软件测试 实验报告—JUnit

1、实验目的

使用 Jnit 对附件中的 Rectangle 程序进行测试：根据已经学习的测试思想，对函数 getArea()、getPerimeter()和 findMax()进行测试。

1、在测试中使用审查或者代码走查的方式进行不运行代码的测试。

2、在测试中使用黑盒测试和白盒测试的方法进行测试。

3、使用 Junit 工具辅助进行测试，并学习如何进行自动化的脚本测试。

2、测试范围

2.1 对 Rect::getArea()（.a）进行测试。

2.2 对 Rect::getPerimeter()（.b）进行测试。

2.3 对 Rect::findMax()（.c）进行测试。

3、测试过程

测试计划概述：

1、先进行一次粗略的代码审查，发现代码中可能存在的错误。

2、按照白盒测试的流程设计白盒测试的测试用例和对应的代码方案。

3、按照黑盒测试的流程设计黑盒测试的测试用例和对应的代码方案。

4、进行测试代码的编写。

5、修改源代码，直到通过单元测试。

3.1 代码审查

先大致浏览一遍代码，发现在以下两处代码可能出现问题：

```
    public int getPerimeter() {
        return 2*length + width;
    }
```

按照预期的周长算法，其实现方式应为

```
2*length + 2*width 或者 2*(length + width)
```

同时，观察 Rect 类的两个比较器的实现，分别为

```
    public static class areaCompare implements Comparator<Rect> {
        @Override
        public int compare(Rect o1, Rect o2) {
            // TODO Auto-generated method stub
            if(o1.getArea() < o2.getArea()) {
                return 1;
```

```
            }else if(o1.getArea() == o2.getArea()) {
                return 0;
            }else{
                return -1;
            }
        }
    }
    public static class perimeterCompare implements Comparator<Rect> {
        @Override
        public int compare(Rect o1, Rect o2) {
            // TODO Auto-generated method stub
            if(o1.getPerimeter() > o2.getPerimeter()) {
                return 1;
            }else if(o1.getPerimeter() == o2.getPerimeter()) {
                return 0;
            }else{
                return -1;
            }
        }
    }
```

两者为大于号时，一个返回 1，一个返回-1，因此其中至少有一个是错误的实现方式。

3.2 白盒测试：

3.2.1 Rect::getArea()

由于代码中仅仅涉及一句代码 length * width，只有一种执行过程，因此对于白盒测试并没有什么必要。也无法进行对应的白盒测试。

3.2.2 Rect::getPerimeter()

和上面的一样，也仅仅涉及到一句代码 2 * height + width，因此白盒测试也没有什么必要。

3.2.3 Rect::findMax()

findMax()代码中涉及到 areaComparer()和 perimeterComparer()两个模块。因此 findMax()的执行结果将会依赖于 areaComparer()和 perimeterComparer()的运行结果。为了让测试准确，因此我们需要先测试 areaComparer()和 perimeterComparer()。
观察代码，我们发现 areaComparer()和 perimeterComparer()中分别由三个代码分支。我们设计一个合适的 rectangle 数组：(4,4) ,(8,2) ,(3,6),(1,8)。这组数据无论是边长的比较器还是面积的比较器，都完成了判定覆盖和基路径覆盖。代码将会在后面进行说明。

3.3 黑盒测试：

黑盒测试需要我们在没有代码，只有程序的预期执行逻辑的情况下进行测试。黑盒测试有等价类测试法和边界值测试法两个原则，同时，对于简单的判定逻辑，也有判定树的测试方法。

我们将会在下面给出具体说明。

3.3.1 Rect::getArea()

我们先分析程序的输入和输出，有两个 int 类型的输入，两个 int 类型的输出。在不考虑边界值的情况下，输入值有正数，负数，零三种情况。在 java 中，int 的取值范围是-2^32~2^32-1，在一般情况下也不要求对溢出进行相应的判断。

因此我们给出下列测试用例：(测试用例的写法为输入 1，输入 2，预期结果或者异常)

约束条件 a	约束条件 b	约束条件 a,b	测试用例
a>0	b>0	a>b	(4,3,12),(4,1,4)
		a=b	(4,4,16),(1,1,1)
		a<b	(3,4,12),(1,4,4)
	b=0		(4,0,Exception)
	b<0		(4,-4,Exception)
a=0	b>0		(0,4,Exception)
a<0	b>0		(-4,4,Exception)

因此要测试 getArea()，需要设计 10 组测试用例进行测试。

3.3.2 Rect::getPerimeter()

类似于 Rect::getArea()，可以采用同样的 10 组测试用例如下：

约束条件 a	约束条件 b	约束条件 a,b	测试用例
a>0	b>0	a>b	(4,3,14),(4,1,8)
		a=b	(4,4,16),(1,1,4)
		a<b	(3,4,14),(1,4,8)
	b=0		(4,0,Exception)
	b<0		(4,-4,Exception)
a=0	b>0		(0,4,Exception)
a<0	b>0		(-4,4,Exception)

3.3.3 Rect::findMax()

由于需要找到最大的元素，如果该集合中含有多个最大的元素，则无法确定最后的结果。我们假设最大的矩形只有一个。根据黑盒测试的用例，我们需要使用等价类划分法来测试用例。我们将测试用例分为以下几个类型：输入数据为多个，且含有非法实例；输入数据为多个，不含有非法实例；输入数据为 1 个非法实例；输入数据为 1 个合法实例；输入数据为 0 个。其中输入数据为多个又分为以下三种：最大的周长和面积的实例为同 1 个，最大的周长和面积的实例不是同 1 个。测试用例如下表。

测试 findMax()对于面积的正确性

多个测试用例	第 1 个面积最大	(2,8),(2,2),(2,4)->(2,8)
	中间的面积最大	(2,2),(2,8),(2,4)->(2,8)
	最后一个面积最大	(2,4),(2,2),(2,8)->(2,8)
单个测试用例		(2,2)->(2,2)

无测试用例		empty->error

测试 findMax()对于周长的正确性
可以沿用上面的测试用例，因此不在说明。

3.4 如何编写第一个测试

测试主要分为测试用例的输入和测试结果的判断两部分。其中测试用例的输入使用.json 配置文件来完成。并使用参数化的方法来进行测试。其中需要用到 json 序列化库 Gson。
每次测试完成时，将会打印测试的日志到对应的文件夹（后来不做了）。本次测试将仅使用 Junit 库来进行测试，从而学习 Junit 框架的用法。
由于需要导入外部的库，本次实验将会使用 gradle 包管理器来辅助实验的进行。

测试用例

首先在资源文件夹中 resources 中创建一个叫做 testCase.json 文件，填入测试用例，例如：

```
{
  "testCase": [
    {
      "scope": "RectGetAreaTest",
      "data": [
        {
          "length": 4,
          "width": 3,
          "error": false,
          "result": 12
        }
      ]
    }
  ]
}
```

限于篇幅，只截取了部分测试用例，其他部分请查看源代码。

基础代码

然后在包 org.tty.test1.foundation 包内创建两个辅助类 ResourceManager 和 TestCaseManager，分别用于辅助读取资源和读取测试用例。由于需要从 json 文件中读取数据，因此使用了 gson 序列化库。
代码如下：

```
package org.tty.test1.foundation;

public class ResourceManager {
    private static ResourceManager gResourceManager = null;

    public String getRelativePath(String filePath) {
        return getClass().getResource(filePath).getPath();
    }

    这里省略了单例模式的代码
}
```

```
package org.tty.test1.foundation;

import com.google.gson.Gson;
import com.google.gson.JsonArray;
import com.google.gson.JsonElement;
import com.google.gson.JsonObject;
import com.google.gson.stream.JsonReader;

import java.io.FileNotFoundException;
import java.io.FileReader;
import java.util.ArrayList;
import java.util.List;

public class TestCaseManager {
    private static final String configPath = "/testCase.json";
    private static TestCaseManager gTestCaseManager = null;

    public <T> List<T> loadTestCase(String scope, Class<T> instanceType) throws
FileNotFoundException {
        Gson gson = new Gson();
        JsonReader reader = gson.newJsonReader(new
FileReader(ResourceManager.getInstance().getRelativePath(configPath)));
        JsonObject jsonObject = gson.fromJson(reader, JsonObject.class);
        JsonObject scopeJsonObject = null;
        for (JsonElement object1 : (JsonArray) jsonObject.get("testCase")) {
            JsonObject currentScopeObject = ((JsonObject) object1);
            String currentScope = currentScopeObject.get("scope").getAsString();
            if (currentScope.equals(scope)){
                scopeJsonObject = currentScopeObject;
            }
        }

        assert scopeJsonObject != null;
        List<T> list = new ArrayList<>();

        JsonArray data = (JsonArray) scopeJsonObject.get("data");
        for (JsonElement object2: data) {
            list.add(gson.fromJson(object2, instanceType));
        }

        return list;
    }

    public <T> List<Object[]> loadTestCaseAndMap(String scope, Class<T> instanceType,
Function<T, Object[]> function) throws FileNotFoundException {
        List<T> testCases = this.loadTestCase(scope, instanceType);
        return testCases.stream().map(function).collect(Collectors.toList());
    }

    这里省略了单例模式的代码
}
```

然后需要在 org.tty.test1 中编写对应的测试用例模型 RectTestCase，代码如下：

```
package org.tty.test1.testCase;

public class RectTestCase {
    private int length;
    private int width;
    private int result;
    private boolean error;

    这里省略了getters/setters

    public Object[] toObjects() {
        if (!this.error) {
            return new Object[] { length, width, result};
        } else {
            return new Object[] { length, width, "error"};
        }
    }
}
```

然后按照网上的参数化教程编写测试类，代码如下：

```
package org.tty.test1.test;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import org.tty.test1.Rect;
import org.tty.test1.foundation.TestCaseManager;
import org.tty.test1.testCase.RectTestCase;

import java.io.FileNotFoundException;
import java.util.List;

import static org.junit.jupiter.api.Assertions.assertEquals;

@RunWith(Parameterized.class)
public class RectGetAreaTest {
    private final int length;
    private final int width;
    private final Object result;

    public RectGetAreaTest(int length, int width, Object result){
        this.length = length;
        this.width = width;
        this.result = result;
    }

    @Parameterized.Parameters(name = "{index}: x={0},y={1},result={2}")
    public static List<Object[]> data() throws FileNotFoundException {
        TestCaseManager testCaseManager = TestCaseManager.getInstance();

        return testCaseManager.loadTestCaseAndMap("RectGetAreaTest",
RectTestCase.class, RectTestCase::toObjects);
```

```
    }

    @Test
    public void test() {
        assertEquals(rectArea(length, width), result);
    }

    private Object rectArea(int length, int width) {
        Rect rect = new Rect(length, width);
        try {
            return rect.getArea();
        } catch (Exception e) {
            return "error";
        }
    }
}
```

测试结果

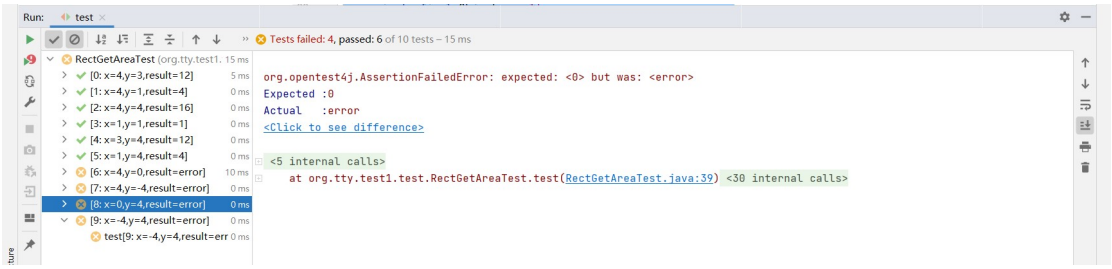

图 1 Rect::getArea()黑盒测试的测试用例和测试结果

3.5 编写后续代码和测试用例

后面的代码和第 1 个测试代码编写类似，所有的测试用例都位于 testCase.json 文件中，下面给出代码未修改前的测试结果。

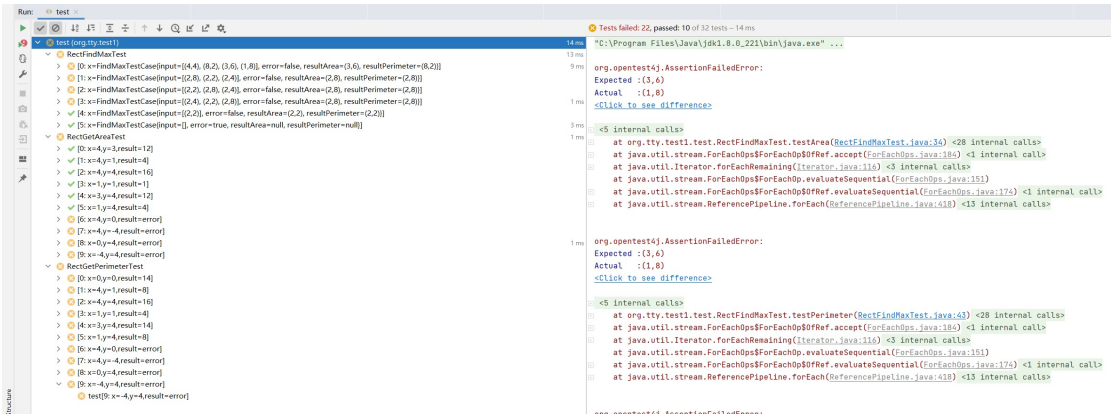

图 2 修改代码前的测试结果图

3.6 修改 Rect 代码直到通过预期的测试

首先修改构造器，让它在进行非法构造是报错。代码如下：

```
public Rect(int length, int width) {
    if (length <= 0){
```

```
        throw new IllegalArgumentException("length 应当大于 0");
    }
    if (width <= 0){
        throw new IllegalArgumentException("width 应当大于 0");
    }
    this.length = length;
    this.width = width;
}
```

然后将 getPerimeter 修改为正确的实现，代码如下：

```
public int getPerimeter() {
    return 2*length + 2*width;
}
```

然后将 areaComparer 中的代码修改为正确实现，代码如下：

```
public static class areaCompare implements Comparator<Rect> {
    @Override
    public int compare(Rect o1, Rect o2) {
        // TODO Auto-generated method stub
        if(o1.getArea() > o2.getArea()) {
            return 1;
        }else if(o1.getArea() == o2.getArea()) {
            return 0;
        }else{
            return -1;
        }
    }
}
```

然后运行测试

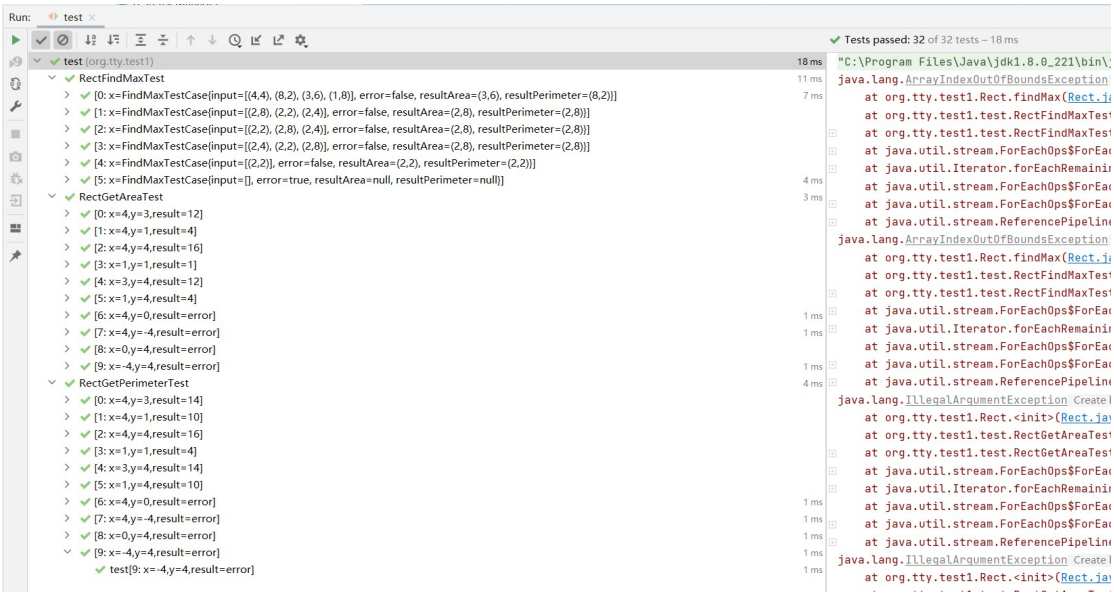

图 3 修改代码后的测试结果图

4、总结与分析

本次测试使用了 Junit 的测试技术来进行单元测试，为了能够配置测试用例，因此使用了外部文件的方式来导入测试用例。并使用了 Junit 中的参数化测试来进行覆盖性全的测试。

测试实例时，经常使用到测试用例，而测试用例一旦过多，就不便于使用程序硬编码的方式来输入测试用例，因此使用外部文件就显得很有必要。
但是，相对来说，本测试用例距离自动化测试还有很大的距离，虽然做到了测试用例的自动导入，但是缺乏成熟框架的支持，因此编写测试用例的时间也占用了较大的时间。同时，不能很好的应对回归测试和较为复杂的测试。