

# CSE338: Software Testing, Validation, and Verification

CESS Spring 2022 Major Task: Banking Application



Name	ID
Salma Ihab Abdelmawgoud	19P8794
Noorhan Hatem Ibrahim Mohamed	19P5821
Abdelhay Nader Abdelhay Abouzayed	18P9033
Serag Eldin Mohamed Ali Mohamed Hussein	19P1183
Mohamed Mohamed Saeed Mohamed	16p6015

## TABLE OF CONTENTS

<b>Introduction</b>	<b>2</b>
<b>Unit and Integration Testing</b>	<b>3</b>
Bank Class	3
Initializing Test and Object Classes	3
Test 1: addAccount and Integration with UserAccount deposit function	4
Test 2: Store Bank data in serializable	6
Item Class	8
Initializing Object Class Constructor	8
Test 1: Setters and Getters	8
Transaction Class	10
Test 1: Setters and Getters	10
User Account Class	13
Test 1: Setters and Getters	13
Test 2: Deposit and checkBalance	18
Test 3: Withdraw	20
Test 4: transferMoney	21
Test 5: Integration Test with Transaction class	23
Test 6: Integration Test with Item Class and purchases function	23
<b>GUI Testing</b>	<b>24</b>
<b>System Testing</b>	<b>27</b>
<b>Performance Testing</b>	<b>29</b>
Stress Testing	31
<b>Tools Used in Development</b>	<b>34</b>
<b>Closing Notes on Reuse and Refactoring</b>	<b>36</b>

# Introduction

Our team has decided upon a banking desktop application, developed fully using Java. The functional requirements are roughly for the user to be able to log in, sign in, and use his account: deposit, withdraw, transfer, and purchase items using a credit card. He can also check his balance and view his bank statement anytime. Some non-functional requirements include but not limited to: fast response time for actions, no crashing when unusual action is taken. All this is done through an intuitive GUI. The application must be developed using Test-driven development (TDD) methodology and with refactoring concepts in mind, and there must be unit, integration, system, performance tests, and optionally more, in this project.

Test-driven development is a style of programming in which three activities are done at once: coding and testing (through writing unit tests) and design (through refactoring code). The developer is forced to consider the requirements of how the functions should act, before coding them. Note that the unit tests will be automated.

## Unit and Integration Testing

TDD steps:

Step 1) Red: writing a test that fails

Step 2) Green: passing the test with as simple code as possible

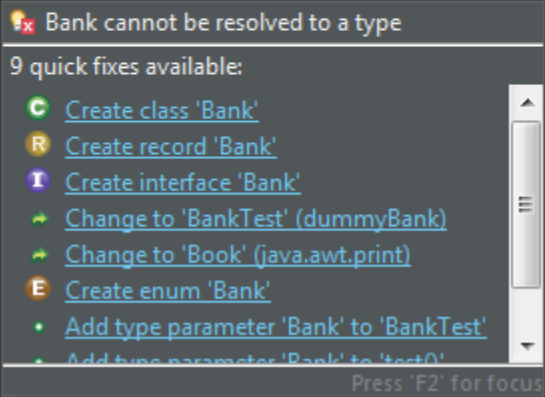
Step 3) Refactor: removing simple code and refactoring it to fit 'simple criteria'

The following are some screenshots showing the steps of writing our unit tests and basic application. We have 5 classes: Bank, Item, Transaction, UserAccount, CreditCard.

## Bank Class

### Initializing Test and Object Classes

```
7
8 public class BankTest {
9
10     @Before
11     public void setUp() throws Exception {
12     }
13
14     @Test
15     public void test() {
16         Bank bank = new Bank();
17     }
18 }
19
20
```




```
2
3 public class Bank {
4
5 }
6
```

Test 1: addAccount and Integration with UserAccount deposit function

**Step 1: test fails**

```
@Test
public void test() {
    Bank bank = new Bank();
    UserAccount user = new UserAccount();
    bank.addAccount(user);
}
```

 The method addAccount(UserAccount) is undefined for the type Bank


2 quick fixes available:

- [Create method 'addAccount\(UserAccount\)' in type 'Bank'](#)
- [Add cast to 'bank'](#)

Press 'F2' for focus

```
public class Bank {

    public void addAccount(UserAccount user) {
        // TODO Auto-generated method stub
        this.userlist.put(user.getUserName(),user);
    }
}
```

 userlist cannot be resolved or is not a field

2 quick fixes available:

- [Create field 'userlist' in type 'Bank'](#)
- [Create constant 'userlist' in type 'Bank'](#)

Press 'F2' for focus

```
@Test
public void test() {
    Bank bank = new Bank();
    UserAccount user = new UserAccount();
    user.setUserName("ahmed113");
    user.deposit(2000);
    bank.addAccount(user);
    assertEquals(2000, bank.getAccount("ahmed113").checkBalance());
}
}
```

The method getAccount(String) is undefined for the type Bank

3 quick fixes available:

- [Change to 'addAccount\(..\)'](#)
- [Create method 'getAccount\(String\)' in type 'Bank'](#)
- [Add cast to 'bank'](#)

Press 'F2' for focus

```
public class Bank {

    private Map<String,UserAccount> userlist;

    public void Bank() {
        userlist = new HashMap<>();
    }

    public void addAccount(UserAccount user) {
        // TODO Auto-generated method stub
        this.userlist.put(user.getUserName(),user);
    }

    public UserAccount getAccount(String string) {
        // TODO Auto-generated method stub
        return this.userlist.get(string);
    }

}
```

## Step 2: test passes

Runs: 1/1    Errors: 0    Failures: 0

Summary: Bank BankTest [Runner: JUnit 4.12.0.002]

test (0.002 s)

```
import static org.junit.Assert.*;

7
8 public class BankTest {
9
10     @Before
11     public void setUp() throws Exception {
12     }
13
14     @Test
15     public void test() {
16         Bank bank = new Bank();
17         UserAccount user = new UserAccount();
18         user.setUserName("ahmed113");
19         user.deposit(2000);
20         bank.addAccount(user);
21         assertEquals(2000, bank.getAccount("ahmed113").checkBalance(), 0);
22     }
23
24 }
25
```

## Test 2: Store Bank data in serializable

### Step 1: test fails

```
@Test
public void test2() {
    Bank bank = new Bank();
    UserAccount user = new UserAccount();
    Bank bank2= new Bank();
    user.setUserName("ahmed113");
    user.deposit(2000);
    bank.addAccount(user);
    bank.serializeDataOut();
    bank2.serializeDataIn();
}

public void serializeDataOut() throws Exception {
    // TODO Auto-generated method stub
    File fileOne=new File("userlist");
    FileOutputStream fos=new FileOutputStream(fileOne);
    ObjectOutputStream oos=new ObjectOutputStream(fos);

    oos.writeObject(this.userlist);

    oos.flush();
    oos.close();
    fos.close();
}

public void serializeDataIn() throws Exception {
    // TODO Auto-generated method stub
    File toRead=new File("userlist");
    FileInputStream fis=new FileInputStream(toRead);
    ObjectInputStream ois=new ObjectInputStream(fis);

    this.userlist=(HashMap<String,UserAccount>)ois.readObject();

    ois.close();
    fis.close();
}

@Test
public void test2() throws Exception {
    Bank bank = new Bank();
    UserAccount user = new UserAccount();
    Bank bank2= new Bank();
    user.setUserName("ahmed113");
    user.deposit(2000);
    bank.addAccount(user);
    bank.serializeDataOut();
    bank2.serializeDataIn();
    assertEquals(bank.getUserList(),bank2.getUserList());
}
```

### Step 2: test passes

```
30 import static org.junit.Assert.*;
7
8 public class BankTest {
9
10     @Before
11     public void setUp() throws Exception {
12     }
13
14     @Test
15     public void test() {
16         Bank bank = new Bank();
17         UserAccount user = new UserAccount();
18         user.setUserName("ahmed113");
19         user.deposit(2000);
20         bank.addAccount(user);
21         assertEquals(2000, bank.getAccount("ahmed113").checkBalance(), 0);
22     }
23
24     @Test
25     public void test2() throws Exception {
26         Bank bank = new Bank();
27         UserAccount user = new UserAccount();
28         Bank bank2 = new Bank();
29         user.setUserName("ahmed113");
30         user.deposit(2000);
31         bank.addAccount(user);
32         bank.serializeDataOut();
33         bank2.serializeDataIn();
34         assertEquals(2000, bank2.getAccount("ahmed113").checkBalance(), 0);
35     }
36 }
```

### Step 3: refactoring

```
30 import static org.junit.Assert.*;
7
8 public class BankTest {
9     Bank bank;
10     Bank bank2;
11     UserAccount user;
12
13     @Before
14     public void setUp() throws Exception {
15         bank = new Bank();
16         user = new UserAccount();
17         user.setUserName("ahmed113");
18         user.deposit(2000);
19         bank.addAccount(user);
20     }
21
22     @Test
23     public void test() {
24         assertEquals(2000, bank.getAccount("ahmed113").checkBalance(), 0);
25     }
26
27     @Test
28     public void test2() throws Exception {
29         bank2 = new Bank();
30         bank.serializeDataOut();
31         bank2.serializeDataIn();
32         assertEquals(2000, bank2.getAccount("ahmed113").checkBalance(), 0);
33     }
34 }
```



# Item Class

## Initializing Object Class Constructor

```
public class Item {  
    public Item(int i) {  
        // TODO Auto-generated  
        this.price=i;  
    }  
}
```

## Test 1: Setters and Getters

### Step 1: setName and getPrice Red

```
@Test  
public void test() {  
    Item item = new Item(1000);  
    item.setName("Pillow");  
}
```

The method setName(String) is undefined for the type Item

2 quick fixes available:

- Create method 'setName(String)' in type 'Item'
- Add cast to 'item'

Press 'F2' for focus

```
}  
  
@Test  
public void test() {  
    Item item = new Item(1000);  
    item.setName("Pillow");  
    assertEquals(1000, item.getPrice(), 0);  
}
```

The method getPrice() is undefined for the type Item

2 quick fixes available:

- Create method 'getPrice()' in type 'Item'
- Add cast to 'item'

Press 'F2' for focus

### Step 2: setName and getPrice Green

```

3 public class Item {
4
5     private double price;
6     private String name;
7
8     public Item(double i) {
9         // TODO Auto-generated constructor stub
10        this.price=i;
11    }
12
13    public void setName(String string) {
14        // TODO Auto-generated method stub
15        this.name=string;
16    }
17
18    public double getPrice() {
19        // TODO Auto-generated method stub
20        return this.price;
21    }
22
23
24

```

### Step 1: getName Red

```

}

@Test
public void test() {
    Item item = new Item(1000);
    item.setName("Pillow");
    assertEquals(1000,item.getPrice(),0);
    assertEquals("Pillow",item.getName());
}

```

 The method getName() is undefined for the type Item

3 quick fixes available:

-  [Change to 'getPrice\(..\)'](#)
-  [Create method 'getName\(\)' in type 'Item'](#)
-  [Add cast to 'item'](#)

Press 'F2' for focus

## Step 2: getName Green

```
3 public class Item {
4
5     private double price;
6     private String name;
7
8     public Item(double i) {
9         // TODO Auto-generated constructor stub
10        this.price=i;
11    }
12
13    public void setName(String string) {
14        // TODO Auto-generated method stub
15        this.name=string;
16    }
17
18    public double getPrice() {
19        // TODO Auto-generated method stub
20        return this.price;
21    }
22
23    public String getName() {
24        // TODO Auto-generated method stub
25        return this.name;
26    }
27
28 }
29
30 }
```

**Note:** sometimes, the refactoring step is unnecessary due to risk of over abstraction of code. In this example, we are just writing setters and getters to demonstrate how TDD may work. In the more complicated classes, we have made sure to refactor the functions to be more abstract and suitable for reuse.

## Transaction Class

### Test 1: Setters and Getters

#### Step 1: Red

```

1 public class TransactionTest {
2
3     @Before
4     public void setUp() throws Exception {
5     }
6
7     @Test
8     public void test() {
9         Transaction t = new Transaction();
10        t.setDescription();
11        t.setType();
12        t.setAmount();
13        t.getDescription();
14        t.getType();
15        t.getAmount();
16    }
17 }

```

## Step 2: writing functions to pass + green test

```

5 public class Transaction {
6     enum TransactionType{
7         DEPOSIT,WITHDRAWAL,PURCHASE;
8     }
9
10    private String description;
11    private TransactionType type;
12    private double amount;
13
14    public void setDescription(String s) {
15        // TODO Auto-generated method stub
16        this.description=s;
17    }
18
19
20    public void setType(int i) {
21        // TODO Auto-generated method stub
22        switch(i) {
23            case 0:
24                this.type=TransactionType.DEPOSIT;
25                break;
26            case 1:
27                this.type=TransactionType.WITHDRAWAL;
28                break;
29            case 2:
30                this.type=TransactionType.PURCHASE;
31                break;
32        }
33    }
34
35
36    }
37
38    public void setAmount(double i) {
39        // TODO Auto-generated method stub
40        this.amount=i;
41    }
42
43    }

```

```
dummyBank.TransactionTest [Runner: JUnit 4] (0.000 s)

8 public class TransactionTest {
9
10     @Before
11     public void setUp() throws Exception {
12     }
13
14     @Test
15     public void test() {
16         Transaction t = new Transaction();
17         t.setDescription("PAYROLL");
18         t.setType(0);
19         t.setAmount(100);
20         assertEquals("PAYROLL", t.getDescription());
21         assertEquals("DEPOSIT", t.getType().toString());
22         assertEquals(100, t.getAmount(), 0);
23     }
24 }
25
26
```

Step 2: writing function to pass

```
5
6
7     public Date getDate() {
8         // TODO Auto-generated method stub
9         return this.date;
10    }
11
12
13
```

Step 2: Green

```
@Test
public void test() {
    Transaction t = new Transaction();
    t.setDescription("PAYROLL");
    t.setType(0);
    t.setAmount(100);
    assertEquals("PAYROLL", t.getDescription());
    assertEquals("DEPOSIT", t.getType().toString());
    assertEquals(100, t.getAmount(), 0);
    assertEquals(new Date(), new Transaction().getDate());
}
```

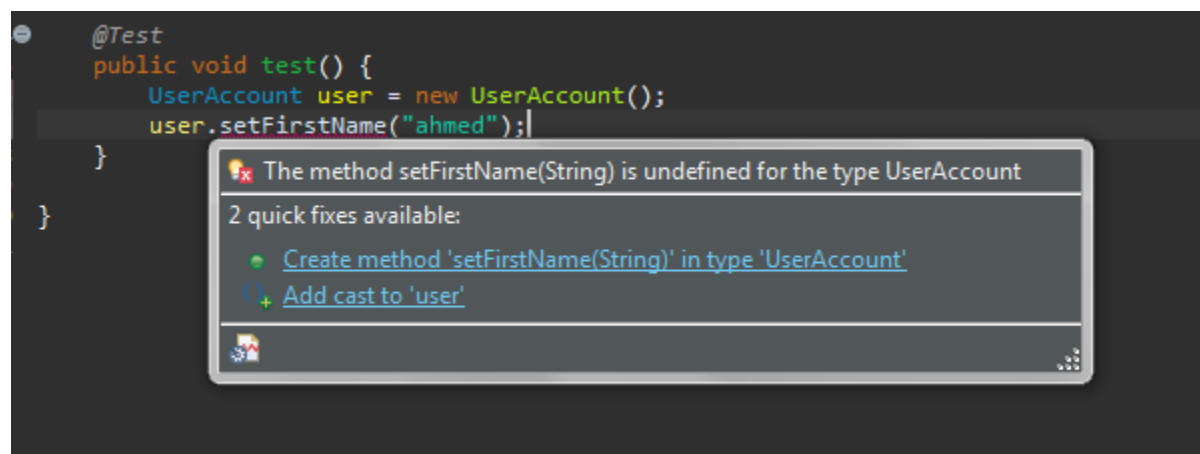
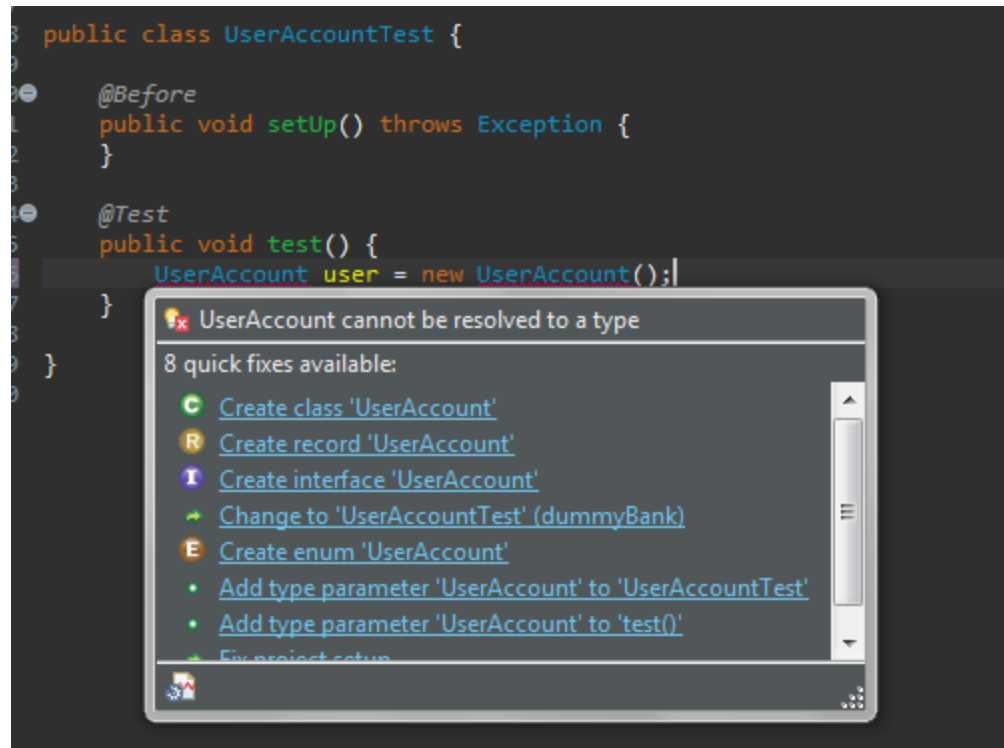
Step 3: refactoring passing Transaction functions (moving them to a more suitable class aka UserAccount)

```
32 }
33
34 @Test
35 public void test2() {
36     assertEquals(1000,user.checkBalance(),0);
37     user.deposit(102.6);
38     assertEquals(1102.6,user.checkBalance(),0);
39 }
40 @Test
41 public void test3() {
42     user.withdraw(500);
43     assertEquals(500,user.checkBalance(),0);
44     user.withdraw(50.1);
45     assertEquals(449.9,user.checkBalance(),0);
46 }
47 @Test
48 public void test4() {
49     UserAccount user2 = new UserAccount();
50     user2.setUserName("mohamed123");
51     bank = new Bank();
52     bank.addAccount(user2);
53     user.transferMoney(bank,"mohamed123",500);
54     assertEquals(500,user.checkBalance(),0);
55     assertEquals(500,user2.checkBalance(),0);
56 }
57 @Test
58 public void test5() {
59     List<Transaction> t = new ArrayList<Transaction>();
60     Transaction e = new Transaction();
61     e.setAmount(1000);
62     e.setDescription("NONE");
63     e.setType(0);
64     t.add(e);
65
66     assertEquals(t.get(0).getType(),user.getStatement().get(0).getType());
67 }
68 }
```

## User Account Class

### Test 1: Setters and Getters

#### Step 1: Red test class



Step 2: writing simple code into object class to pass test

```
public class UserAccount {
    public void setFirstName(String string) {
        // TODO Auto-generated method stub
    }
}
```

```
public class UserAccount {
    public void setFirstName(String string) {
        // TODO Auto-generated method stub
        this.firstName=string;
    }
}
```

firstName cannot be resolved or is not a field

1 quick fix available:

- Create field 'firstName' in type 'UserAccount'

### Step 1: Red getter

```
@Test
public void test() {
    UserAccount user = new UserAccount();
    user.setFirstName("ahmed");
    assertEquals("ahmed", user.getFirstName());
}
```

The method getFirstName() is undefined for the type UserAccount

2 quick fixes available:

- Create method 'getFirstName()' in type 'UserAccount'
- Add cast to 'user'

Press 'F2' for focus

### Step 2: writing getter to pass + Green



```

public class UserAccount {

    private String firstName;

    public void setFirstName(String string) {
        // TODO Auto-generated method stub
        this.firstName=string;
    }

    public String getFirstName() {
        // TODO Auto-generated method stub
        return this.firstName;
    }

}

```

Finished after 0.041 seconds

Runs: 1/1    ❌ Errors: 0    ❌ Failures: 0

dummyBank.UserAccountTest [Runner: JUnit 4] (0.002 s)

test (0.002 s)

```

2
3 import static org.junit.Assert.*;
7
8 public class UserAccountTest {
9
10    @Before
11    public void setUp() throws Exception {
12
13
14    @Test
15    public void test() {
16        UserAccount user = new UserAccount();
17        user.setFirstName("ahmed");
18        assertEquals("ahmed",user.getFirstName());
19    }
20
21 }
22

```

### Step 1: Red setters

```

public class UserAccountTest {

    @Before
    public void setUp() throws Exception {
    }

    @Test
    public void test() {
        UserAccount user = new UserAccount();
        user.setFirstName("ahmed");
        assertEquals("ahmed",user.getFirstName());
        user.setLastName("Mohamed");
        user.setUserName("ahmed113");
        user.setPassword("123456");
    }

}

```

### Step 2: Green setters + Step 1: Red getters

```

public class UserAccountTest {

    @Before
    public void setUp() throws Exception {
    }

    @Test
    public void test() {
        UserAccount user = new UserAccount();
        user.setFirstName("ahmed");
        assertEquals("ahmed",user.getFirstName());
        user.setLastName("Mohamed");
        user.setUserName("ahmed113");
        user.setPassword("123456");
        assertEquals("Mohamed",user.getLastName());
        assertEquals("ahmed113",user.getUserName());
        assertEquals("123445",user.getPassword());
    }

}

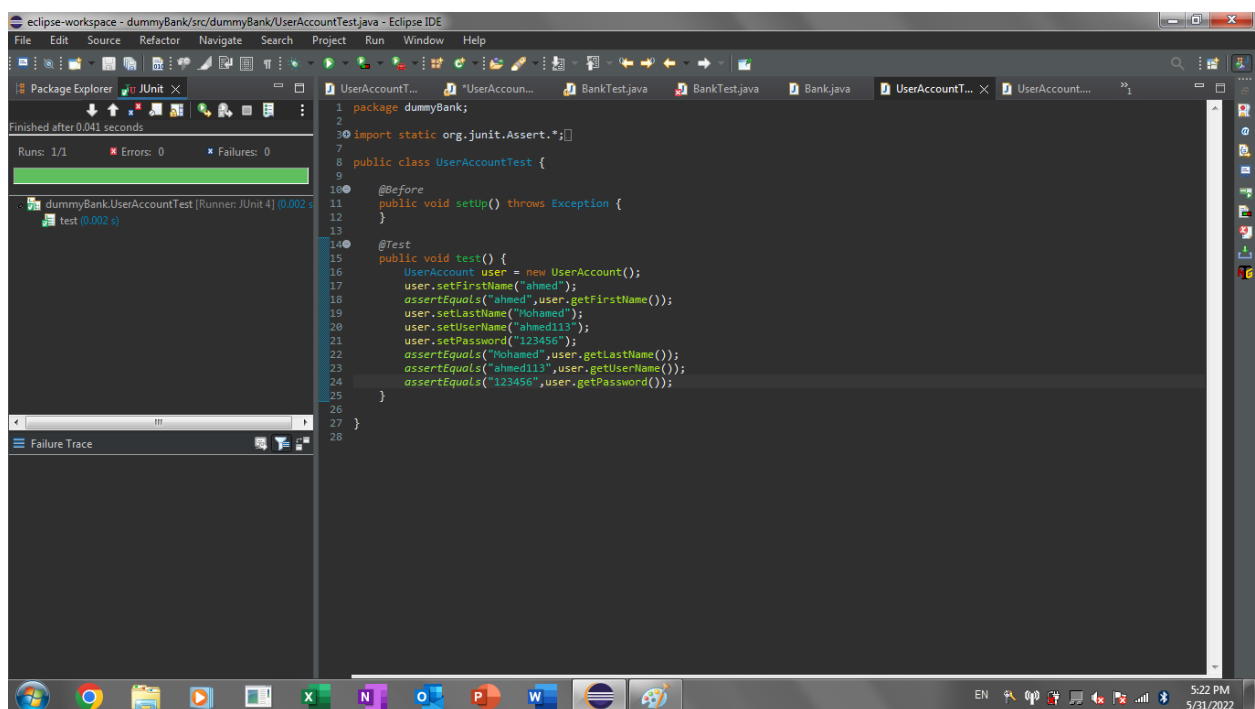
```

**Step 2: writing getters + Green getters assertions**

```

36     }
37
38     public String getLastName() {
39         // TODO Auto-generated method stub
40         return this.lastName;
41     }
42
43     public String getUsername() {
44         // TODO Auto-generated method stub
45         return this.userName;
46     }
47
48     public String getPassword() {
49         // TODO Auto-generated method stub
50         return this.password;
51     }
52
53 }

```



## Test 2: Deposit and checkBalance

### Step 1: Red deposit test

```
@Test
public void test2() {
    UserAccount user = new UserAccount();
    user.setFirstName("ahmed");
    user.setLastName("Mohamed");
    user.setUserName("ahmed113");
    user.setPassword("123456");
    user.deposit(1000);
}
}
```

The method deposit(int) is undefined for the type UserAccount

2 quick fixes available:

- Create method 'deposit(int)' in type 'UserAccount'
- Add cast to 'user'

Press 'F2' for focus

## Step 2: writing deposit function

```
public void deposit(int i) {
    // TODO Auto-generated method stub
    this.balance+=i;
}
}
```

balance cannot be resolved or is not a field

1 quick fix available:

- Create field 'balance' in type 'UserAccount'

## Step 1: Red checkBalance function

```
@Test
public void test2() {
    UserAccount user = new UserAccount();
    user.setFirstName("ahmed");
    user.setLastName("Mohamed");
    user.setUserName("ahmed113");
    user.setPassword("123456");
    user.deposit(1000);
    assertEquals(1000, user.checkBalance());
}
}
```

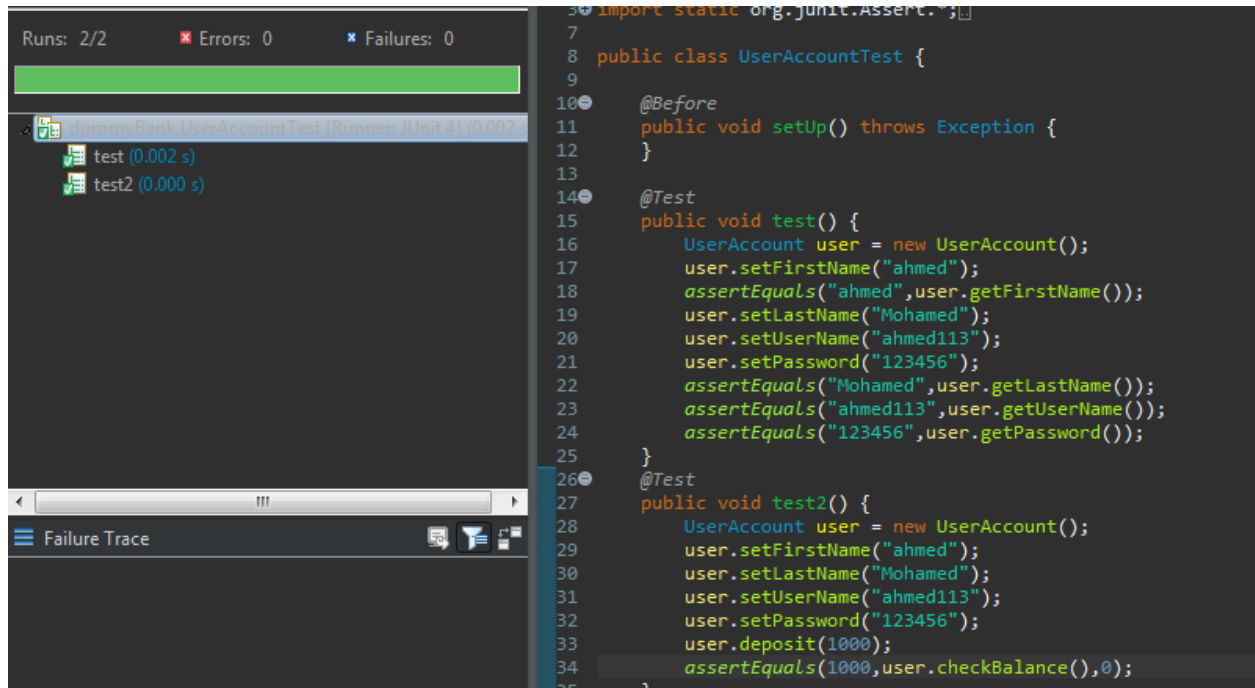
The method checkBalance() is undefined for the type UserAccount

2 quick fixes available:

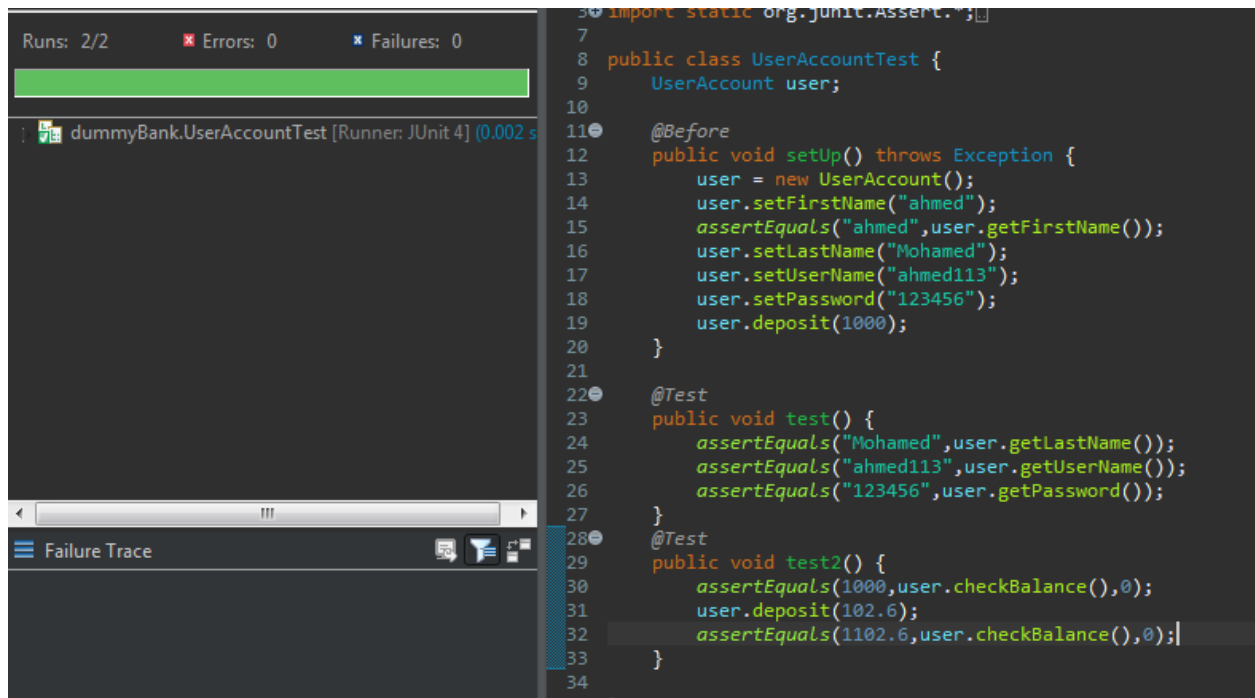
- Create method 'checkBalance()' in type 'UserAccount'
- Add cast to 'user'

Press 'F2' for focus

## Step 2 and 3: Green checkBalance test + separating (refactoring) assert functions



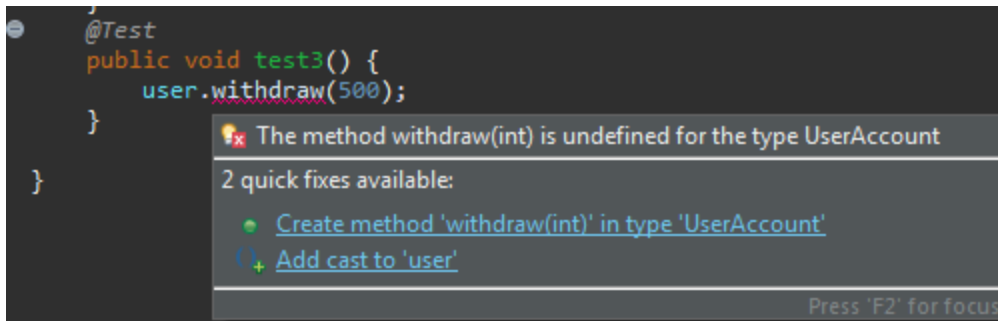
```
30 import static org.junit.Assert.*;
7
8 public class UserAccountTest {
9
10     @Before
11     public void setUp() throws Exception {
12     }
13
14     @Test
15     public void test() {
16         UserAccount user = new UserAccount();
17         user.setFirstName("ahmed");
18         assertEquals("ahmed",user.getFirstName());
19         user.setLastName("Mohamed");
20         user.setUserName("ahmed113");
21         user.setPassword("123456");
22         assertEquals("Mohamed",user.getLastName());
23         assertEquals("ahmed113",user.getUserName());
24         assertEquals("123456",user.getPassword());
25     }
26
27     @Test
28     public void test2() {
29         UserAccount user = new UserAccount();
30         user.setFirstName("ahmed");
31         user.setLastName("Mohamed");
32         user.setUserName("ahmed113");
33         user.setPassword("123456");
34         user.deposit(1000);
35         assertEquals(1000,user.checkBalance(),0);
36     }
37 }
```



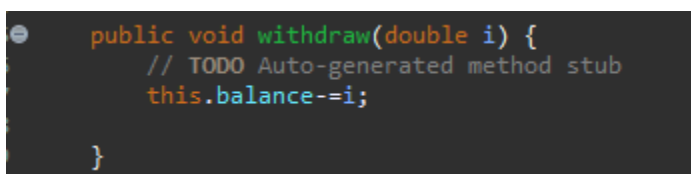
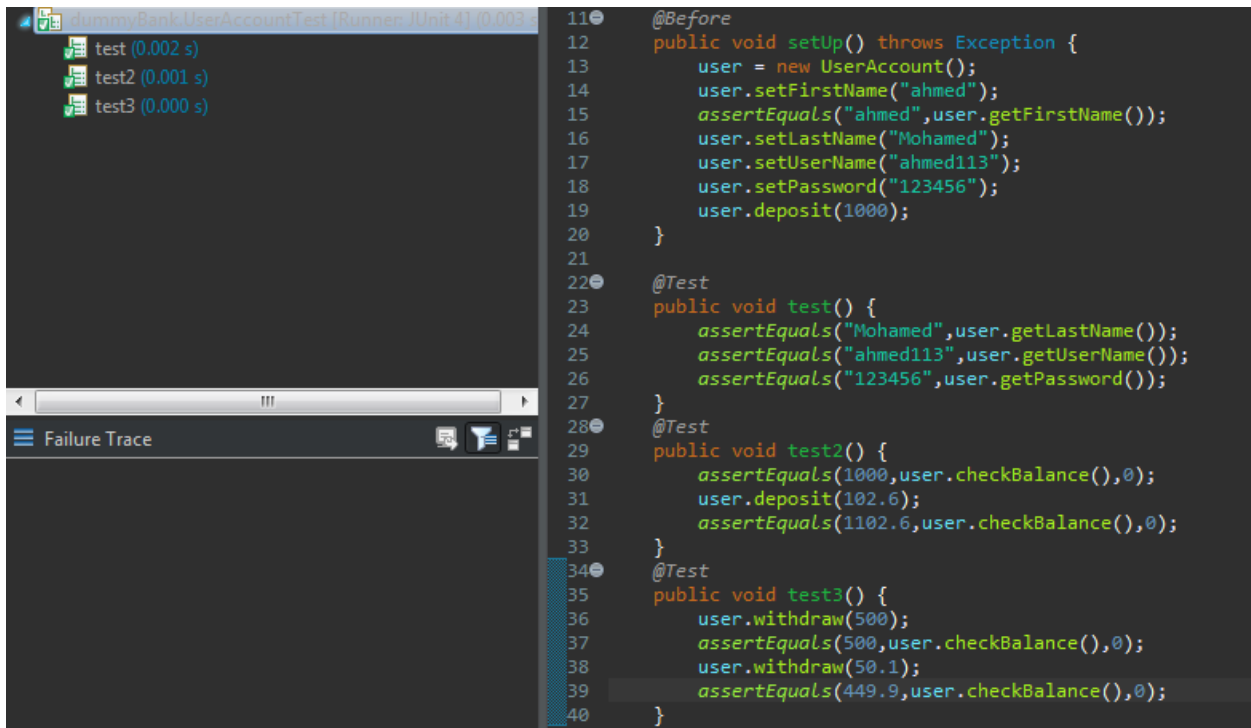
```
30 import static org.junit.Assert.*;
7
8 public class UserAccountTest {
9     UserAccount user;
10
11     @Before
12     public void setUp() throws Exception {
13         user = new UserAccount();
14         user.setFirstName("ahmed");
15         assertEquals("ahmed",user.getFirstName());
16         user.setLastName("Mohamed");
17         user.setUserName("ahmed113");
18         user.setPassword("123456");
19         user.deposit(1000);
20     }
21
22     @Test
23     public void test() {
24         assertEquals("Mohamed",user.getLastName());
25         assertEquals("ahmed113",user.getUserName());
26         assertEquals("123456",user.getPassword());
27     }
28
29     @Test
30     public void test2() {
31         assertEquals(1000,user.checkBalance(),0);
32         user.deposit(102.6);
33         assertEquals(1102.6,user.checkBalance(),0);
34     }
35 }
```

## Test 3: Withdraw

### Step 1: Red withdraw test

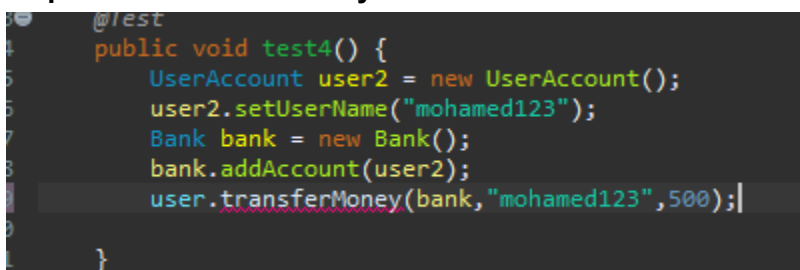


## Step 2: Green withdraw test + wrote withdraw function



## Test 4: transferMoney

### Step 1: Red transferMoney function



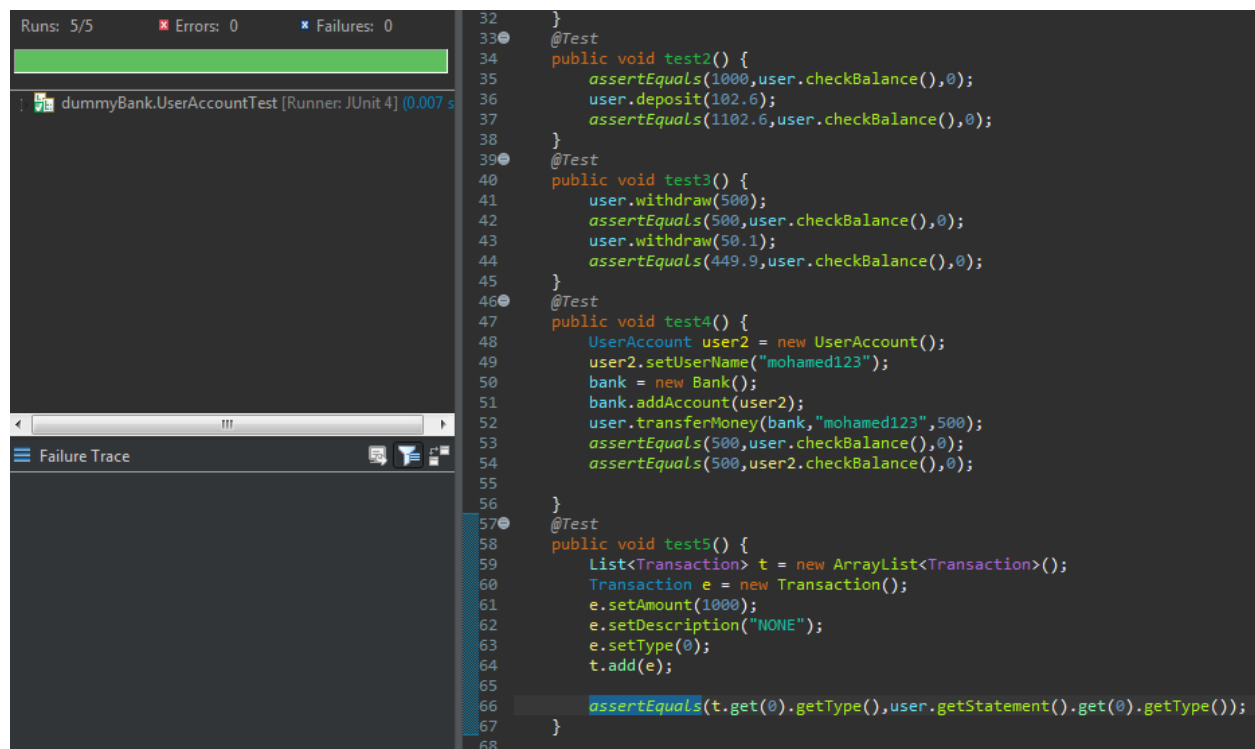
## Step 2: Writing transferMoney function

```
public void transferMoney(Bank bank, String string, double i) {  
    // TODO Auto-generated method stub  
    this.balance -= i;  
    bank.getAccount(string).deposit(i);  
}
```

The screenshot displays an IDE with two main panels. The left panel shows the test results for 'dummyBank.UserAccountTest'. It indicates 4 runs, 0 errors, and 0 failures. The tests listed are 'test' (0.001 s), 'test2' (0.000 s), 'test3' (0.000 s), and 'test4' (0.001 s). Below the test results is a 'Failure Trace' section, which is currently empty. The right panel shows the source code for 'UserAccountTest.java'. The code includes several test methods: 'test' (initializes a user and deposits 1000), 'test2' (deposits 1000 and then 102.6), 'test3' (withdraws 500 and then 50.1), and 'test4' (creates a new user, adds them to the bank, and transfers 500 from the first user to the second). The 'transferMoney' method is called in 'test4'.

```
16     user.setFirstName("ahmed");  
17     assertEquals("ahmed", user.getFirstName());  
18     user.setLastName("Mohamed");  
19     user.setUserName("ahmed113");  
20     user.setPassword("123456");  
21     user.deposit(1000);  
22 }  
23  
24 @Test  
25 public void test() {  
26     assertEquals("Mohamed", user.getLastName());  
27     assertEquals("ahmed113", user.getUserName());  
28     assertEquals("123456", user.getPassword());  
29 }  
30 @Test  
31 public void test2() {  
32     assertEquals(1000, user.checkBalance(), 0);  
33     user.deposit(102.6);  
34     assertEquals(1102.6, user.checkBalance(), 0);  
35 }  
36 @Test  
37 public void test3() {  
38     user.withdraw(500);  
39     assertEquals(500, user.checkBalance(), 0);  
40     user.withdraw(50.1);  
41     assertEquals(449.9, user.checkBalance(), 0);  
42 }  
43 @Test  
44 public void test4() {  
45     UserAccount user2 = new UserAccount();  
46     user2.setUserName("mohamed123");  
47     Bank bank = new Bank();  
48     bank.addAccount(user2);  
49     user.transferMoney(bank, "mohamed123", 500);  
50     assertEquals(500, user.checkBalance(), 0);  
51     assertEquals(500, user2.checkBalance(), 0);  
52 }  
53 }
```

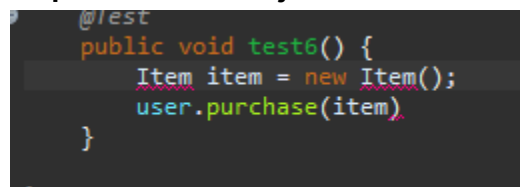
## Test 5: Integration Test with Transaction class



```
32 }
33 @Test
34 public void test2() {
35     assertEquals(1000, user.checkBalance(), 0);
36     user.deposit(102.6);
37     assertEquals(1102.6, user.checkBalance(), 0);
38 }
39 @Test
40 public void test3() {
41     user.withdraw(500);
42     assertEquals(500, user.checkBalance(), 0);
43     user.withdraw(50.1);
44     assertEquals(449.9, user.checkBalance(), 0);
45 }
46 @Test
47 public void test4() {
48     UserAccount user2 = new UserAccount();
49     user2.setUserName("mohamed123");
50     Bank bank = new Bank();
51     bank.addAccount(user2);
52     user.transferMoney(bank, "mohamed123", 500);
53     assertEquals(500, user.checkBalance(), 0);
54     assertEquals(500, user2.checkBalance(), 0);
55 }
56 }
57 @Test
58 public void test5() {
59     List<Transaction> t = new ArrayList<Transaction>();
60     Transaction e = new Transaction();
61     e.setAmount(1000);
62     e.setDescription("NONE");
63     e.setType(0);
64     t.add(e);
65     assertEquals(t.get(0).getType(), user.getStatement().get(0).getType());
66 }
67 }
68 }
```

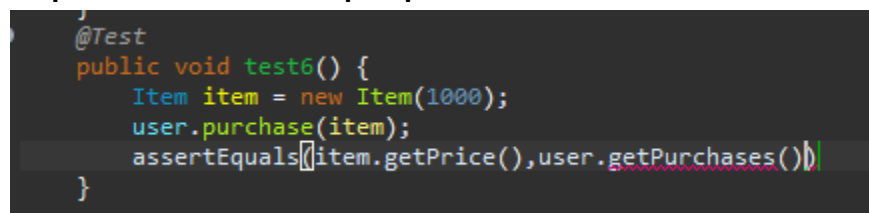
## Test 6: Integration Test with Item Class and purchases function

### Step 1: Red Item object instantiation test



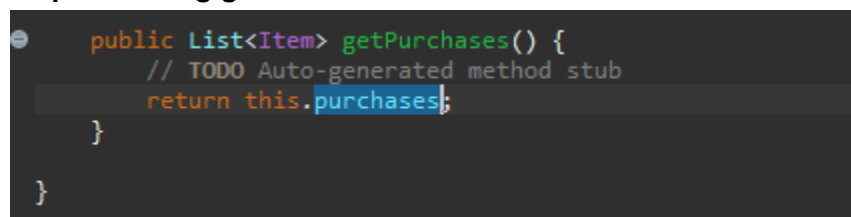
```
@Test
public void test6() {
    Item item = new Item();
    user.purchase(item);
}
```

### Step 2: Green Item + Step 1: purchases function



```
@Test
public void test6() {
    Item item = new Item(1000);
    user.purchase(item);
    assertEquals(item.getPrice(), user.getPurchases().get(0).getPrice());
}
```

### Step 2: Writing getPurchases

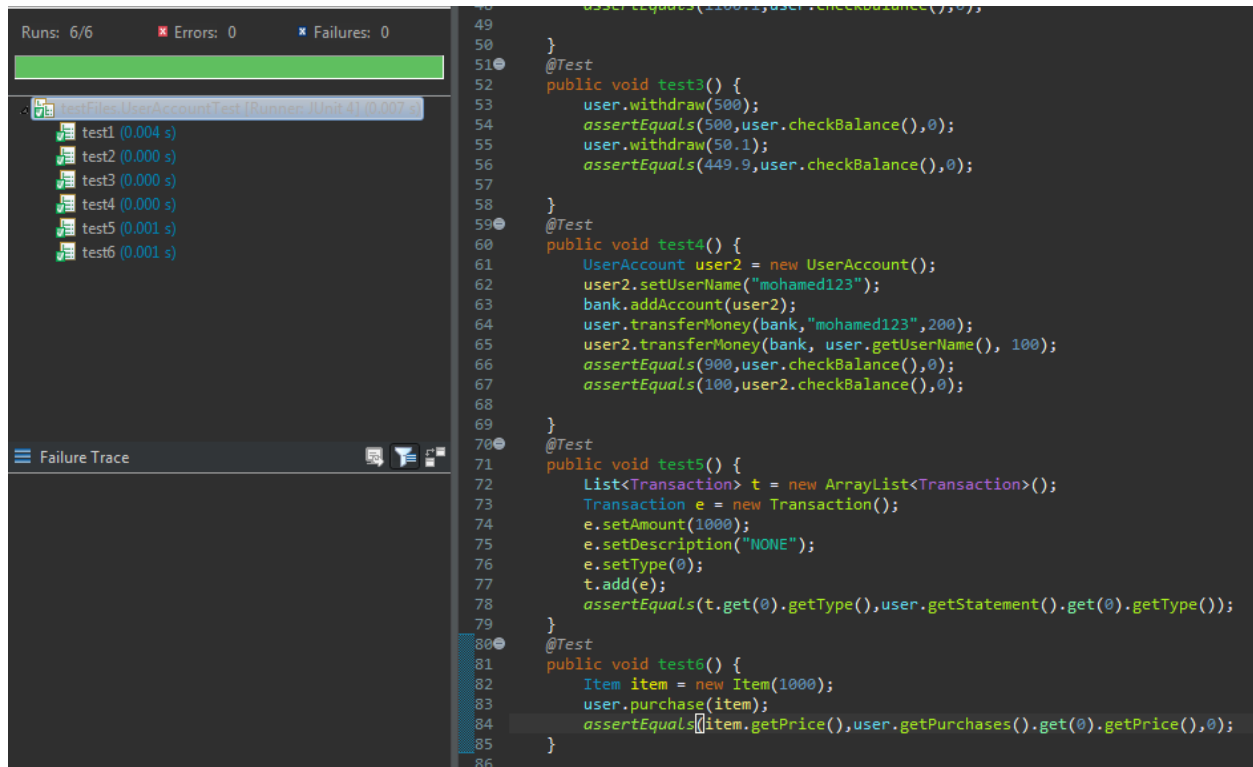


```
public List<Item> getPurchases() {
    // TODO Auto-generated method stub
    return this.purchases;
}

}
```

All tests now pass...





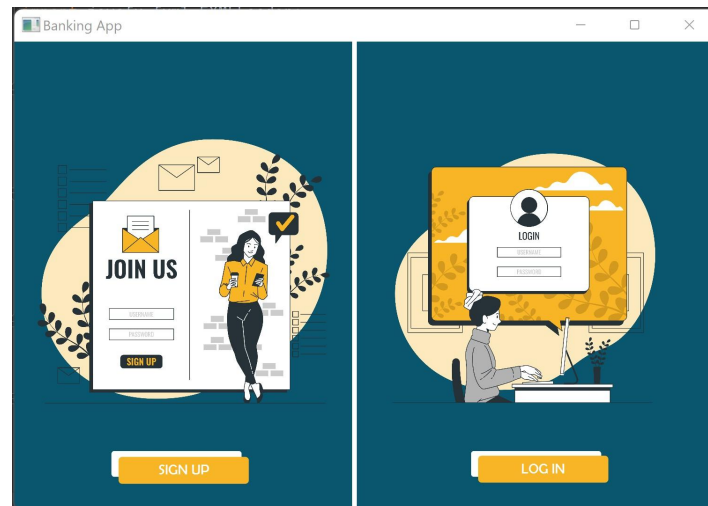
## GUI Testing

This kind of testing can be done manually or automated using any testing framework such as TestFX. Our TestFX code is now redundant since it was decided that manual tests were easier to run without many dependencies or libraries and easier for anyone to replicate. Regardless, here are some manually tested cases.

- Ensure buttons work as intended.
- Ensure error messages pop up when incorrect action taken.
- Ensure buttons that cause window to change work.

These test cases ensure our GUI works well in general.

First GUI Window



SignUp Window

The image shows a window titled "Sign Up" with a dark blue background. It features a "Back" button in the top left corner. The form is divided into two columns. The left column has a circular profile picture placeholder, followed by fields for "FIRST NAME", "LAST NAME", and "USERNAME". The right column has fields for "EMAIL", "PASSWORD", and "DATE OF BIRTH" with a calendar icon. A "Sign Up" button is located at the bottom center of the form.

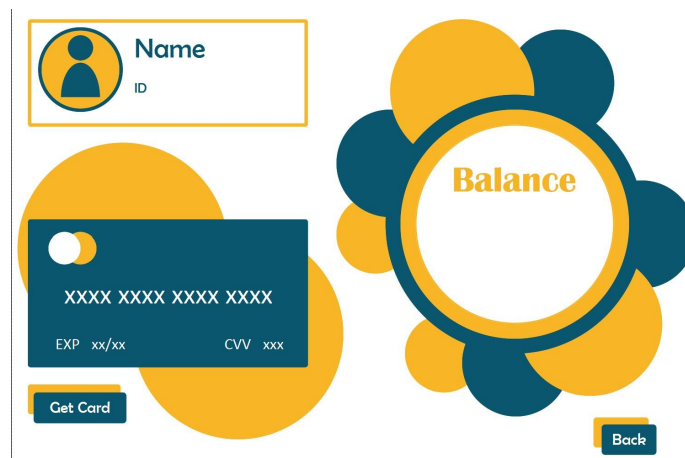
Login Window

The image shows a window titled "Log In" with a dark blue background. It features a "Back" button in the top left corner. The form is displayed on a smartphone screen, which is held by a large yellow hand. The form has a circular profile picture placeholder, followed by fields for "USERNAME" and "PASSWORD", and a "Log In" button. A yellow plant is visible in the bottom right corner of the window.

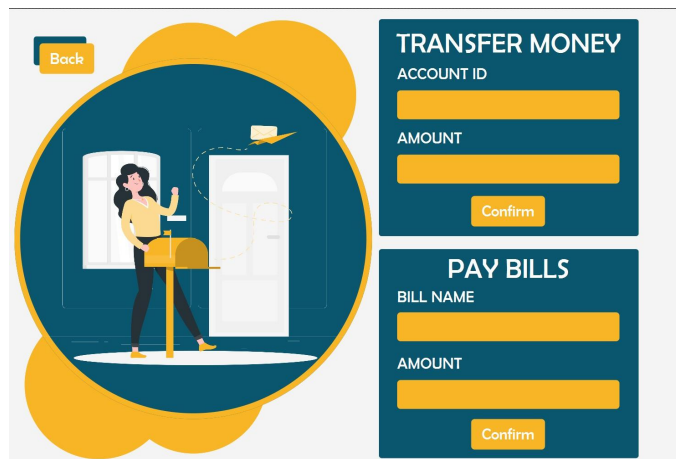
## Account Dashboard After Login



View Balance (This is the scenebuilder view. When the app runs, “Name” and “ID” will be replaced with the user’s name and ID respectively. His/Her balance amount will appear under Balance inside the circle on the right and his/her credit card information will appear in the card on the left.)



## Transfer Money | Pay Bills



Deposit | Withdraw

The image shows a UI mockup for a banking application. On the left, there is a circular illustration of a person at an ATM. To the right, there are two main sections: 'WITHDRAW MONEY' and 'DEPOSIT MONEY'. Each section has a title, an 'AMOUNT' label, a text input field, and a 'Confirm' button. A 'Back' button is located at the top left of the mockup.

View Bank Statement (will be filled by the logged in user's transactions)

The image shows a UI mockup for a 'Bank Statement' screen. It features a 'Back' button at the top left. The title 'Bank Statement' is centered at the top. Below the title is a table with five columns: 'Date', 'Description', 'Withdrawals', 'Deposits', and 'Balance'. The table is currently empty, and the text 'No content in table' is displayed in the center.

Date	Description	Withdrawals	Deposits	Balance
No content in table				

## System Testing

System testing tests the whole software end-to-end. Thus, we now need to run the project and check for some real-life scenarios.

- After signing up for an account, we can log in and out of the account more than once.  
(home screen->go to online bank->sign up-> enter details and click sign up-> enter details and log in-> back-> login-> enter details and login-> back-> login -> enter details and login)
- After logging in, we can deposit 20000\$, withdraw 100\$, transfer 100\$, and check bank account for the remaining balance equal 19800\$.

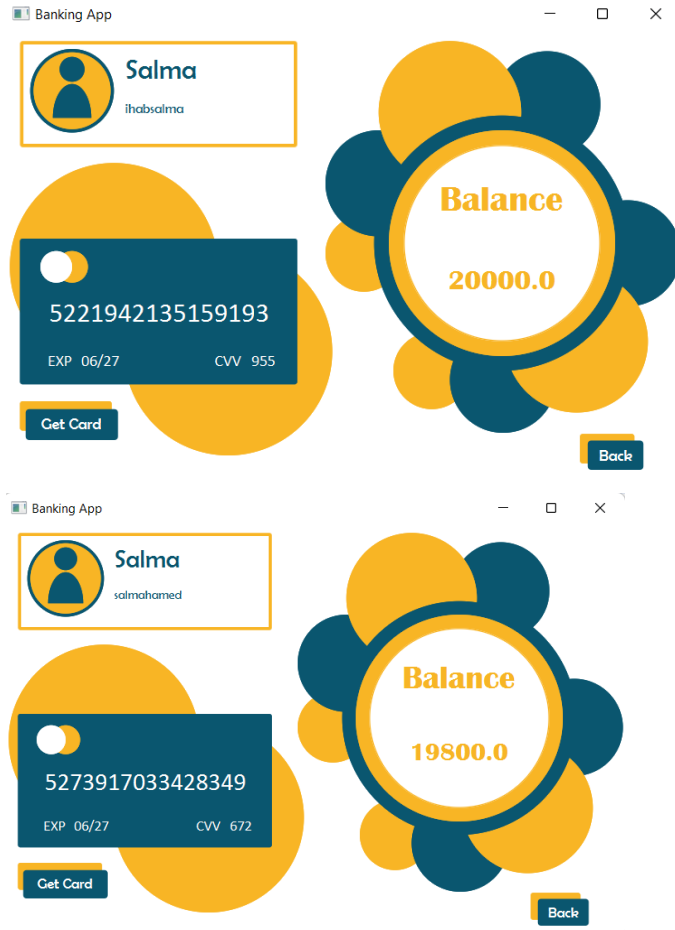
Banking App

Back

Bank Statement

Date	Description	Type	Amount	Balance	
Sat Jun 11 22:31:...	NONE	DEPOSIT	20000.0	0.0	
Sat Jun 11 22:31:...	NONE	WITHDRAWAL	100.0	20000.0	
Sat Jun 11 22:31:...	NONE	TRANSFER	100.0	19800.0	

- After checking balance, we can log out and back in with no problem.
- After signing out, we can sign up for a new account, and log out and into our old account.



## Performance Testing

Performance testing is the practice of testing the quality of capability of a software. It involves non-functional testing of the system. This extends to throughput, response time, latency and scalability.

To simulate load testing, we will create many accounts in the one running instance, and see how the program behaves. We can also use the IntelliJ performance plugin to check start-up as follows.

Startup Time Cost per Plugin		✕
Plugin	Startup Time (ms)	
Kotlin	25025	
Android	18410	
Code With Me	14398	
Package Search	8712	
Database Navigator	7180	
Maven	6599	
IntelliLang	4034	
Space	3452	
Configuration Script	2592	
Groovy	2577	
Shared Indexes	1708	
WSL File System Support	1185	
Windows 10 Light Theme	1175	
Git	1118	
Grazie	1116	
EditorConfig	1034	
Markdown	953	
GitHub	867	
Machine Learning Code Com...	820	
Task Management	779	
Ant	765	
Gradle	764	
Plugin DevKit	703	
Lombok	629	
IDE Features Trainer	562	
JUnit	543	
Mercurial	537	
XPathView + XSLT	517	
Properties	473	
JavaFX	467	
Gradle-Java	426	
Subversion	378	
Disable Selected Plugins		OK Cancel

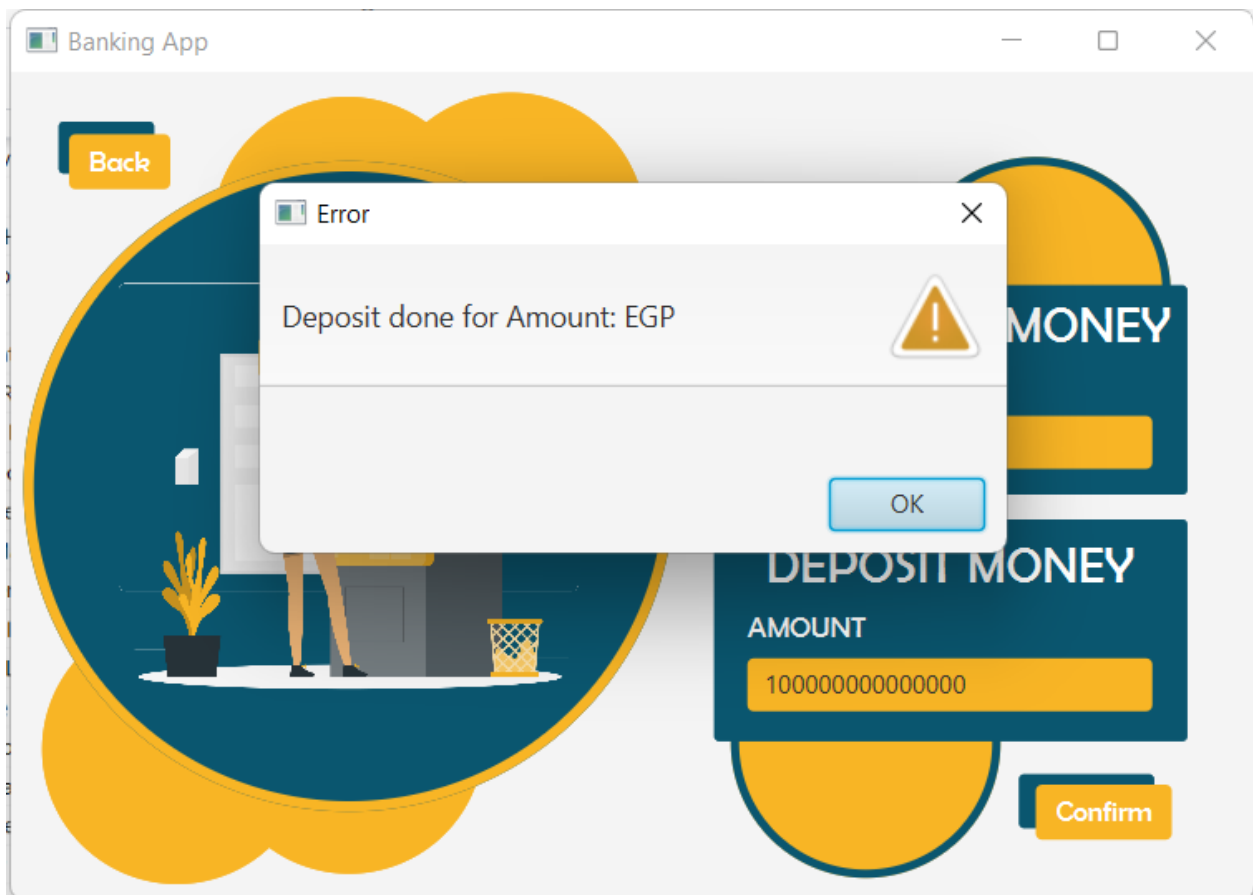
Startup Time Cost per Plugin	
Plugin	Startup Time (ms)
Qodana	375
TextMate Bundles	331
Java Internationalization	303
Images	299
JetBrains Repository Search	296
TestNG	281
Shell Script	244
UI Designer	221
WebP Support	192
Copyright	191
Code Coverage for Java	164
YAML	141
Perforce Helix Core	130
XSLT Debugger	127
Terminal	111
Eclipse Interoperability	101
Settings Repository	73
JetBrains Marketplace Licensi...	54
Java Bytecode Decompiler	49
Gradle Dependency Updater L...	45
Gradle-Maven	32
Dependency Management Ap...	21
com.intellij.tracing.ide	15
Machine Learning in Search E...	5
Bytecode Viewer	5
Smali Support	2
Java Stream Debugger	2
ChangeReminder	1
Projector Libraries for Code ...	1
Gradle DSL API	0
JetBrains maven model api cl...	0
IDE Features Trainer: Git Less...	0

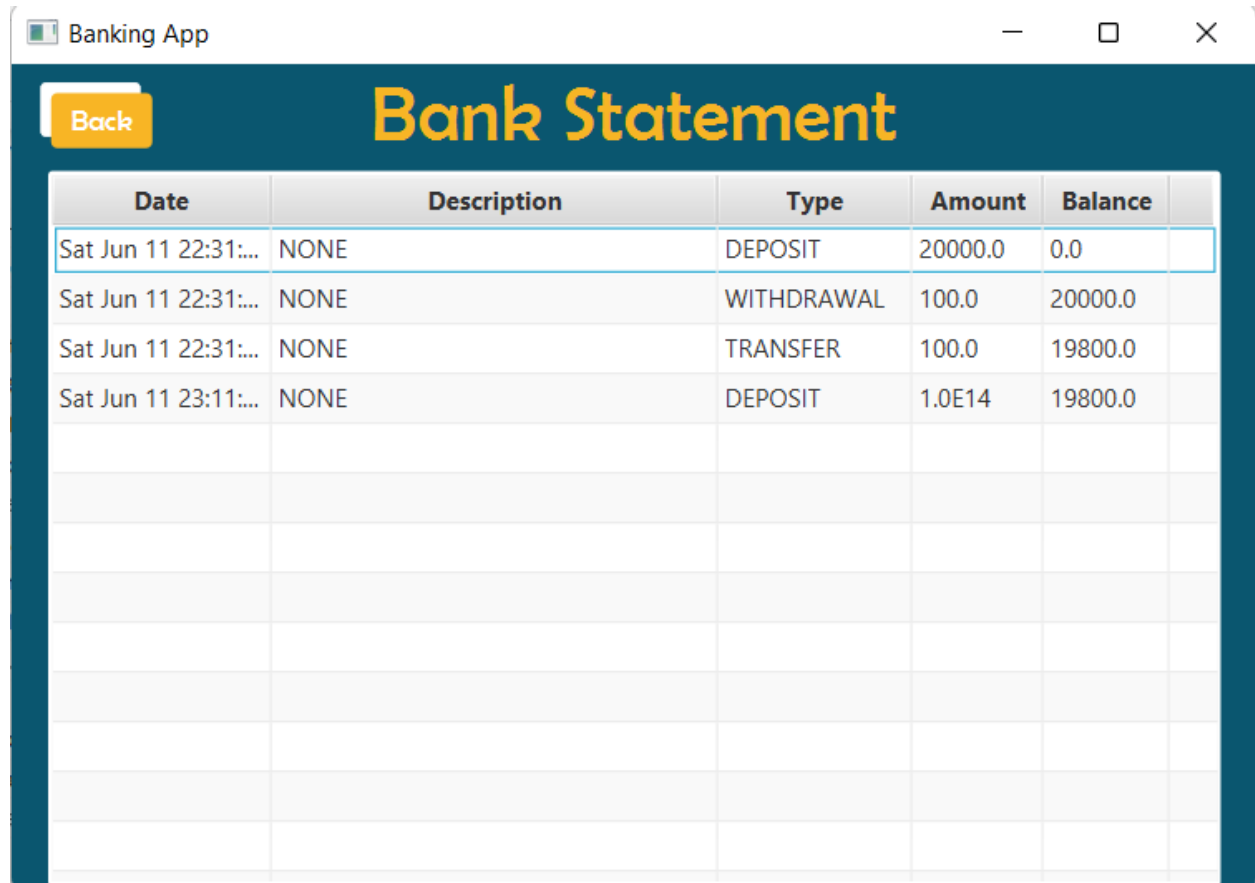
## Stress Testing

Insert a large number to deposit, the app will not crash and the bank statement works as usual.



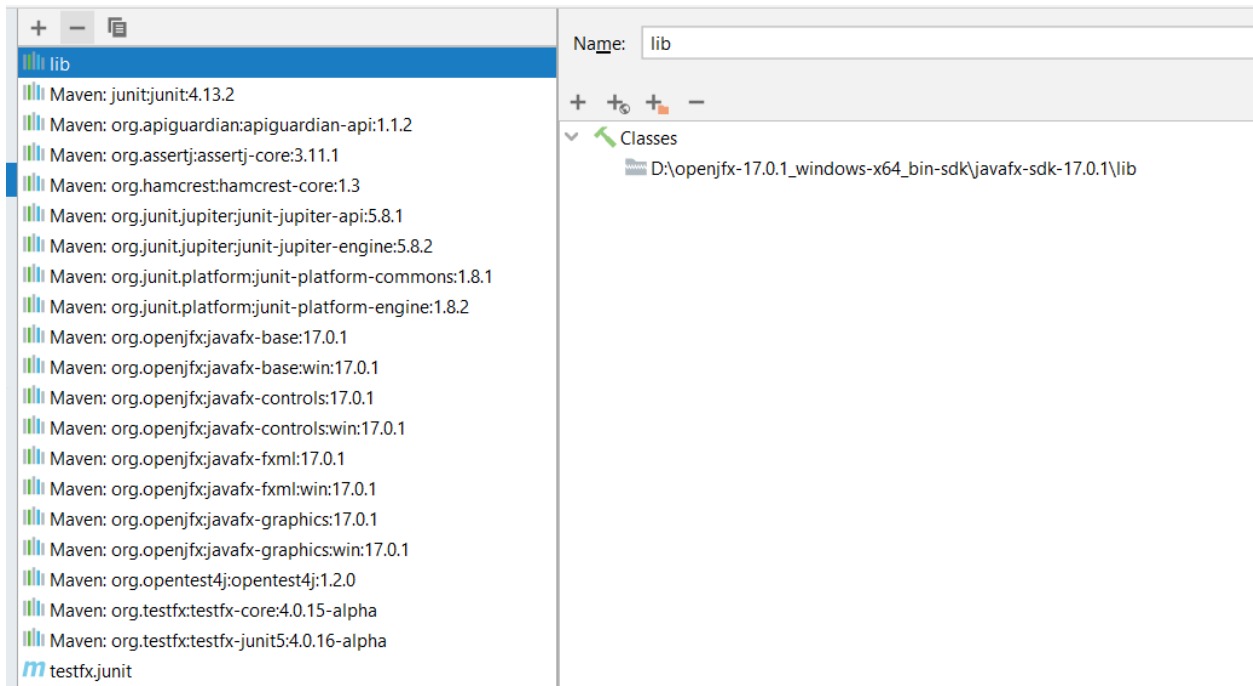
Confirm



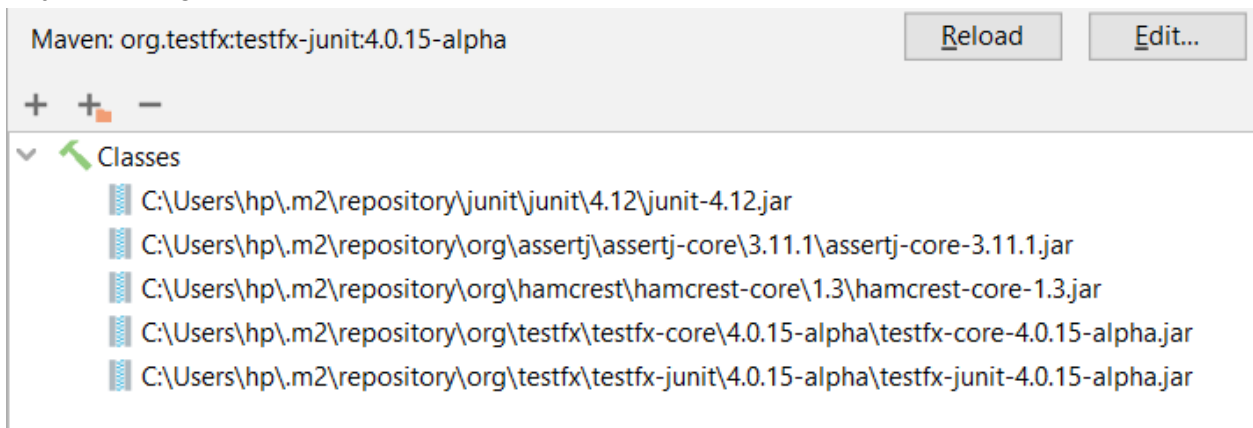


## Tools Used in Development

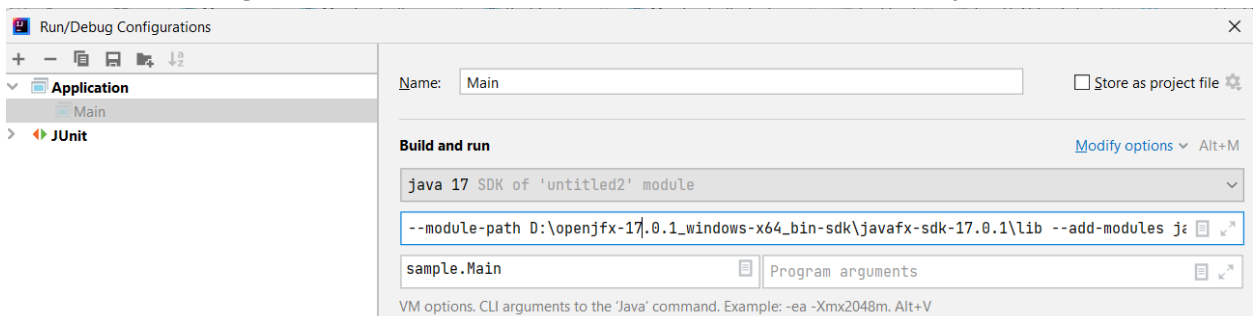
- IntelliJ IDEA as IDE
- Java SDK 17
- JavaFX and SceneBuilder for GUI
- Maven to create project with dependencies
- Junit for unit tests
- TestFX for GUI testing
- Jmeter for performance testing
- **Libraries:**



'lib' is javafx lib, and 'testfx' is the following: (More details about dependencies in pom.xml file in project zip or github)



- **Run Configuration:** this is important to setup to run a JavaFX project in IntelliJ



## Closing Notes on Reuse and Refactoring

The Item class unit tests, while useful, did not make it to the final version of the app since the purchases with credit card option were removed as it was not able to integrate with GUI.

Therefore, Item class was unnecessary in the end, but perhaps in a future update, the user will be able to make purchases with his credit card. The fxml files for the purchases page also exist in the project folder and can easily be reintegrated once regression testing has proven they will not break the app again.