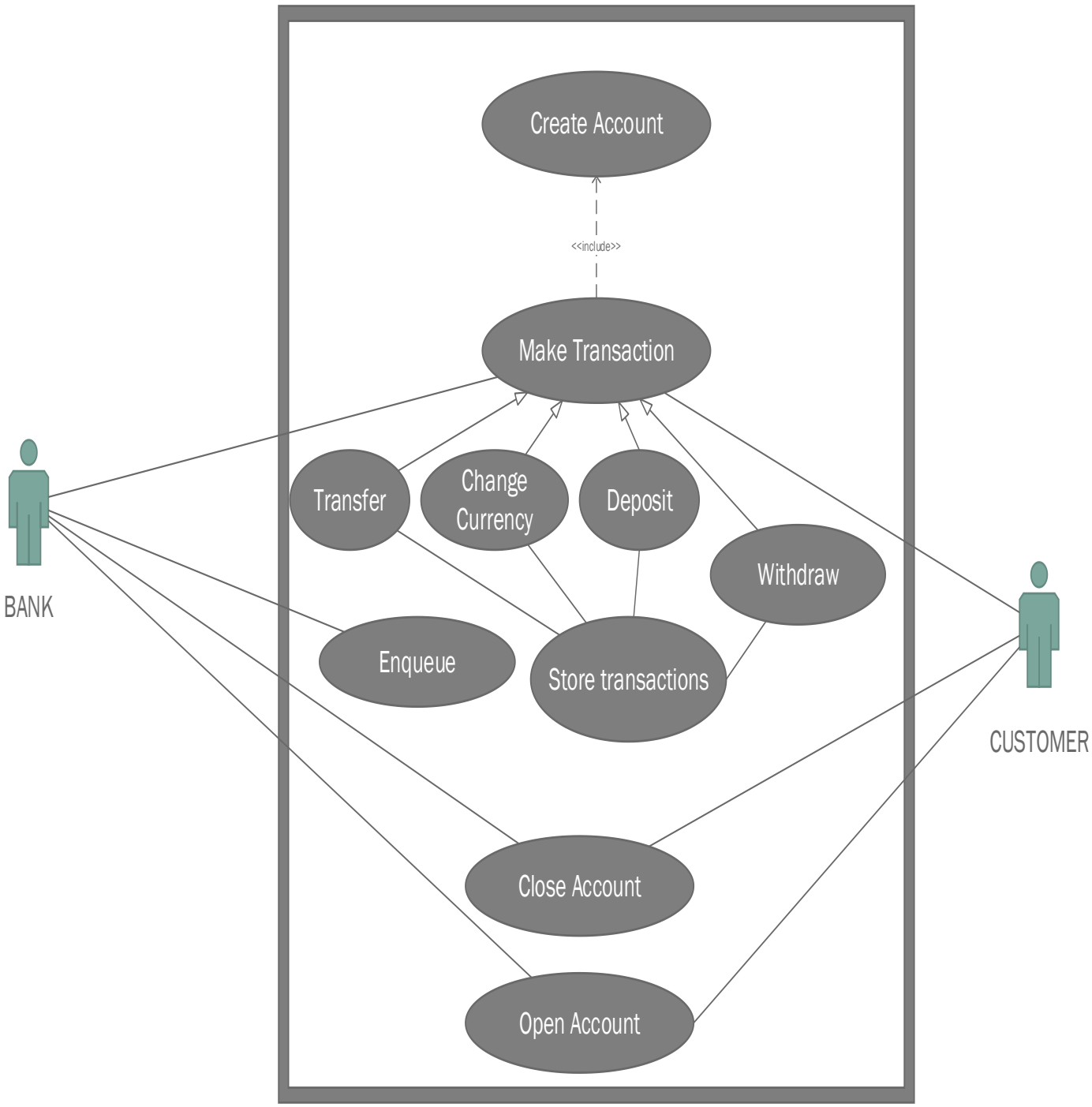
1- unit integration for every leaf component
2- Integration testing and unit testing for the above level components
3- GUI Testing (the highest level) using Sikuli

NOTE : All the components are tested from 2 different persepectives :

1- Functionality testing
2- Exception testing

Bottom-up strategy is used to make it easy to change the higher level components (like GUI) which takes a big part of the project.

Use Case


Create Account
<<include>>
Make Transaction
Transfer
Change Currency
Deposit
Withdraw
Enqueue
Store transactions
Close Account
Open Account
BANK
CUSTOMER