

**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 | 周牧云 |
| 学号 | 1180300315 |
| 班号 | 1836101 |
| 电子邮件 | 1036314134@qq.com |
| 手机号码 | 13912263240 |

**目录**

[1 实验目标概述 1](#_Toc42416161)

[2 实验环境配置 1](#_Toc42416162)

[3 实验过程 3](#_Toc42416163)

[3.1 Error and Exception Handling 3](#_Toc42416164)

[3.1.1 处理输入文本中的三类错误 3](#_Toc42416165)

[3.1.2 处理客户端操作时产生的异常 4](#_Toc42416166)

[3.2 Assertion and Defensive Programming 4](#_Toc42416167)

[3.2.1 checkRep()检查rep invariants 4](#_Toc42416168)

[3.2.2 Assertion/异常机制来保障pre-/post-condition 6](#_Toc42416169)

[3.2.3 你的代码的防御式策略概述 6](#_Toc42416170)

[3.3 Logging 6](#_Toc42416171)

[3.3.1 异常处理的日志功能 6](#_Toc42416172)

[3.3.2 应用层操作的日志功能 6](#_Toc42416173)

[3.3.3 日志查询功能 6](#_Toc42416174)

[3.4 Testing for Robustness and Correctness 7](#_Toc42416175)

[3.4.1 Testing strategy 7](#_Toc42416176)

[3.4.2 测试用例设计 7](#_Toc42416177)

[3.4.3 测试运行结果与EclEmma覆盖度报告 7](#_Toc42416178)

[3.5 SpotBugs tool 7](#_Toc42416179)

[3.6 Debugging 7](#_Toc42416180)

[3.6.1 EventManager程序 7](#_Toc42416181)

[3.6.2 LowestPrice程序 8](#_Toc42416182)

[3.6.3 FlightClient/Flight/Plane程序 8](#_Toc42416183)

[4 实验进度记录 8](#_Toc42416184)

[5 实验过程中遇到的困难与解决途径 8](#_Toc42416185)

[6 实验过程中收获的经验、教训、感想 9](#_Toc42416186)

[6.1 实验过程中收获的经验和教训 9](#_Toc42416187)

[6.2 针对以下方面的感受 9](#_Toc42416188)

# 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。

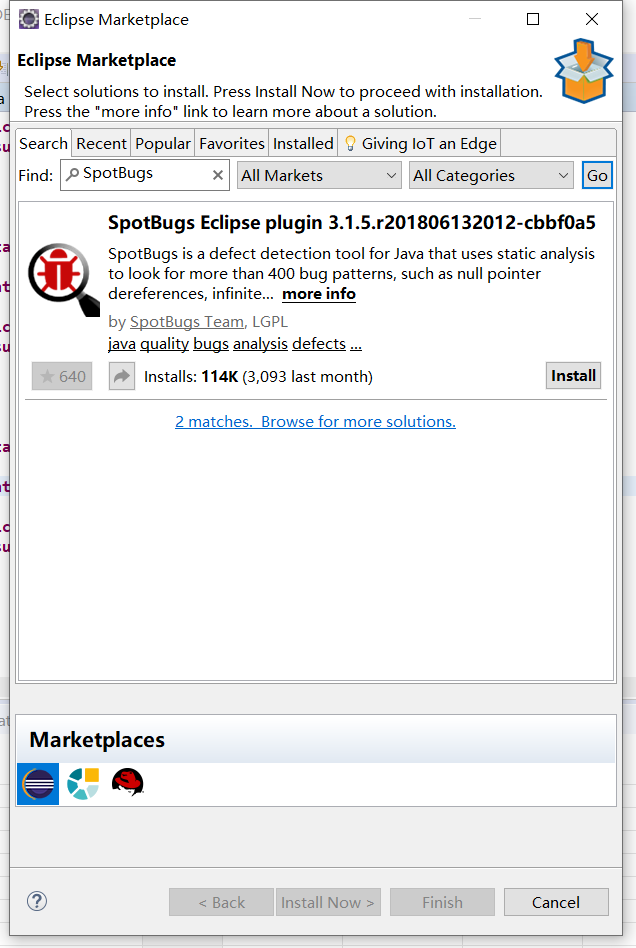
实验针对Lab 3中写好的ADT代码和基于该ADT的三个应用的代码，使用以下技术进行改造，提高其健壮性和正确性：

* 错误处理
* 异常处理
* 防御式编程和Assertion
* 日志
* 调试技术
* 黑盒测试及代码覆盖度

# 实验环境配置

本次实验需要使用SpotBugs

在eclipse中点击help，点击eclipse marketplace，搜索SpotBugs，再点击install。



在这里给出你的GitHub Lab4仓库的URL地址（Lab4-学号）。

https://github.com/ComputerScienceHIT/Lab4-1180300315

# 实验过程

请仔细对照实验手册，针对每一项任务，在下面各节中记录你的实验过程、阐述你的设计思路和问题求解思路，可辅之以示意图或关键源代码加以说明（但千万不要把你的源代码全部粘贴过来！）。

## Error and Exception Handling

### 处理输入文本中的三类错误

1、输入文件中存在不符合语法规则的语句

1）航班信息没有以Flight:……开头

会抛出异常"航班表头信息格式错误"

2）始发机场信息没有以“DepartureAirport:机场名”为格式表示

会抛出异常"航班始发机场表示错误"

3）到达机场信息没有以“ArrivalAirport:机场名”为格式表示

会抛出异常"航班到达机场表示错误"

4）出发时间没有以“xxxx-xx-xx xx:xx”为格式表示

会抛出异常"航班始发时间格式不对"

5）到达时间没有以“xxxx-xx-xx xx:xx”为格式表示

会抛出异常"航班到达时间格式不对"

6）飞机编号没有以“Plane:xxxxx”为格式表示

会抛出异常"航班飞机编号格式不对"

7）飞机种类没有以“Type:xxxx”为格式表示

会抛出异常"航班飞机种类格式不对"

8)飞机座位数没有以”Seats:xxx”为格式表示

会抛出异常"航班飞机座位数格式不对"

9）飞机机龄没有以“Age:23.7”为格式表示

会抛出异常"航班飞机机龄格式不对"

2、输入文件中存在标签完全一样的元素

如果存在相同航班号的航班，会抛出异常"航班已经存在"

3、输入文件中各元素之间的依赖关系不正确

1）如果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

会抛出异常"航班信息出发时间不匹配"

2）如果出发时间晚于到达时间

会抛出异常"航班时间出发晚于到达"

3）如果出发时间与到达时间相差一天以上

会抛出异常"航班时间出发与到达相差一天以上"

4）如果两个航班出现同一架飞机，但是信息不一致

会抛出异常"飞机信息不一致"

### 处理客户端操作时产生的异常

1）如果要删除的资源已经被分配，会抛出异常"资源已被分配"。如果该资源不存在，会抛出异常"资源不存在"。

2）如果要删除的位置已经被分配，会抛出异常"位置已被分配"，如果该位置没被加入，会抛出异常"位置不存在"。

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，会抛出异常"该计划目前状态不能取消"。

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”，会抛出异常"资源独占冲突"

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”，会抛出异常"位置独占冲突"。我的项目里只有课程表允许修改地址，所以该异常由CourseEntry类抛出

## Assertion and Defensive Programming

### checkRep()检查rep invariants

对CommonPlanningEntry类，检查计划名是否为空

**protected** **void** checkRep() {

**assert** **this**.name != **null**;

}

对Location类，检查位置名是否为空

**protected** **void** checkRep() {

**assert** **this**.name != **null**;

}

对PlanningEntryCollection类，检查其中的每一个PlanningEntry类中的资源是否是存在的。

**protected** **void** checkRep() {

**for** (PlanningEntry<R> i : **this**.plans) {

**for** (R j: i.getResources()) {

**assert** resources.contains(j);

}

}

}

对Resource类，检查资源名是否为空

**protected** **void** checkRep() {

**assert** **this**.name != **null**;

}

对State类，检查状态名是否为空

**protected** **void** checkRep() {

**assert** **this**.name != **null**;

}

对Time类，检查时间是否合法。

**protected** **void** checkRep() {

**assert** **this**.year >= 0;

**assert** **this**.month >= 0 && **this**.month <= 12;

**assert** **this**.day >= 0 && **this**.day <= 30;

**assert** **this**.hour >= 0 && **this**.hour < 24;

**assert** **this**.minute >= 0 && **this**.minute < 60;

}

对Timeslot类，检查开始时间是否晚于结束时间

**protected** **void** checkRep() {

**assert** **this**.end.compareto(**this**.begin) >= 0;

}

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

1. 增加资源时，如果资源已经存在，或者资源名为空，则会抛出异常。
2. 删除资源时，如果资源不存在，则会抛出异常。
3. 增加位置时，如果位置已经存在，或者位置名为空，则会抛出异常。
4. 删除位置时，如果位置不存在，则会抛出异常。
5. 添加计划时，如果计划已经存在，则会抛出异常。如果时间格式输入错误，也会抛出异常。如果位置信息没有加入，也会抛出异常。

### 你的代码的防御式策略概述

APP类会将用户输入的内容转变为符合ADT的pre-condition的形式给ADT读入，让ADT去执行操作。

如果用户的输入违反了pre-condition，ADT会抛出异常，APP会将异常捕捉并报给用户。此次输入将不会进行任何操作，用户需要重新输入正确的操作。

## Logging

使用java自带的java logging实现

配置文件为log.properties, 放在路径“P3//Exception//”下

日志所有内容会输出到“txt\\Log.log”中去。

### 异常处理的日志功能

当ADT抛出异常后，APP类会进行捕捉，同时将异常信息记录到日志中，例如：

2020-06-06 17:36:29 P2.FlightSchedule.FlightScheduleAPP print

信息: 航班始发机场表示错误

### 应用层操作的日志功能

当用户执行一次操作后，APP类会将该操作记录到日志中去，例如：

2020-06-06 17:36:43 P2.FlightSchedule.FlightScheduleAPP print

信息: 添加飞机B9218成功

### 日志查询功能

能力不够……确实没做出来，我认为我的日志写的是有问题的，格式几乎一致，没有办法按照题目所说的方法进行查询操作……

## Testing for Robustness and Correctness

### Testing strategy

采用等价类和边界值思想

例如，对PlanningEntry类的测试

//对PlanningEntry类

//对状态改变函数，测试默认状态下能不能改变

//对getState(), 测试默认状态是否为"WAITING"

//对addResource(), 测试resources为空时能否加入，不为空时能否加入，是否能避免重复加入。

//对removeResource(), 测试能否正确删除，资源不存在时是否会误删

//对addLocation(), 测试locations为空时能否加入，不为空时能否加入，是否能避免重复加入。

//对Timeslot类

//对islegal()，测试能否判断出输入是否合法

//对isConflict()，测试时间段不冲突和冲突两种情况

//对addTimeslot()，测试timeslots为空时能否加入，不为空时能否加入，是否避免重复加入。

//对getBegintime()和getBegintime()，测试能否正确提取开始与结束时间

对于每个方法，采用等价类思想测试不同情况下的输入是否正确，

对于测试用例，采用边界值的方法将所有边界情况考虑进去，例如对addResource()，考虑当前计划无资源、有资源、有重复资源的情况，将这些边界条件均进行测试。

其他的测试基本大同小异

### 测试用例设计

例如，对PlanningEntry类的测试

@Test(expected = AssertionError.**class**)

**public** **void** testAssertionsEnabled() {

**assert** **false**;

}

@Test

**public** **void** testgetName() {

String name = "wuhuqifei";

PlanningEntry<Resource> plan = PlanningEntry.*getNewPlanningEntry*(name);

*assertEquals*(plan.getName(), name);

}

@Test

**public** **void** teststart() {

String name = "wuhuqifei";

PlanningEntry<Resource> plan = PlanningEntry.*getNewPlanningEntry*(name);

*assertEquals*(plan.start(), **false**);

}

@Test

**public** **void** testcancel() {

String name = "wuhuqifei";

PlanningEntry<Resource> plan = PlanningEntry.*getNewPlanningEntry*(name);

*assertEquals*(plan.cancel(), **true**);

}

@Test

**public** **void** testblock() {

String name = "wuhuqifei";

PlanningEntry<Resource> plan = PlanningEntry.*getNewPlanningEntry*(name);

*assertEquals*(plan.block(), **false**);

}

@Test

**public** **void** testcomplete() {

String name = "wuhuqifei";

PlanningEntry<Resource> plan = PlanningEntry.*getNewPlanningEntry*(name);

*assertEquals*(plan.complete(), **false**);

}

@Test

**public** **void** testgetState() {

String name = "wuhuqifei";

PlanningEntry<Resource> plan = PlanningEntry.*getNewPlanningEntry*(name);

*assertEquals*(plan.getState().getName(), "WAITING");

}

@Test

**public** **void** testResource() {

String name = "wuhuqifei";

PlanningEntry<Resource> plan = PlanningEntry.*getNewPlanningEntry*(name);

*assertEquals*(plan.getResources().isEmpty(), **true**);

*assertEquals*(plan.addResource(**new** Resource("feiji1")), **true**);

*assertEquals*(plan.getResources().size(), 1);

*assertEquals*(plan.addResource(**new** Resource("feiji2")), **true**);

*assertEquals*(plan.getResources().size(), 2);

*assertEquals*(plan.addResource(**new** Resource("feiji2")), **false**);

*assertEquals*(plan.getResources().size(), 2);

*assertEquals*(plan.removeResource("feiji2"), **true**);

*assertEquals*(plan.getResources().size(), 1);

*assertEquals*(plan.removeResource("feiji2"), **false**);

}

@Test

**public** **void** testLocation() {

String name = "wuhuqifei";

PlanningEntry<Resource> plan = PlanningEntry.*getNewPlanningEntry*(name);

Location location1 = Location.*getNewLocation*("wuhu");

Location location2 = Location.*getNewLocation*("nantong");

*assertEquals*(plan.addLocation(location1), **true**);

*assertEquals*(plan.addLocation(location2), **true**);

*assertEquals*(plan.addLocation(location2), **false**);

*assertEquals*(plan.getLocations().get(0).getName().equals(location1.getName()), **true**);

*assertEquals*(plan.getLocation("wuhu").getName().equals("wuhu"), **true**);

*assertEquals*(plan.removeLocation(location1.getName()), **true**);

*assertEquals*(plan.removeLocation(location1.getName()), **false**);

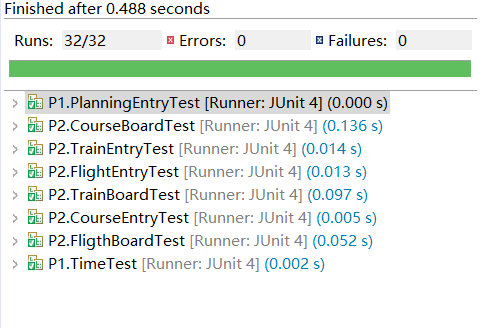
}

按照测试策略，尽量覆盖了每一行代码，覆盖了每一个函数。

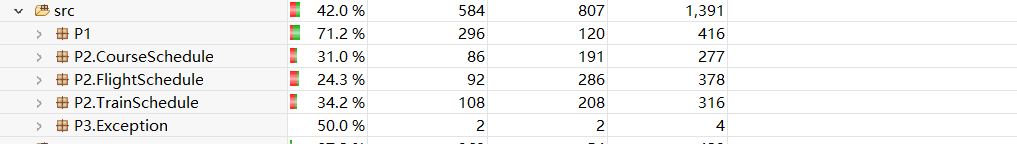
其余的测试用例也均如此

### 测试运行结果与EclEmma覆盖度报告

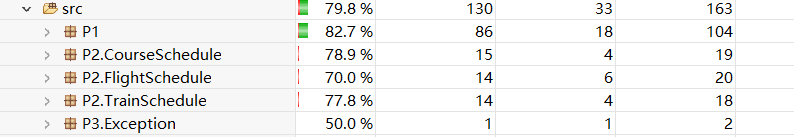
测试结果如下：



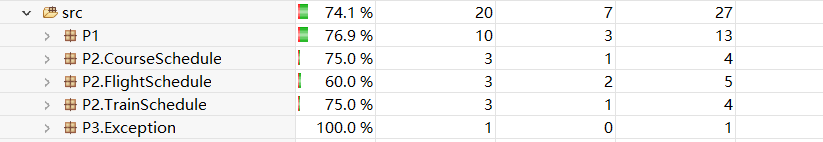
Line Counters



Method Counters



Tpye Counrers



EclEmma覆盖度报告生成在txt文件夹下

由于P2的包里包含有主程序等类，没有办法进行覆盖，导致覆盖了不高。

## SpotBugs tool

似乎并没有发现什么错误。。。倒是帮我找到了3.6里的问题



这里应该是对字符串使用了==进行比较。

应该使用equals()进行比较

很好奇为什么静态分析没有发现.

## Debugging

### EventManager程序

book方法的static要删除，它不是静态方法

在遍历temp的value时会产生空指针异常。

老实说有点看不懂这个函数在干嘛，第一遍put的时候就使用get,肯定返回的是null，然后又把null给put进去，这不肯定出错吗。

而且整个程序和day一点关系没有…

我猜这个程序大概是想每次增加一个计划后先存起来，然后把每天的最大重叠都算出来，最后返回最大值。但是实现很乱。

我的想法是，按照它的思路用treemap存，只不过key是day，value是这一天所有的时间段的集合。

每次book会输入一个新时间，先存进map，然后算出这一天的最大重叠，和总最大重叠比较，更新总最大重叠，然后返回总最大重叠。

计算重叠数的方法是，遍历这一天的24小时，计算每小时的重叠数，取最大值，这便是这一天的最大重叠数。由于每次只增加一次计划，只会改变这一天的最大重叠数，所以我用一个max储存一个总最大重叠数，如果这一天的最大重叠数没有超过max，则不用更新总最大重叠。这样就不用每次都把每一天的最大重叠数都算一次了。

**static** TreeMap<Integer, List<Time>> *temp* = **new** TreeMap<Integer, List<Time>>();

**public** **int** max = 1;

/\*\*

\* **@param** day the day now

\* **@param** start start time of the event to be added, should be in [0, 24)

\* **@param** end end time of the event to be added, should be in (0, 24]

\* **@return** the max number of concurrent events in the same hour

\*/

**public** **int** book(**int** day, **int** start, **int** end) {

**if**(*temp*.get(day) == **null**) {

List<Time> k = **new** ArrayList<>();

k.add(**new** Time(start, end));

*temp*.put(day, k);

**return** max;

}

*temp*.get(day).add(**new** Time(start, end));

**int** ans = 1;

**for** (**int** i = 0; i <= 24; i++) {

**int** active = 0;

**for** (Time t : *temp*.get(day)) {

**if** (t.getStart() < i && i < t.getEnd()) {

active++;

}

}

**if**(active > ans) {

ans = active;

}

}

**if**(ans > max) {

**this**.max = ans;

}

**return** max;

}

**private** **class** Time {

**private** **final** **int** start;

**private** **final** **int** end;

**public** Time(**int** start, **int** end) {

**this**.start = start;

**this**.end = end;

}

**public** **int** getStart() {

**return** **this**.start;

}

**public** **int** getEnd() {

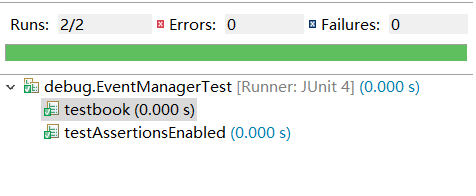
**return** **this**.end;

}

}

测试数据就是它给的例子再增加了一部分内容，测试了如果往其他天数增加计划能否正确返回冲突数

测试结果如下：



### LowestPrice程序

该程序是计算最少价格

先假设价格为全部用原价购买的价格，然后遍历所有的优惠，计算该优惠是否可用。如果可用，则递归调用该方法，直到无优惠可用时，计算当前价格，更新当前最低价格，全部遍历完毕后，返回最低价格。

1）



存在数组越界

等号应当删除

2）



判断优惠是否可用应该判断使用这次优惠是否会多购买物品，如果正好买完也是可行的，所以diff可以等于零，等号删除。同时，如果这个优惠不可用，就不用计算这个优惠后面的价格了，直接看下一个优惠，所以continue应该改为break；



3）

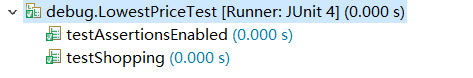


j == nees.size()说明这次优惠已经判断出是可行的，那么应该对比使用这次优惠和不使用这次优惠的价格，进行递归调用，优惠价格是该列表最后一位，所以将index改为s.size()-1



测试数据是题目给的例子，分别计算了有优惠和没优惠两种情况下的最少价格

测试结果如下：



### FlightClient/Flight/Plane程序

该程序用来给航班分配飞机并保证没有冲突。

先将所有航班按日期排序，然后枚举航班，再枚举所有飞机，如果这架飞机没有别的航班使用或者别的航班使用了但两个航班的时间不冲突，那么就把这架飞机分配给该航班。

如果全部分配完毕返回true，不能分配完成返回false

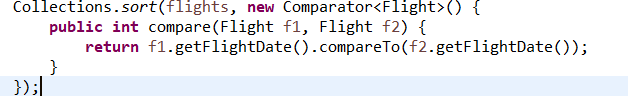
1）

字符串不应该用==，而应该用equals()方法

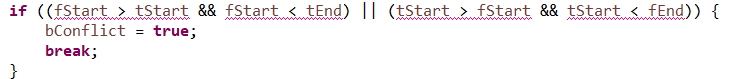


2）

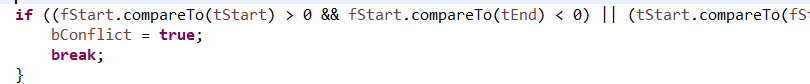
对于排序，我们需要自己写compare（）,用航班日期排序



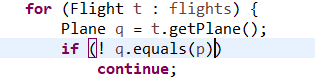
3）



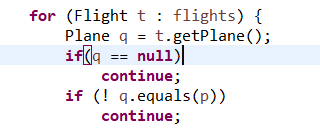
Calendar类不能用>和<来比较



4）

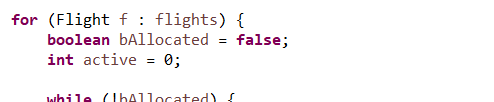


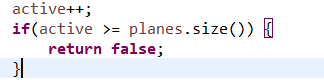
航班存在没有飞机的情况，会导致空指针异常，应当添加为空的判断



5）

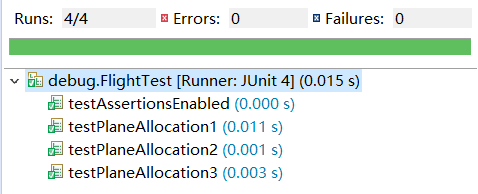
如果某个航班找不到适合的飞机，会陷入死循环，因此如果找不到合适的飞机，应当直接结束程序，返回失败





测试数据测试了航班时间冲突飞机不够导致失败的情况，飞机足够的情况，和航班不冲突使得飞机可以应用到多个航班的情况

测试结果如下：



# 实验进度记录

请使用表格方式记录你的进度情况，以超过半小时的连续编程时间为一行。

每次结束编程时，请向该表格中增加一行。不要事后胡乱填写。

不要嫌烦，该表格可帮助你汇总你在每个任务上付出的时间和精力，发现自己不擅长的任务，后续有意识的弥补。

|  |  |  |  |
| --- | --- | --- | --- |
| 日期 | 时间段 | 计划任务 | 实际完成情况 |
| 2020-5-19 | 18：30-22：00 | 完成3.1 | 未完成 |
| 2020-5-23 | 14：00-18：00 | 完成3.1与3.2 | 未完成 |
| 2020-5-23 | 19：00-21：00 | 完成3.1与3.2 | 完成 |
| 2020-5-26 | 18：30-20：15 | 学习logging，完成3.3 | 未完成 |
| 2020-5-30 | 9：00-11：00 | 完成3.3 | 未完成 |
| 2020-5-30 | 14：00-18：10 | 完成3.3 | 日志查询未完成 |
| 2020-6-2 | 18：30-20：15 | 完成3.4 | 完成 |
| 2020-6-6 | 13：00-20：00 | 完成3.5与3.6 | 3.6未完成 |
| 2020-6-7 | 14：00-17：00 | 完成3.6 | 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 |
| 遇到大量nullpointerexception并且错误提示不明显 | 最终通过EclEmma分析出了错误代码的位置，发现是checkRep（）的问题， |
| 不会使用日志 | 上网寻找logging的教程并自己尝试了简单的应用 |
| 查找冲突如果直接用之前的代码需要先把资源加入，但报错时没法把错误加入的资源删除 | 新写了删除资源的代码，查找冲突时如果发现冲突先把资源删除再进行报错 |

# 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

## 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？

正确性只需要关注自己的代码有没有问题，相对简单，而考虑健壮性需要有发散性的思维，要把客户当成一个专门找茬的人，考虑各种奇奇怪怪的情况，对程序员考验更大。

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？）

我觉得还是看需求吧，如果这个程序要求非常严格的健壮性，还是必须得考虑那1%的，但如果要求不那么严格的话其实也无所谓。。。健壮性是要有的，但不一定要健壮到特别极端的情况，那样性价比太低。

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？

前者需要程序员付出更多，后者需要用户付出更多。

我觉得大多数情况下都应该考虑遵循前者。毕竟写代码是程序员的工作，保证软件的健壮性就是程序员的职责之一。

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？

我认为还是有必要的。

过分防御虽然会有性能损耗，但是会保证程序的安全性，相比之下还是安全性的优先级更高，性能可以被取舍。

我认为还是优先确保安全性的前提下在尽量提高程序的性能。

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？

我比较喜欢用EclEmma跑测试数据发现错误，通过查看代码的覆盖程度可以发现程序执行到哪一步之后出现了问题，可以极大地缩小找bug的范围。一般来说程序在哪里停下来就说明哪里出现了问题。

Assertion和log技术对我而言还不够熟悉，目前的帮助不算大，未来等运用得更加熟练之后应该会帮助很大。

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？

充分的测试应当遵循等价类和边界值原则，把所有的情况覆盖到。并不是指代码覆盖度到100%就意味着充分，有些代码仅仅跑一次并不能看出问题，需要通过等价类与边界值把所有情况覆盖到才能保证没问题。

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。

对我而言乐趣不大。希望每个程序员能把自己写的程序都debug好，不要麻烦别人-\_-！

无注释无文档修改程序真的折磨，需要自己去理解程序在干什么，如果程序很大的话真的特别麻烦。

1. 关于本实验的工作量、难度、deadline。

都比之前少了，但是ddl也更早了，对我而言还是有难度。尤其是要接触之前没了解过的日志功能，手很生。

1. 到目前为止你对《软件构造》课程的评价和建议。

学分高，内容多，难度大，压力大，希望这样的课少一点少一点少一点（划掉）

1. 期末考试临近，你对占成绩60%的闭卷考试有什么预期？

希望不要太难