****

**软件测试技术第一次实验报告**



**学 院 智能与计算学部**

**专 业 软件工程**

**年 级 2016级**

**姓 名 曾晓东**

**2019年 3 月 12 日**

# 软件测试技术第一次实验报告

Github URL：

<https://github.com/nkaccounting/softwaretest18192/tree/master/lab1/src/moneyfun>

1. 需求分析（描述具体需求）

有一张50元，一张20元，2张5元和3枚1元硬币，给定一个数值x，问能不能够凑出刚好这个数值这么多的钱。

需要用Junit进行测试，并测算覆盖率。

1. 概要设计（简单描述设计思路，配合UML图）

1.junit, hamcrest，eclemma的安装过程：

在所在项目文件夹中右键build path，进入后选择Add External JARS，导入老师在群里上传的两个文件，点击ok就完成junit, hamcrest的安装。

Eclemma我是在上方help功能栏里面，找到install from catalog功能选项，进入到eclipse Marketplace里面，在find里输出eclemma，自动安装并重启即可，之后只要点击Coverage As --Junit test即可运行并查看覆盖率。

2.基本设计思路：

初始化所差金额为x，每次尽可能多的拿出一个最大的，同时不超过所差金额的钱币，直至x刚好等于0，说明x是可行的；若无法达到0，则说明x不可行。

1. 详细设计（详细描述具体如何实现，附代码及说明）

钱币数量可以用一个[1,1,2,3]的数组指示，每次要拿这个数量的货币时需要所在位>0,每次拿出后所在位-1；由于新的所差金额实际上是一个子问题，于是将上述基本设计思路设计成递归实现。代码如下：

代码：功能函数

**package** moneyfun;

**public** **class** moneycount {

// int []havemoney={50,20,5,5,1,1,1};

**int** []havemoney={1,1,2,3};

**int** flag=1;

**public** **int** minus(**int** m){

**if**(m>=50){

**if**(havemoney[0]==1){

m=m-50;

havemoney[0]=0;

minus(m);

}

**else**{

flag=0;

}

}

**else** **if**(m>=20){

**if**(havemoney[1]==1){

m=m-20;

havemoney[1]=0;

minus(m);

}

**else**{

flag=0;

}

}

**else** **if**(m>=5){

**if**(havemoney[2]>0){

m=m-5;

havemoney[2]=havemoney[2]-1;

minus(m);

}

**else**{

flag=0;

}

}

**else** **if**(m>=1){

**if**(havemoney[3]>0){

m=m-1;

havemoney[3]=havemoney[3]-1;

minus(m);

}

**else**{

flag=0;

}

}

**return** flag;

}

}

选取了20个测试用例，不同的参数，尽可能地对其进行覆盖。

设计@RunWith的参数化测试方法，定义input，expect，放入collection中，最后统一执行，让一个函数完成多个参数的测试，便于测试，也便于调试。

测试函数：

**package** moneyfun;

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

**import** java.util.Arrays;

**import** java.util.Collection;

**import** org.junit.Before;

**import** org.junit.Test;

**import** org.junit.runner.RunWith;

**import** org.junit.runners.Parameterized;

**import** org.junit.runners.Parameterized.Parameters;

@RunWith(Parameterized.**class**)

**public** **class** testj {

**public** **int** money;

**public** **int** result;

**public** moneycount mc=**null**;

**public** testj(**int** m,**int** r){

**this**.money=m;

**this**.result=r;

}

@Before

**public** **void** setup(){

mc=**new** moneycount();

}

@Parameters

**public** **static** Collection<Object[]>getData(){

**return** Arrays.*asList*(**new** Object[][]{

{100,0},{87,0},{84,0},{83,1},{36,0},{35,0},{33,1},{50,1},{20,1},{21,1},{5,1},{90,0},{15,0},{4,0},{10,1},{14,0},{13,1},{3,1},{2,1},{1,1}

});

}

@Test

**public** **void** testmoney(){

*assertEquals*(**this**.result,mc.minus(money));

}

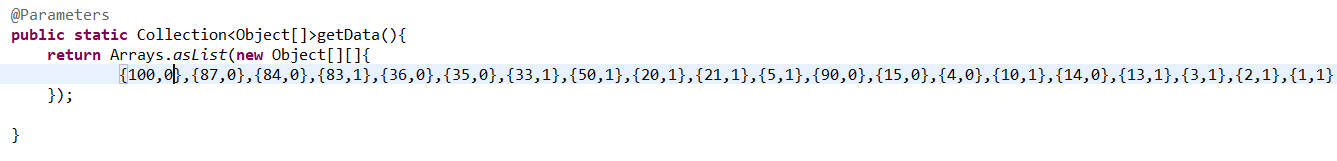
}

1. 调试分析（在实验过程中遇到的问题以及如何解决）

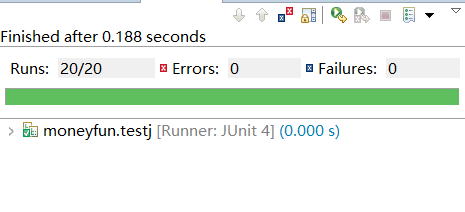
一开始我采用单测试输入数据测试，导致无论如何覆盖率都达不到100%，且测试过程很繁琐，需要一次一次输入，一次一次地查看结果，而后采用课上所讲的参数化的方法，最终实现了多个数据同时进行测试且覆盖率=100%的测试方式。

1. 测试结果（描述输入和输出）

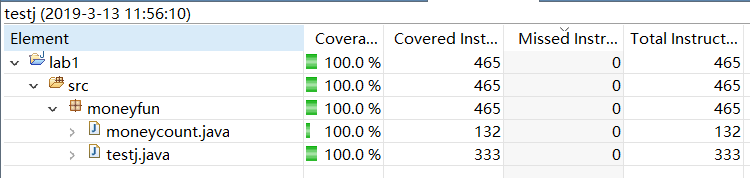
输入：



输出：



覆盖率情况：



1. 总结

通过本次实验我掌握了使用Junit进行测试的方法，Junit的使用非常方便，首先第一点不需要写main函数。

其次assertEquals的方法简单明了，能够直接比对数据，在左边的Junit一栏里面用颜色表示通过或者不通过测试，避免了手工写system.out需要一个一个去检查的工作。

同时在@Runwith的测试模式下，能够显示通过的比率，有多少通过，有多少未通过，一目了然。

再者Junit在配合eclemma的使用情况下还可以查看代码覆盖率，用百分比，颜色的方式很直观的表示出哪些代码有执行到，哪些代码没有执行到，是实际设计测试用例中保证全覆盖的一个基础，使用起来很好。

另外解决这个钱币问题还有别的方法，比如用if else暴力求解，或者将全部的可能的数据X算出存入数组中，再迭代一次比对x与X是否相等 等解法。

Github上URL：

https://github.com/nkaccounting/softwaretest18192/tree/master/lab1/src/moneyfun