软件测试上机报告

第一次上机作业

学　　院___智能与计算学部__
专　　业___软件工程_____
姓　　名___王兆盟_____
学　　号___3017218072_____
年　　级____2017 级_____
班　　级___软件工程一班____

1、Experiment Requirement

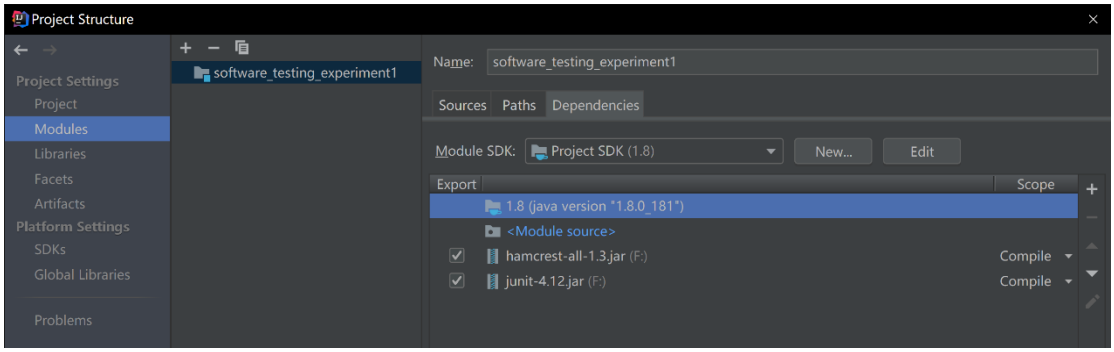
1. Install Junit(4.12), Hamcrest(1.3) with Eclipse/IDEA
2. Install Eclemma with Eclipse or other tools for studying the testing code coverage.
3. Write a java program for the given problem and test the program with Junit.

   The Description of the problem:

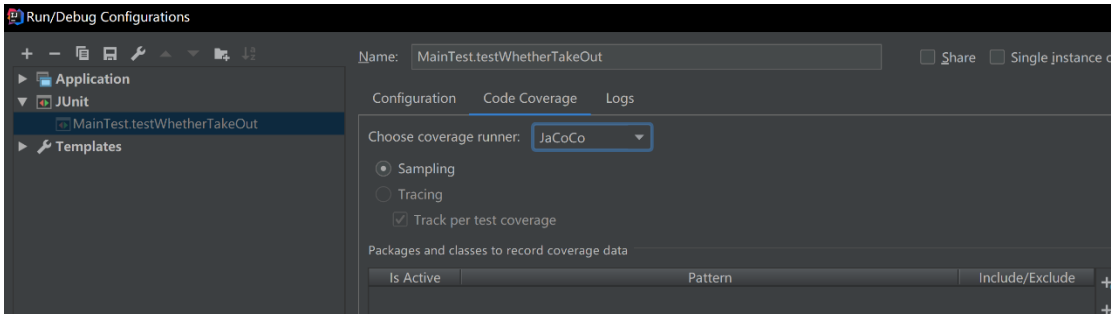
   There is one 50 yuan, one 20 yuan, one 10 yuan, two 5 yuan bills and three 1 yuan coins in your pocket. Write a program to find out whether you can take out a given number (x) yuan.
4. Check in your java code and junit test program to github
5. Briefly descript the test result and coverage report (print screen) of the tests on this problem.

2、Source Code

Firstly, I import the package Junnit.jar and hamcreast.jar into my project in order to do the unit test.



And because the IntelliJ idea seemingly doesn't support the plugin Eclemma, so I choose to use another coverage runner called JaCoCo.



The source codes and annotation of the algorithm to solve this problem are shown below:

```java
public static Map<Double, Integer> getTakeOutSolution(NavigableMap<Double,
Integer> denominationsSortedByKeyDesc, double target) {

    Map<Double, Integer> result = new HashMap<Double, Integer>();
    // 获取最大值
    double max = denominationsSortedByKeyDesc.firstKey();
    int inventory = denominationsSortedByKeyDesc.firstEntry().getValue();
    int times = (int) (target / max);
    // 可以直接由最大金额组成，无需往下判断
    if (target % max == 0 && times <= inventory) {
        result.put(max, times);
        return result;
    }
    // 只有一个面额的情况
    if (denominationsSortedByKeyDesc.size() == 1) {
        throw new IllegalArgumentException("can not take out this given
number.");
    }
    double min = denominationsSortedByKeyDesc.lastKey();
    // 目标金额比最小面额还小，无法组合
    if (target < min) {
        throw new IllegalArgumentException("can not take out this given
number.");
    }
    if (times > inventory) {
        times = inventory;
    }
    for (int i = times; i >= 0; i--) {
        if (i > 0) {
            result.put(max, i);
        }
        try {

result.putAll(getTakeOutSolution(denominationsSortedByKeyDesc.subMap(max,
false, min, true), target - i * max));
            return result;
        } catch (Exception e) {
            result.clear();
            continue;
        }
    }
    throw new IllegalArgumentException("can not take out this given
number.");
}

public static boolean whetherTakeOut(double given_number){
    NavigableMap<Double, Integer> denominations = new TreeMap<Double,
Integer>(new Comparator<Double>() {
```

```
        @Override
        public int compare(Double o1, Double o2) {
            if (o2.doubleValue() > o1.doubleValue()) {
                return 1;
            } else if (o2.doubleValue() == o1.doubleValue()) {
                return 0;
            } else {
                return -1;
            }
        }
    });
    denominations.put(50d, 1);
    denominations.put(20d, 1);
    denominations.put(10d, 1);
    denominations.put(5d, 2);
    denominations.put(1d, 3);
    if (given_number == 0)
        return true;
    try {
        getTakeOutSolution(denominations,given_number);
    }
    catch (IllegalArgumentException e){
        return false;
    }
    return true;
}
```

And I design a Java class as a test class for testing the class shown above which developed to solve the problem. In this I use Junit as a tool to do the unit test. I annotate test class with @RunWith(Parameterized.class) and Create a public static method annotated with @Parameters that returns a collection of objects (as Array) as test data set. In total I have designed 11 test cases to test which is shown blow (in every case, the first element is the given number of money and second is the expected result):

```
{0.1,false},
{100,false},
{54,false},
{4,false},
{9,false},
{0,true},
{20,true},
{30,true},
{40,true},
{90,true},
{93,true},
```

In the end, I use the method known as “assertEquals()” which tests if the program can return the proper result as expected in each test case.

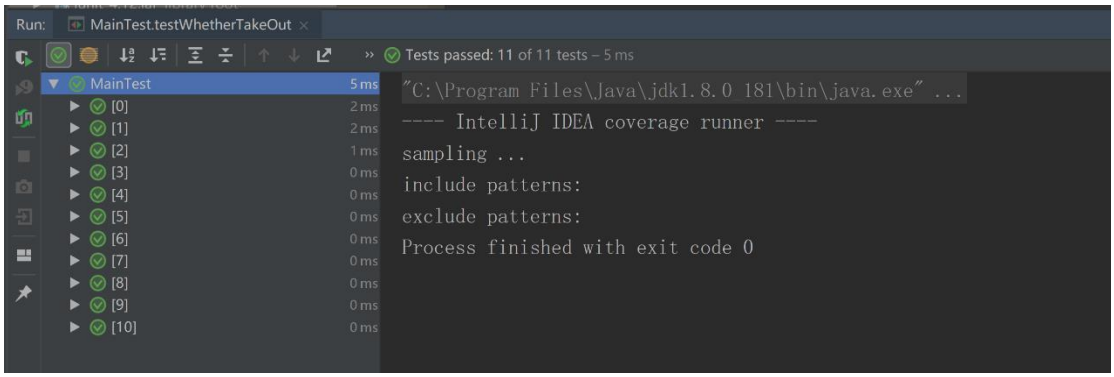
The source code of the test class is shown below:

```
@RunWith(Parameterized.class)
public class MainTest {
    private double given_number;
    private boolean expected;
    private MainProblem test = null;

    public MainTest(double given_number,boolean expected){
            this.given_number = given_number;
            this.expected = expected;
    }
    @Before
    public void setUp()
    {
        test = new MainProblem();
    }
    @Parameterized.Parameters
    public static Collection<Object[]> getTestCase(){
        return Arrays.asList(new Object[][]{
                {0.1,false},
                {100,false},
                {54,false},
                {4,false},
                {9,false},
                {0,true},
                {20,true},
                {30,true},
                {40,true},
                {90,true},
                {93,true},
        });
    }
    @Test
    public void testWhetherTakeOut(){

assertEquals(this.expected,test.whetherTakeOut(given_number));
    }
}
```

3、Experiment Result

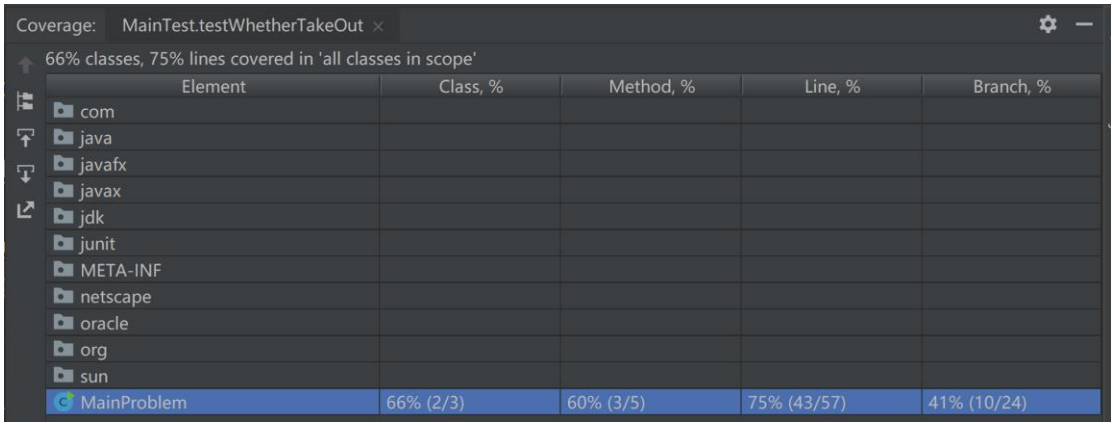
I run the 11 test cases and the result came out which demonstrate the program work properly and as the result produced as expected. Just like the saying goes: make the bar green and make your code clean.

The screen shot of the result is shown below:



Later I run it again in order to study the code coverage of this test process. I click the button “run with coverage” and generate the report which shows the visualization of the coverage.

And the statistical result are shown below:



Then I opened the report in the browser and found the segment that have not execute which marked in a red background. And the that part is the “main” function of that class which is designed for print some info during my developing period. So it won’t be executed in the test process.

```
72.     public static void main(String[] args) {
73.         NavigableMap<Double, Integer> denominations = new TreeMap<Double, Integer>(new Comparator<Double>() {
74.             @Override
75.             public int compare(Double o1, Double o2) {
76.                 if (o2.doubleValue() > o1.doubleValue())
77.                     return 1;
78.                 } else if (o2.doubleValue() == o1.doubleValue())
79.                     return 0;
80.                 } else {
81.                     return -1;
82.                 }
83.             }
84.         });
85.         denominations.put(50d, 1);
86.         denominations.put(20d, 1);
87.         denominations.put(10d, 1);
88.         denominations.put(5d, 2);
89.         denominations.put(1d, 2);
90.
91.         System.out.println(getTakeOutSolution(denominations, 80));
92.         //System.out.println(whether7akeOut(80));
93.     }
94. }
```