

**2020年春季学期  
计算机学院《软件构造》课程**

**Lab 4实验报告**

|  |  |
| --- | --- |
| 姓名 | 梁书育 |
| 学号 | 1180300208 |
| 班号 | 1803002 |
| 电子邮件 | 951009723@qq.com |
| 手机号码 | 18845773527 |

**目录**

[1 实验目标概述 1](#_Toc39518185)

[2 实验环境配置 1](#_Toc39518186)

[3 实验过程 2](#_Toc39518187)

[3.1 Error and Exception Handling 2](#_Toc39518188)

[3.1.1 处理输入文本中的三类错误 2](#_Toc39518189)

[3.1.2 处理客户端操作时产生的异常 2](#_Toc39518190)

[3.2 Assertion and Defensive Programming 2](#_Toc39518191)

[3.2.1 checkRep()检查rep invariants 2](#_Toc39518192)

[3.2.2 Assertion/异常机制来保障pre-/post-condition 2](#_Toc39518193)

[3.2.3 你的代码的防御式策略概述 2](#_Toc39518194)

[3.3 Logging 2](#_Toc39518195)

[3.3.1 异常处理的日志功能 2](#_Toc39518196)

[3.3.2 应用层操作的日志功能 3](#_Toc39518197)

[3.3.3 日志查询功能 3](#_Toc39518198)

[3.4 Testing for Robustness and Correctness 3](#_Toc39518199)

[3.4.1 Testing strategy 3](#_Toc39518200)

[3.4.2 测试用例设计 3](#_Toc39518201)

[3.4.3 测试运行结果与EclEmma覆盖度报告 3](#_Toc39518202)

[3.5 SpotBugs tool 4](#_Toc39518203)

[3.6 Debugging 5](#_Toc39518204)

[3.6.1 EventManager程序 5](#_Toc39518205)

[3.6.2 LowestPrice程序 5](#_Toc39518206)

[3.6.3 FlightClient/Flight/Plane程序 6](#_Toc39518207)

[4 实验进度记录 6](#_Toc39518208)

[5 实验过程中遇到的困难与解决途径 6](#_Toc39518209)

[6 实验过程中收获的经验、教训、感想 7](#_Toc39518210)

[6.1 实验过程中收获的经验和教训 7](#_Toc39518211)

[6.2 针对以下方面的感受 7](#_Toc39518212)

# 实验目标概述

本次实验重点训练学生面向健壮性和正确性的编程技能，利用错误和异常处 理、断言与防御式编程技术、日志/断点等调试技术、黑盒测试编程技术，使程序 可在不同的健壮性/正确性需求下能恰当的处理各种例外与错误情况，在出错后 可优雅的退出或继续执行，发现错误之后可有效的定位错误并做出修改。

实验针对 Lab 3 中写好的 ADT 代码和基于该 ADT 的三个应用的代码，使用以下技术进行改造，提高其健壮性和正确性：

·错误处理

·异常处理

·Assertion和防御式编程

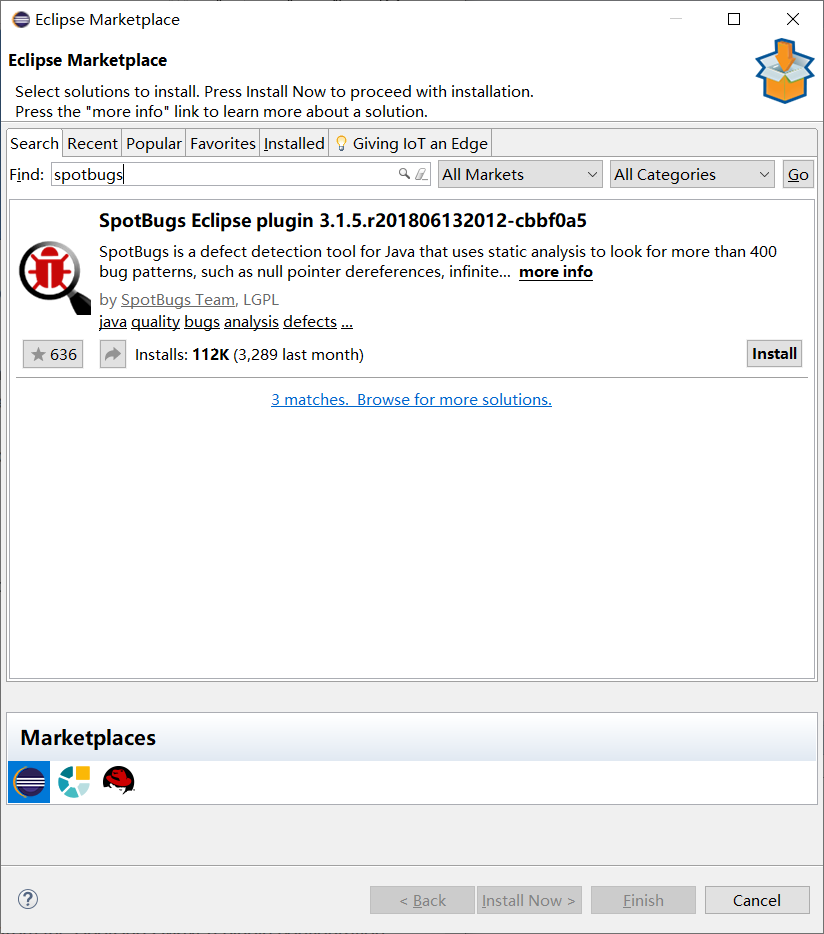
·日志

·调试技术

·黑盒测试及代码覆盖度

# 实验环境配置

本次实验要求安装SpotBugs，在eclipse中的Marketplace中找到SpotBugs插件，进行安装。



在这里给出你的GitHub Lab4仓库的URL地址（Lab4-学号）。

<https://github.com/ComputerScienceHIT/Lab4-1180300208.git>

# 实验过程

## Error and Exception Handling

### 处理输入文本中的三类错误

三类错误分别为：输入文件中存在不符合语法规则的语句、输入文件中存在标签完全一样的元素以及输入文件中各元素之间的依赖关系不正确。下面分别细说。

第一种错误：输入文件中存在不符合语法规则的语句

1、输入日期格式不合法

DateInputIlleagalException

2、航班名称要求2位大写字母和2-4位数字但却使用了非大写字母或小于2位或超过4位数字

FlightNameIllegalException

3、最多1位小数但使用了2位小数，或者范围不在[0, 30]内

PlaneAgeException

4、第一位不为 N 或 B，或者后面不是四位数字

PlaneIDException

5、位置名称不合法，出现了非字母

LocationNameException

6、飞机座位数范围为在[50,600]之外

PlaneSeatNumberException

7、机型信息中含有除大小写字母和数字以外的其他字符

PlaneTypeException

8、输入的时间格式不合法

TimeInputIlleagalException

第二种错误：输入文件中存在标签完全一样的元素

存在多个航班计划项的“日期,航班号”信息完全一样的情况

SameLabelException

第三种错误：输入文件中各元素之间的依赖关系不正确

1、第一行出现的航班日期与内部出现的起飞时间中的日期不一致

FlightDateException

2、同一个航班号，虽然日期不同，但其出发或到达机场、 出发或到达时间有差异

FlightMessageException

3、在不同航班计划项中出现编号一样的飞机，但飞机的类型、座位数或机龄却不一致

PlaneMessageException

4、降落时间中的日期与航班日期差距大于 1 天。

OverOneDayException

### 处理客户端操作时产生的异常

1、在删除某资源的时候，如果有尚未结束的计划项正在占用该资源

ResourceOccupiedException

2、在删除某位置的时候，如果有尚未结束的计划项会在该位置执行

LocationOccupiedException

3、在取消某计划项的时候，如果该计划项的当前状态不允许取消

EntryCancelException

4、在为某计划项分配某资源的时候，如果分配后会导致与已有的其他计划项产生“资源独占冲突”

ResourceConflictException

5、在为某计划项变更位置的时候，如果变更后会导致与已有的其他计划项产生“位置独占冲突”

LocationConflictException

## Assertion and Defensive Programming

### checkRep()检查rep invariants

1、Timeslot

// Abstraction function:

// 一个时间对，包含起始时间和终止时间

// Representation invariant:

// 起始时间一定要在终止时间之前

// Safety from rep exposure:

// 使用的变量均为不可变类型

**public** **void** checkRep() {

**assert** startTime.isBefore(endTime);

}

2、Location

// Abstraction function:

// 表示一个位置信息

// Representation invariant:

// 经度纬度应该符合实际

// 即经度范围为-180 ~ 180（用负数表示西经，用正数表示东经）

// 维度范围为-90 ~ 90（用负数表示南纬，用正数表示北纬）

// Safety from rep exposure:

// 使用的均为不可变类型的变量

**public** **void** checkRep() {

**assert** longitude >= -180;

**assert** longitude <= 180;

**assert** latitude >= -90;

**assert** latitude <= 90;

}

注：还有一些类的ri没有具体要求，没有编写checkRep，在这里就不再放出了。

同样还有资源类：Plane、Carriage、Teacher由于其中的信息应该由外层的用户来决定，因此也没有在这些类中使用checkRep来检查ri。

### Assertion/异常机制来保障pre-/post-condition

1、BlockableEntryImpl

// Abstraction function:

// 表示可阻塞时间，由总的起始时间和阻塞起始时间构成

// Representation invariant:

// 阻塞起始时间应当在总起止时间的范围内。

// Safety from rep exposure:

// 使用的rep均为不可变类型

**public** **void** checkRep() {

**assert** totalTimeslot.getStartTime().isBefore(getTotalEndTime());

**for**(Timeslot t : blockTimeslotList) {

**assert** t.getStartTime().isBefore(t.getStartTime());

**assert** t.getEndTime().isAfter(t.getEndTime());

}

}

2、DoubleLocationEntryImpl

// Abstraction function:

// 两个位置分别表示起始位置和终止位置

// Representation invariant:

// 起始位置和终止位置应不相同

// Safety from rep exposure:

// 使用的rep均为不可变类型

**public** **void** checkRep() {

**assert** !start.equals(end);

}

3、MultipleLocationEntryImpl

// Abstraction function:

// 表示计划项发生的起止位置以及中间位置

// Representation invariant:

// 起始位置、中间位置应各不相同

// Safety from rep exposure:

// 使用的rep均为不可变类型

**public** **void** checkRep() {

**assert** !start.equals(end);

**for**(Location l : midLocList) {

**assert** !start.equals(l);

**assert** !end.equals(l);

**for**(Location m : midLocList) {

**if**(m == l) **continue**;

**else** **assert** !l.equals(m);

}

}

}

4、UnblockableEntryImpl

// Abstraction function:

// 表示计划项的起止时间。

// Representation invariant:

// 起始时间应当早于终止时间。

// Safety from rep exposure:

// 使用的rep均为不可变类型

**public** **void** checkRep() {

**assert** timeslot.getStartTime().isBefore(timeslot.getEndTime());

}

### 你的代码的防御式策略概述

1、在代码中使用不可变类型的数据，防止数据被篡改

2、代码在没有必要的情况下不提供mutator，也就不存在数据篡改的可能

3、在有mutator的情况下，用checkRep来对RI进行检测

## Logging

首先定义日志的格式，日志格式自定义的代码如下：

**public** String format(LogRecord record) {

StringBuilder builder = **new** StringBuilder(1000);

builder.append(df.format(**new** Date(record.getMillis()))).append("\n");

builder.append("[").append(record.getLevel()).append("]\n");

builder.append("[").append(record.getSourceClassName()).append("]\n");

builder.append("[").append(record.getSourceMethodName()).append("]\n");

builder.append("[").append(record.getThrown()).append("]\n");

builder.append(formatMessage(record));

builder.append("\n\n");

**return** builder.toString();

}

### 异常处理的日志功能

在异常捕获的catch语句中进行异常的日志记录，其中记录的日志级别应为WARNING级别，例如：

**try** {

*logger*.log(Level.***INFO***, "从文件" + fileName + "中读入数据");

generateFlightApp(fileName);

} **catch** (FlightNameIllegalException e) {

System.***out***.println("在读入第" + number + "组数据时发生错误！");

e.printMessage();

**this**.illegal = **true**;

*logger*.log(Level.***WARNING***, "航班名称要求2位大写字母和2-4位数字但却使用了非大 写字母或小于2位或超过4位数字", e);

}

### 应用层操作的日志功能

在执行的操作方法中对操作进行记录，其中日志等级应为INFO级别，例如：

**public** **boolean** addResource(Plane resource) {

*logger*.log(Level.***INFO***, "添加了" + resource.getPlaneID() + "飞机");

**return** fso.addResource(resource);

}

### 日志查询功能

为每个app添加日志查询功能，用户可以用不同的过滤条件：日志时间、日志级别、涉及操作，来对日志进行查询。

核心代码如下：

**while** ((lineTxt = br.readLine()) != **null**) {

lineNum = lineNum % 7;

**switch** (lineNum) {

**case** 0:

**if** (time.toLowerCase().equals("null") || time.equals(lineTxt)) {

out += lineTxt + "\n";

} **else** {

canPrint = **false**;

}

**break**;

**case** 1:

**if** (level.toLowerCase().equals("null") || ("[" + level + "]").equals(lineTxt)) {

out += "重要性：" + lineTxt + "\n";

} **else** {

canPrint = **false**;

}

**break**;

**case** 3:

**if** (operation.toLowerCase().equals("null") || ("[" + operation + "]").equals(lineTxt)) {

out += "调用方法：" + lineTxt + "\n";

} **else** {

canPrint = **false**;

}

**break**;

**case** 5:

out += "具体操作：" + lineTxt + "\n";

**break**;

**case** 6:

**if** (canPrint) {

System.***out***.println(out);

} **else** {

canPrint = **true**;

}

out = "";

**break**;

}

lineNum++;

}

查询结果将会打印在控制台上。

## Testing for Robustness and Correctness

### Testing strategy

利用等价类和边界值的思想来进行测试，再加上非法用例来构成测试。程序源代码和测试代码完成后，再进行覆盖度的检测，发现有没有检测全面的地方再设计新的测试用例，来使测试覆盖度提高，达到更加全面测试的效果。

### 测试用例设计

设计了大量的测试用例，核心思想就是定价类和边界值的思想，下面拿CourseData中的DataOperation的测试用例进行举例：

@Test

**public** **void** testDataOperation() {

String name1 = "English1";

String name2 = "English2";

String name3 = "English3";

Location loc1 = **new** Location("正心231", **false**);

Location loc2 = **new** Location("正心232", **false**);

Location loc3 = **new** Location("正心233", **false**);

Timeslot timeslot1 = **new** Timeslot(2020,5,15,14, 5, 2020, 5, 15, 15, 7);

Timeslot timeslot2 = **new** Timeslot(2020,5,14,14, 5, 2020, 5, 14, 15, 7);

Timeslot timeslot3 = **new** Timeslot(2020,5,13,14, 5, 2020, 5, 13, 15, 7);

CourseEntry ce1 = **new** CourseFactory().getEntry(name1, loc1, timeslot1);

CourseEntry ce2 = **new** CourseFactory().getEntry(name2, loc2, timeslot2);

CourseEntry ce3 = **new** CourseFactory().getEntry(name3, loc3, timeslot3);

List<CourseEntry> list = **new** ArrayList<CourseEntry>();

list.add(ce1); list.add(ce2); list.add(ce3);

CourseData cd = **new** CourseData();

cd.addData(ce1); cd.addData(ce2); cd.addData(ce3);

*assertEquals*(list, cd.getDataList());

cd.deleteData(ce3);

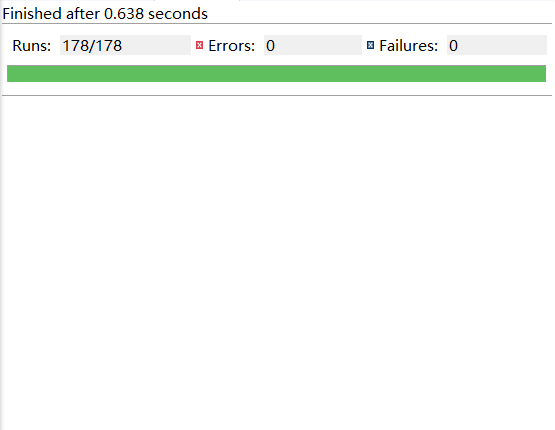
list.remove(ce3);

*assertEquals*(list, cd.getDataList());

}

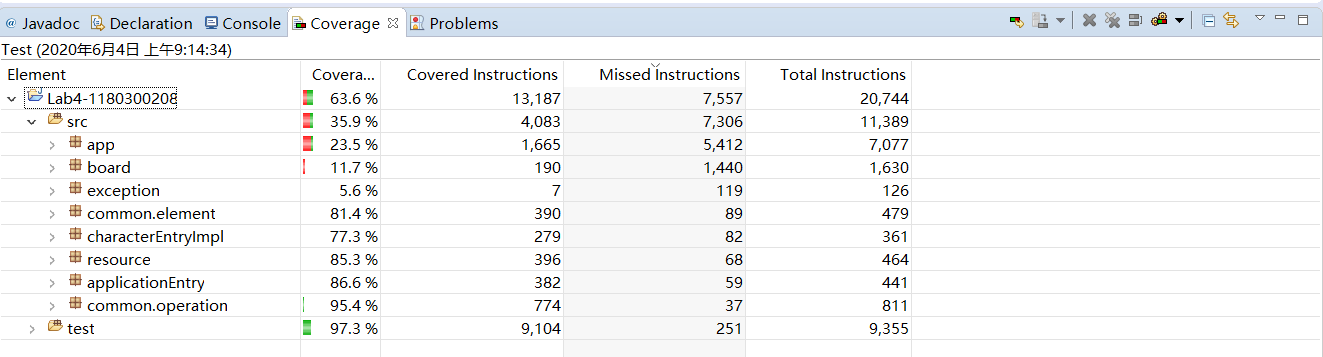
### 测试运行结果与EclEmma覆盖度报告

运行测试，得到如下结果：

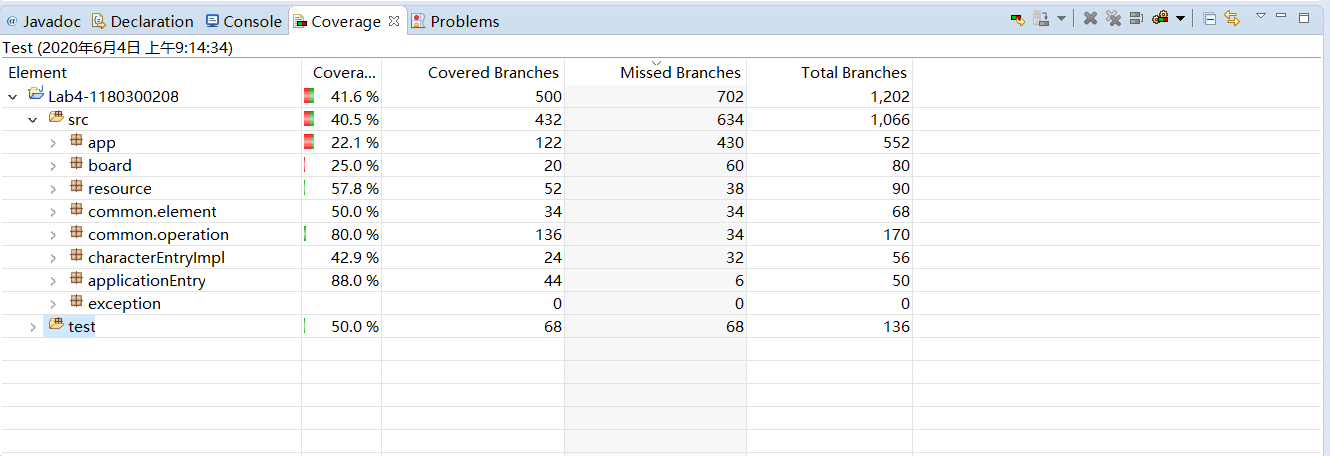


测试覆盖度如下：

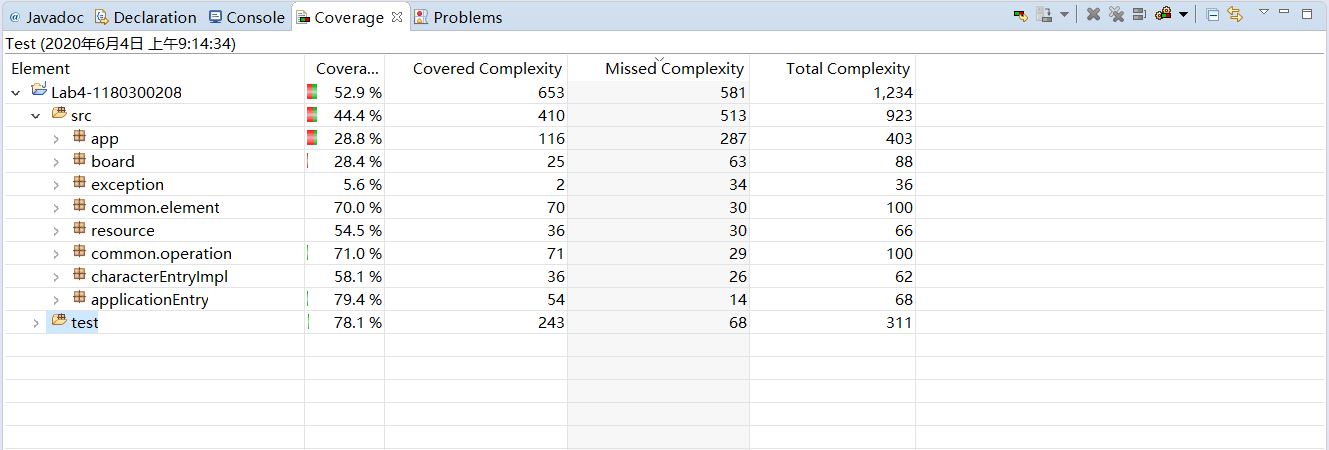
语句覆盖度：



分支覆盖度：

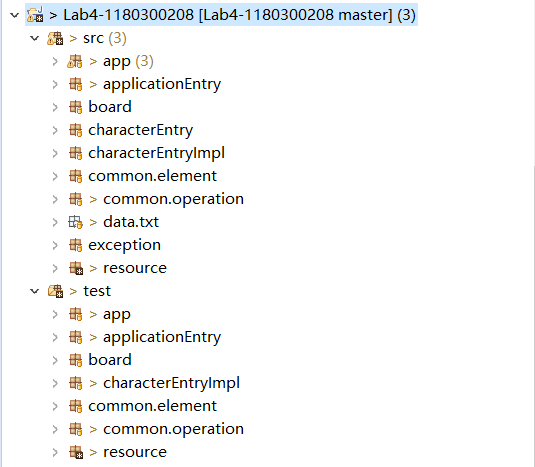


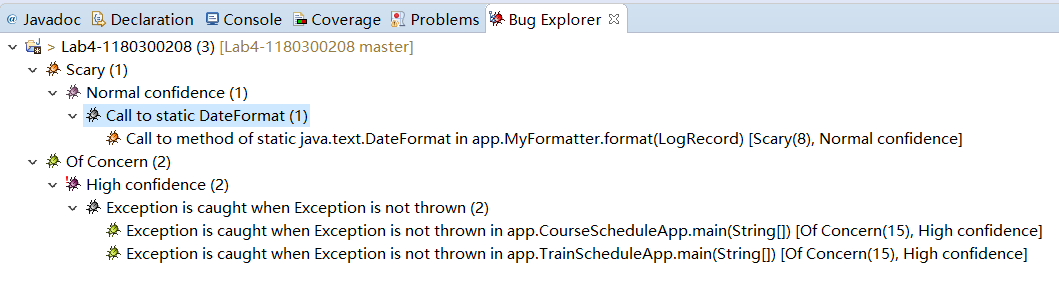
路径覆盖度：



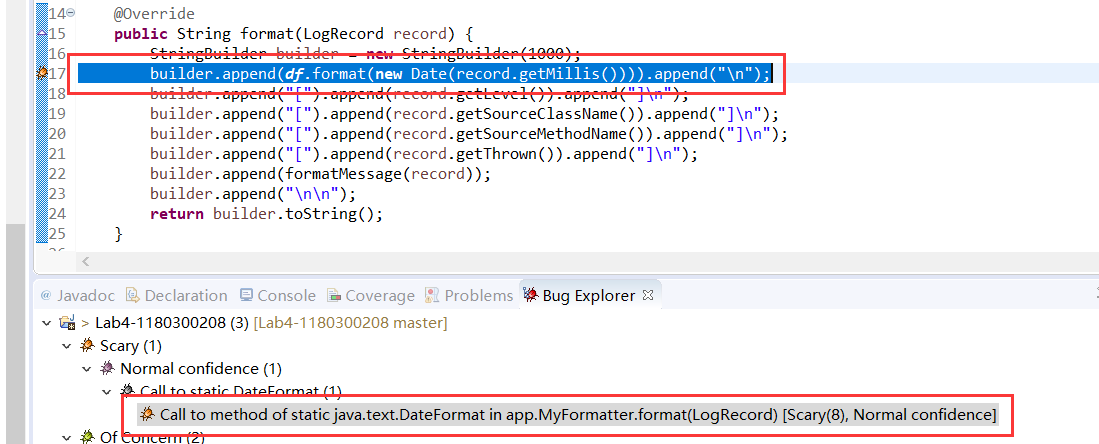
详细的测试覆盖度报告放在了doc文件夹中的testReport文件夹中。

## SpotBugs tool





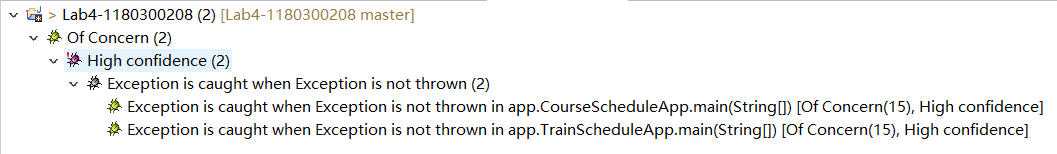
将SpotBugs的识别等级设置为15，运行SpotBugs，一共发现了3个错误（说明我的代码质量还是可以的哈），一个一个查看，发现第一个是Scary类型的，错误如下所示：



可以看到问题在于使用的df，即日期的格式使用了static类型，而静态类型在多线程的情况下是由危险的，把static删掉，解决了这个潜在的问题。修改后的代码如下：



剩下的两个问题是相同的，如下图所示：



SpotBugs认为我使用了没有必要的try-catch语句，但实际上我是用这个语句来过滤掉每个小功能中可能出现的输入不合法导致的问题，因此，这个语句是必要的，因此我选择忽略这条错误提示，不删除这里的try-catch语句。

## Debugging

### EventManager程序

一、理解待调试程序的代码思想

该程序的思想是对每一个小时进行标记，每个小时的初始值设为0，如果该小时作为活动的起始时间，则将其标记值加一，如果为终止时间，则将其标记值减一，标记完成后，按照时间顺序将标记值一一相加，相加得到的值就是该小时同时进行的活动数量，将活动数量最大的时刻记录下来，就是要返回的该天最大同时活动数。

二、发现并定位错误的过程

仔细阅读原来的错误代码，可以发现原来的代码有如下的几个问题：

1、保存标记的TreeMap temp没有初始化；

2、没有考虑该题要求记录的是每一天的同时最大活动数；

3、没有对标记的小时进行加一减一的处理。

三、你如何修正错误

针对以上的问题，分别进行以下修正：

1、将static类型的book的static去掉，并为类EventManager添加构造方法，在构造方法中对temp进行初始化，初始化时为将不同的天区分开，需要将一年中每一天的每一小时进行初始化，具体操作为将从1到365\*24的所有数字标记为0。

2、为了考虑每一天，将不同天的同一小时区分开，具体做法是将小时定义为：小时+(一年中的天数-1)\*24，如第二天的10时应该实际存储为10+(2-1)\*24=34

3、将原本的

*temp*.put(start, *temp*.get(start));

*temp*.put(end, *temp*.get(end));

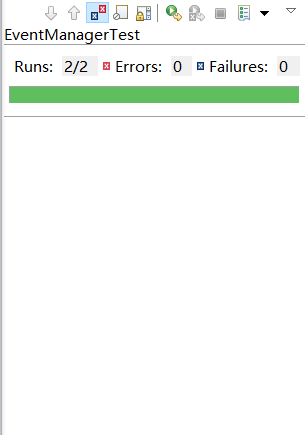
改为：

*temp*.put(start + (day - 1) \* 24, *temp*.get(start + (day - 1) \* 24) + 1);

*temp*.put(end + (day - 1) \* 24, *temp*.get(end + (day - 1) \* 24) - 1);

四、修复之后的测试结果

通过了设置的测试用例。



### LowestPrice程序

一、理解待调试程序的代码思想

该程序的思想为通过递归调用，来不断进行筛选合适的策略，通过这种方式来与已有的最小值进行比对，最终得到总体的最小的price。

二、发现并定位错误的过程

仔细阅读原来的错误代码，可以发现原来的代码有如下的几个问题：

1、可以看到在遍历clone列表时发生了数组越界，检查代码发现在循环时发生了从0遍历到needs.size()；

2、发现在对clone数组遍历后，判断条件if(j == needs.size())并不能起到判断作用；

3、代码错误的将diff == 0的情况过滤掉了导致会出问题；

4、在进行更新res时出现错误。

三、你如何修正错误

针对以上的问题，分别进行以下修正：

1、将for (j = 0; j <= needs.size(); j++)改为for (j = 0; j < needs.size(); j++)；

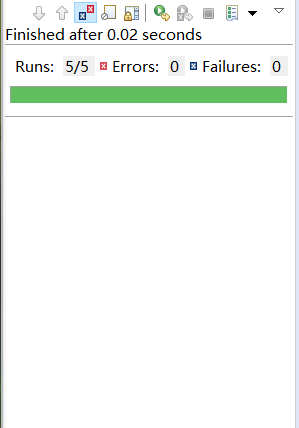
2、将continue语句改为break；

3、将if(diff <= 0)改为if(diff < 0);

4、将res = Math.min(res, s.get(j - 1) + shopping(price, special, clone))改为res = Math.min(res, s.get(j) + shopping(price, special, clone))。

四、修复之后的测试结果

如下图，通过了设计的测试用例。



### FlightClient/Flight/Plane程序

一、理解待调试程序的代码思想

用planes给flights分配飞机资源，使用随机的机制，为flights中的航线进行资源分配，分配前要检测一下该飞机与其他已经分配的资源是否有重复的情况，如果所有的flight都可以被分配则返回true，如果有一个航线对所有的飞机都有冲突，则立刻返回false。

二、发现并定位错误的过程

1、随机机制难以设置停机条件；

2、bConflict作为判断某一航线是否与所有飞机有冲突的变量，定义在循环中不合适。

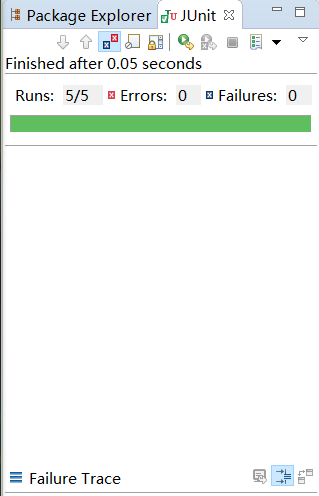
三、你如何修正错误

1、考虑到随机难以设置停机条件，因此改用遍历

2、将bConfict的定义放在循环以外，使其在出了循环之后可以判断最终是否可以分配到飞机。

四、修复之后的测试结果

如下图，通过了设计的测试用例。



# 实验进度记录

|  |  |  |  |
| --- | --- | --- | --- |
| 日期 | 时间段 | 计划任务 | 实际完成情况 |
| 2020/5/21 | 14:00-23:00 | 准备实验环境，完成第一部分 | 完成 |
| 2020/5/30 | 14:30-23:30 | 添加测试策略，完成测试部分 | 完成 |
| 2020/6/3 | 14:00-22:30 | 完成logging部分，进行覆盖度测试 | 完成 |
| 2020/6/5 | 14:30-19:30 | 用SpotBugs查找潜在bug | 完成 |
| 2020/6/7 | 14:00-23:00 | 完成debugging部分，完善实验报告 | 完成 |

# 实验过程中遇到的困难与解决途径

|  |  |
| --- | --- |
| 遇到的难点 | 解决途径 |
| 在进行日志记录时遇到了问题 | 请教同学+上网查阅相关资料 |
| 完成debug部分的时候理解第一个也就是EventManager的代码没有理解 | 在Piazza上看到了同学们对这一问题的提问以及老师的回答，理解了算法的思想 |
| 日志记录logging的过程中，日志的查询功能出现问题 | 在网上查阅相关资料，完成了这一部分 |

# 实验过程中收获的经验、教训、感想

## 实验过程中收获的经验和教训

经验：在编写程序的过程中一定要注意代码的测试以及防御式编程的思想，由于我在lab3中在这些方面做的比较到位，因此在lab4中就十分轻松，在今后的工作也是如此，有良好的编程习惯一定可以让工作变得更加轻松。

教训：在编程前一定要求充分的考虑，要有规划，否则之后的修改会非常麻烦。

## 针对以下方面的感受

1. 健壮性和正确性，二者对编程中程序员的思路有什么不同的影响？

健壮性的思路是要求程序猿编写的程序能够对用户的任何输入做出反应，对于正确的输入应该给出正确的结果，而对于非法的输入则要优雅的提示；而正确性侧重的思路是程序对于正确的输入应当给出正确的结果，当遇到错误的输入时应当及早停止程序。

1. 为了应对1%可能出现的错误或异常，需要增加很多行的代码，这是否划算？（考虑这个反例：民航飞机上为何不安装降落伞？）

我认为这一问题需要考虑程序面向的领域，如果是十分需要安全而且出不起问题的程序，那么添加代码也是必须的；而如果程序面向的对象对程序的安全性没有那么高的要求，则无需考虑这些错误异常，毕竟重启一下程序也不是很难的事情。

1. “让自己的程序能应对更多的异常情况”和“让客户端/程序的用户承担确保正确性的职责”，二者有什么差异？你在哪些编程场景下会考虑遵循前者、在哪些场景下考虑遵循后者？

二者的差异在于前者要求程序猿做的更多，而后者则把问题的责任交给了用户。我认为在消费性的软件的开发中我会选择前者，因为用户不一定具有很好的专业知识，如果因为操作不规范多次导致程序崩溃，不利于软件的推广，而一些专业程序上我选择后者，因为使用者有更强的专业能力，具备分析问题解决问题的能力。

1. 过分谨慎的“防御”（excessively defensive）真的有必要吗？你如何看待过分防御所带来的性能损耗？如何在二者之间取得平衡？

我认为这一问题有没有必要还是要看应用场景，如果程序涉及到重大的经济、安全问题，那么牺牲一些性能也是可以接受的，而如果只是一般使用，那么可以适当减少防御，来提升性能

1. 通过调试发现并定位错误，你自己的编程经历中有总结出一些有效的方法吗？请分享之。Assertion和log技术是否会帮助你更有效的定位错误？

总结出来的有效方式就是最好使用自己定义的格式，这样能够更好的反映错误信息。Assertion和log技术可以帮助我更好的定位错误。

1. 怎么才是“充分的测试”？代码覆盖度100%是否就意味着100%充分的测试？

充分的测试是指测试样例遍及了所有的情况，代码覆盖度100%也不一定代表进行了100%充分的测试，因为用户的输入很有可能会超过程序的处理范围，同时不同机器、不同系统上，程序的运行表现也不一定相同。

1. Debug一个错误的程序，有乐趣吗？体验一下无注释、无文档的程序修改。

很有乐趣，发现别人程序中的错误其实也是在提高自己的编程能力。无注释、无文档的程序修改就没有什么乐趣可言了，因为每个人的编程风格都不一样，没有注释意味着要一步步看别人的代码，这就太艰难了。

1. 关于本实验的工作量、难度、deadline。

本次实验工作量和前一个实验相比，工作量、难度都不算大，ddl刚好够用。

1. 到目前为止你对《软件构造》课程的评价和建议。

软件构造课程作为一个以实验为核心的课程，我本人还是非常喜欢的，可以学到不少东西。

1. 期末考试临近，你对占成绩60%的闭卷考试有什么预期？

希望学校和学院还是能够审慎考虑考试形式和时间，同时也希望能够下调考试成绩所占的比例。