自动售货机（VendingMachine）程序

测试报告

版本 <1.2>

**曹金坤、李晗东、程浩、吴正雨**

修订历史记录

|  |  |  |  |
| --- | --- | --- | --- |
| **日期** | **版本** | **说明** | **作者** |
| 2018/5/01 | 1.0 | 开关机功能测试样例的填写 | 李晗东 |
| 2018/5/02 | 1.1 | 购买商品功能测试样例的填写 | 李晗东 |
| 2018/5/03 | 1.2 | 退币功能测试样例的填写 | 李晗东 |
| 2018/5/03 | 1.2.1 | 添加功能代码说明 | 曹金坤 |
|  |  |  |  |

目录

1. 简介 4

1.1 目的 4

1.2 范围 4

1.3 参考资料 4

1.4 概述 4

2. 开关机功能 4

3. 投币购买商品 5

4. 退币功能 7

5. 测试结果 7

6. 附录 7

测试报告

# 简介

本测试用例报告主要针对“自动售货机”（VendingMachine）项目第二次迭代完成以后，对该系统的开关机，投币售卖商品以及退币等功能的测试用例设计、测试过程、测试结果等项目进行报告说明。

本次用于测试的自动售货机程序主要包括以下功能：

* + 接受用户投入的10元，20元，50元与100元面值的硬币进行交易，也可以通过找零返回上述面值的硬币；
  + 售货机贩售三种商品，分别是售价为60的A，100的B和170的C，用户可以一次选择购买一种商品，多次购买可以连续操作；
  + 开关机功能，对于非法操作的警告报错等非功能性的功能；

针对上述的自动售货机功能，基于“白盒测试”的理论，我们设计了一系列的测试用例，对于自动售货机的功能逻辑和其他部分进行测试，测试代码基于Junit测试工具进行了实现，相关功能代码和代码可见其后部分的说明。对于测试的具体项目和测试的结果报告也提供了说明。该报告完整呈现了整个测试过程的不同环节。

## 目的

基于“白盒测试”的要求，针对客户端程序的功能和实现逻辑，对各个功能进行用例设计，进行测试，并撰写用例报告和之后的结果报告和分析。综上工作，测试软件功能的实现是否符合设计要求，发现软件开发中存在的各种问题，便于改正和进一步完善工作。

## 范围

本文档适用于“自动售货机”(VendingMachine)项目第二次迭代产生的用户程序开发的测试工作。

## 参考资料

《软件测试——一个软件工艺师的方法》、《测试计划》

## 概述

本文档主要基于“白盒测试”理论进行撰写，与前期的《测试计划》为同一工作的不同阶段的说明总结。在基于Junit测试工具的测试过程中，被测试的客户端程序功能主要包括对开关机，投币购买商品并找零，退币等。第二章提供了客户端程序的代码文件的简要介绍，相关的测试逻辑用例在第三章及后部分被提供，报告的最后还提供了测试结果的说明总结。在测试的过程中，有以下假设条件需要被说明：

* 自动售货机初始默认为关闭的状态，此时对于它的一系列操作都不应该生效，在售货机开启后才可以执行一系列操作；
* 基于自动售货机程序的设计初衷，合法操作中，单次仅仅可以购买一件商品；
* 当售货机中有足够的硬币余额但是面值组合不支持找零时，售货机程序会拒绝此次的购买请求，并且提供相关的信息说明。

本次测试工作全程在IDEA软件中进行，相关的工具栈在报告后面部分中被详细说明。

# 主程序代码说明

为了帮助说明测试代码的组织，特对主程序的功能代码也进行简单的说明：

* VendingMachine.java：入口文件，提供了对于底层功能函数的逻辑组织；
* Coin.java： 硬币类，提供了分别对应于10、50、20、100四种面值的硬币的属性和函数接口；
* Item.java： 物品类，提供了对应于售货机中的商品的属性和函数接口；
* Outputter.java： 输出类，是一个抽象类，定义了VendItem和ReturnCoin两个底层函数，分别对应于用户和售货机的两种基础的互动操作：购买物品和硬币退还；
* Storage.java & StorageImpl.java：存储类，其中Storage.java是预先定义的抽象类，包含了remove和add两个基础的抽象函数用来对售货机后台的货币剩余进行修改，StorageImpl.java提供了上述函数的具体实现；
* Bank.java：银行类，是一个和Storage类合作的类，用来计算找零和应付金额的数目。

# 测试用例说明

## 开关机功能

该部分测试用例用于测试自动售货机的开关机功能，关机状态下，所有对售货机的操作（开启售货机除外）都应该不生效。在关闭售货机时，其中的商品和钱应该被退出来。本测试对先对单个函数进行了测试，再对不同函数之间的可能组合进行了测试。具体的测试用例和设计的测试程序代码见下表：

表1. 开关机功能测试用例

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 功能 | 编号 | 测试函数名 | 前置条件 | 测试输入 | 预期结果 | 后置条件 |
| 开关机 | HH-A-RL-1 | shouldBeOffByDefault | 售货机保持关闭状态 | 无输入，检查售货机的状态 | 售货机保持关闭 | 无 |
| HH-A-RL-2 | shouldNotHaveSelectedItemByDefault | 售货机保持关闭状态 | 无输入，检查售货机已选择的项目 | 无选择项目 | 无 |
| HH-A-RL-3 | shouldTurnOn | 售货机保持关闭状态 | 执行开机操作，检查售货机的状态 | 售货机状态为开启 | 无 |
| HH-A-RL-4 | shouldTurnOff | 售货机保持开启状态 | 执行关机操作，检查售货机的状态 | 售货机状态为关闭 | 无 |
| HH-A-RL-5 | shouldReturnMoneyWhenTurnOff | 售货机保持开机状态，并有余额 | 执行关机操作，检查退出的钱 | 售货机退出所包含的 钱 | 无 |
| HH-A-RL-6 | shouldReturnItemsAndCoinsWhenTurnOff | 售货机保持开机状态，并有商品与余额 | 执行关机操作，检查退出的商品和钱 | 售货机退出所含有的商品和钱 | 无 |
| HH-A-RL-7 | shouldThrownExceptionIfCoinInsertedWhenMachineIsOff | 售货机保持关闭状态 | 投入硬币 | 售货机抛出错误，并退还硬币 | 无 |
| HH-A-RL-8 | shouldThrowExceptionIfPressReturnMoneyWhenMachineIsOff | 售货机保持关闭状态 | 按下退钱按钮 | 售货机抛出错误 | 无 |
| HH-A-RL-9 | shouldThrowExceptionIfSelectProductWhenMachineIsOff | 售货机保持关闭状态 | 按下商品选择按钮，选择A商品 | 售货机抛出错误 | 无 |

## 投币购买商品

在程序的操作逻辑中，用户可以选择投入10元，20元，50元与100元硬币，之后可以选择一款有库存的商品，和我们的日常经验相符合，只有当投入的硬币足够支付该商品并且售货机中有足够的硬币来进行找零，用户才可以成功的购买对应的商品。售货机会识别投入硬币的种类。针对这一售货机的主题功能，相关的测试内容如下表：

表2. 投币购买功能测试用例

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 功能 | 编号 | 测试函数名 | 前置条件 | 测试输入 | 预期结果 | 后置条件 |
| 投币购买商品 | HH-W-DA-1 | shouldThrowExceptionIfUnrecognizedCoinWasInserted | 售货机开启 | 投入10美元的硬币 | 售货机抛出错误：无法识别的硬币，并退出所投入的硬币 | 跳转到主界面 |
| HH-W-DA-2 | shouldThrowExceptionIfSelectedItemIsUnavailable | 售货机开启，B商品无库存 | 按下选择按钮选择B商品 | 售货机抛出错误：不可选择的商品 | 跳转到主界面 |
| HH-W-DA-3 | shouldVendSelectedItemWhenSelectItemAndInsertedCoinsAreExactlyPrice | 售货机开启，A商品有库存 | 投入60元硬币，选择购买A商品，检查出货情况与售货机余额 | 弹出消息：购买成功，给出商品A，售货机余额增加60元 | 跳转到主界面 |
| HH-W-DA-4 | shouldVendSelectedItemAndReturnChangeWhenSelectItemAndInsertedCoinsAreMoreThanPrice | 售货机开启，C商品有库存 | 投入2枚50元的硬币，1枚100元的硬币，选择购买C商品，检查出货情况，找零情况与售货机余额 | 弹出消息：购买成功，给出C商品，并退出30元，售货机余额增加170元 | 跳转到主界面 |
| HH-W-DA-5 | shouldVendSelectedItemAndReturnChangeWhenInsertCoinsMoreThanPrice | 售货机开启，A商品有库存 | 投入1枚50元，1枚100元，选择购买商品，检查出货情况与售货机余额 | 弹出消息：购买成功，给出A商品，并退出90元，售货机余额增加60元 | 跳转到主界面 |
| HH-W-DA-6 | shouldThrowExceptionIfInsufficientFundsInMachine | 售货机开启，A商品有库存，内部不含任何可找零的硬币 | 投入一枚100元，选择购买A商品，检查出货情况与退币情况 | 售货机抛出错误：售货机余额不足，并退出所投入的硬币 |  |
| HH-W-DA-7 | shouldSelectItemIfAvailable | 售货机开启，A商品有库存 | 按下选择按钮选择A商品，检查售货机的选择状态 | 售货机的状态中只选择了A | 无 |

## 退币功能

当使用者投入硬币后不想购买，直接按退币按钮可以退出投入的硬币。相关的测试用例见下表：

表3. 退出硬币功能测试用例

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 功能 | 编号 | 测试函数名 | 前置条件 | 测试输入 | 预期结果 | 后置条件 |
| 退币 | HH-W-UM-1 | shouldReturnInsertedCoinsWhenPressReturnMoney | 售货机开启 | 投入1枚50元，一枚10元的硬币，按下退币按钮 | 售货机退还投入的10元与50元硬币 | 无 |

# 测试结果

利用junit进行测试后的测试结果显示：所有测试均通过，其中类覆盖率达到100%，方法覆盖率达 到93%，代码覆盖率达到97%。基于IDEA的测试界面结果截图见下两图。

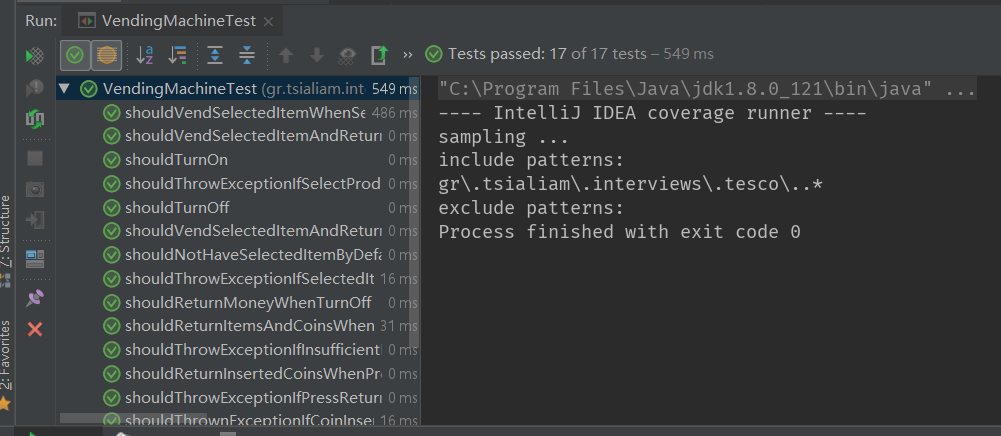


图1. Junit测试代码结果截图

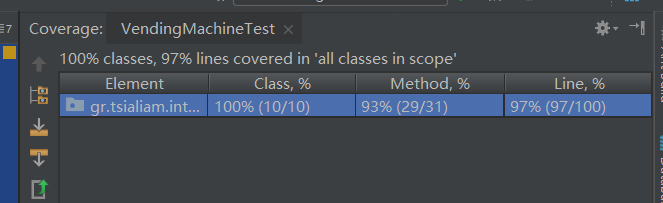


图2. 测试代码覆盖情况截图

# 附录

1. 测试代码

public class VendingMachineTest {

@Rule

public ExpectedException expectedException = ExpectedException.none();

//初始时售货机应当保持关闭的状态

@Test

public void shouldBeOffByDefault() {

VendingMachine testee = new VendingMachine(new Outputter(), createABank(), createAStorage());

assertThat(testee.isOn(), is(false));

}

//初始时售货机应当默认没有选择商品

@Test

public void shouldNotHaveSelectedItemByDefault() {

VendingMachine testee = new VendingMachine(new Outputter(), createABank(), createAStorage());

assertThat(testee.getSelectedItem(), is(nullValue()));

}

//按下开机键后售货机应当开机

@Test

public void shouldTurnOn() {

VendingMachine testee = new VendingMachine(new Outputter(), createABank(), createAStorage());

testee.setOn();

assertThat(testee.isOn(), is(true));

}

//按下关机键后售货机应当关机

@Test

public void shouldTurnOff() {

VendingMachine testee = new VendingMachine(new Outputter(), createABank(), createAStorage());

testee.setOn();

testee.setOff();

assertThat(testee.isOn(), is(false));

}

//关机时应当退出售货机中的钱

@Test

public void shouldReturnMoneyWhenTurnOff() {

Outputter outputter = mock(Outputter.class);

VendingMachine testee = new VendingMachine(outputter, createABank(), createAStorage());

testee.setOn();

testee.insertCoin(Coin.FIFTY\_PENCE.getId());

testee.insertCoin(Coin.ONE\_POUND.getId());

testee.setOff();

verify(outputter, times(1)).returnCoin(Arrays.asList(Coin.FIFTY\_PENCE, Coin.ONE\_POUND));

}

//关机时应当退出售货机中的商品与硬币

@Test

public void shouldReturnItemsAndCoinsWhenTurnOff() {

Outputter outputter = mock(Outputter.class);

Storage storage = mock(Storage.class);

VendingMachine testee = new VendingMachine(outputter, createABank(), storage);

testee.setOn();

Item anyAvailableItem = Item.C;

testee.selectItem(anyAvailableItem);

testee.insertCoin(Coin.TEN\_PENCE.getId());

storage.add(Item.C);

storage.remove(Item.C);

testee.setOff();

assertThat(testee.getSelectedItem(), is(nullValue()));

assertThat(testee.isOn(), is(false));

verify(outputter, times(1)).returnCoin(Collections.singletonList(Coin.TEN\_PENCE));

InOrder order = Mockito.inOrder(storage);

order.verify(storage).remove(anyAvailableItem);

order.verify(storage).add(anyAvailableItem);

}

//当售货机关机时，投入硬币应当报错

@Test

public void shouldThrownExceptionIfCoinInsertedWhenMachineIsOff() {

VendingMachine testee = new VendingMachine(new Outputter(), createABank(), createAStorage());

expectedException.expect(VendingMachineIllegalStateException.class);

testee.insertCoin(Coin.TEN\_PENCE.getId());

}

//购买商品时若投入无法识别的硬币应当报错

@Test

public void shouldThrowExceptionIfUnrecognizedCoinWasInserted() {

VendingMachine testee = new VendingMachine(new Outputter(), createABank(), createAStorage());

testee.setOn();

String unrecognizedCoin = "$10";

expectedException.expect(VendingMachineUnrecognizedCoinException.class);

expectedException.expectMessage("Unable to recognise coin " + unrecognizedCoin);

testee.insertCoin(unrecognizedCoin);

}

//当售货机关机时，按退钱按钮应当报错

@Test

public void shouldThrowExceptionIfPressReturnMoneyWhenMachineIsOff() {

VendingMachine testee = new VendingMachine(new Outputter(), createABank(), createAStorage());

expectedException.expect(VendingMachineIllegalStateException.class);

testee.returnMoney();

}

//购买商品时，投入硬币以后不想买了，按退钱按钮应当退出所投入的钱

@Test

public void shouldReturnInsertedCoinsWhenPressReturnMoney() {

Outputter outputter = mock(Outputter.class);

VendingMachine testee = new VendingMachine(outputter, createABank(), createAStorage());

testee.setOn();

testee.insertCoin(Coin.FIFTY\_PENCE.getId());

testee.insertCoin(Coin.TEN\_PENCE.getId());

testee.returnMoney();

verify(outputter, times(1)).returnCoin(Arrays.asList(Coin.FIFTY\_PENCE, Coin.TEN\_PENCE));

assertThat(testee.getSelectedItem(), is(nullValue()));

}

//售货机关机时，若选择商品应该报错

@Test

public void shouldThrowExceptionIfSelectProductWhenMachineIsOff() {

VendingMachine testee = new VendingMachine(new Outputter(), createABank(), createAStorage());

expectedException.expect(VendingMachineIllegalStateException.class);

testee.selectItem(Item.A);

}

//购买商品时，若售货机发现无法找零应该报错并退还所投入的硬币

@Test

public void shouldThrowExceptionIfInsufficientFundsInMachine(){

Map<Coin, Integer> money = new EnumMap<>(Coin.class);

money.put(Coin.TEN\_PENCE, 0);

money.put(Coin.TWENTY\_PENCE, 0);

money.put(Coin.FIFTY\_PENCE, 0);

money.put(Coin.ONE\_POUND, 0);

Outputter outputter = mock(Outputter.class);

VendingMachine testee = new VendingMachine(outputter, new FakeBank(money), createAStorage());

testee.setOn();

testee.insertCoin(Coin.ONE\_POUND.getId());

expectedException.expect(VendingMachineInsufficientFundsException.class);

testee.selectItem(Item.A);

}

//购买商品时，若所选的项目已经售空，应当报错

@Test

public void shouldThrowExceptionIfSelectedItemIsUnavailable(){

VendingMachine testee = new VendingMachine(new Outputter(), createABank(), createAStorage());

testee.setOn();

Item anyUnavailableItem = Item.B;

expectedException.expect(VendingMachineInsufficientStockException.class);

expectedException.expectMessage("Insufficient stock " + anyUnavailableItem);

testee.selectItem(anyUnavailableItem);

}

//购买商品时，投入的硬币数量刚好与商品价格相同时，应当给出商品并增加余额

@Test

public void shouldVendSelectedItemWhenSelectItemAndInsertedCoinsAreExactlyPrice(){

Outputter outputter = mock(Outputter.class);

VendingMachine testee = new VendingMachine(outputter, createABank(), createAStorage());

testee.setOn();

testee.insertCoin(Coin.FIFTY\_PENCE.getId());

testee.insertCoin(Coin.TEN\_PENCE.getId());

testee.selectItem(Item.A);

verify(outputter, times(1)).vendItem(Item.A);

}

//购买商品时，若投入的钱高于商品价格，应当给出商品，找零并增加余额

@Test

public void shouldVendSelectedItemAndReturnChangeWhenSelectItemAndInsertedCoinsAreMoreThanPrice(){

Outputter outputter = mock(Outputter.class);

VendingMachine testee = new VendingMachine(outputter, createABank(), createAStorage());

testee.setOn();

testee.insertCoin(Coin.FIFTY\_PENCE.getId());

testee.insertCoin(Coin.ONE\_POUND.getId());

testee.selectItem(Item.A);

verify(outputter).vendItem(Item.A);

outputter.vendItem(Item.A);

ArgumentCaptor<List<Coin>> argument = ArgumentCaptor.forClass((Class)List.class);

verify(outputter).returnCoin(argument.capture());

List<Coin> charge = argument.getValue();

assertThat(charge, is(notNullValue()));

assertThat(charge.stream().mapToInt(Coin::getValueInPence).sum(),is(90));

assertThat(testee.getSelectedItem(), is(nullValue()));

}

//购买商品时，若投入的硬币总价高于商品价格，应当给出商品，找零并增加余额

@Test

public void shouldVendSelectedItemAndReturnChangeWhenInsertCoinsMoreThanPrice(){

Outputter outputter = mock(Outputter.class);

VendingMachine testee = new VendingMachine(outputter, createABank(), createAStorage());

testee.setOn();

testee.insertCoin(Coin.FIFTY\_PENCE.getId());

testee.selectItem(Item.C);

testee.insertCoin(Coin.ONE\_POUND.getId());

testee.insertCoin(Coin.FIFTY\_PENCE.getId());

verify(outputter).vendItem(Item.C);

ArgumentCaptor<List<Coin>> argument = ArgumentCaptor.forClass((Class)List.class);

verify(outputter).returnCoin(argument.capture());

List<Coin> charge = argument.getValue();

assertThat(charge, is(notNullValue()));

assertThat(charge.stream().mapToInt(Coin::getValueInPence).sum(),is(30));

assertThat(testee.getSelectedItem(), is(nullValue()));

}

//购买商品时，若商品有库存，应当能成功选择该商品

@Test

public void shouldSelectItemIfAvailable() {

VendingMachine testee = new VendingMachine(new Outputter(), createABank(), createAStorage());

testee.setOn();

testee.selectItem(Item.A);

assertThat(testee.getSelectedItem(), is(Item.A));

}

//创建售货机的库存：包含两个A商品，一个C商品

private static Storage createAStorage() {

Map<Item, Integer> stock = new EnumMap<>(Item.class);

stock.put(Item.A, 2);

stock.put(Item.C, 1);

return new StorageImpl(stock);

}

//创建售货机的硬币库存：包含5枚10元，5枚20元，5枚50元，5枚100元

private static Bank createABank() {

Map<Coin, Integer> money = new EnumMap<>(Coin.class);

money.put(Coin.TEN\_PENCE, 5);

money.put(Coin.TWENTY\_PENCE, 5);

money.put(Coin.FIFTY\_PENCE, 5);

money.put(Coin.ONE\_POUND, 5);

return new FakeBank(money);

}

}