

**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 | cycleke |
| 学号 | xxxxxxxxxx |
| 班号 | xxxxxxx |
| 电子邮件 | [cycleke@gmail.com](mailto:cycleke@gmail.com) |
| 手机号码 | xxxxxxxxxx |

**目录**

[1 实验目标概述 3](#_Toc39518185)

[2 实验环境配置 3](#_Toc39518186)

[3 实验过程 4](#_Toc39518187)

[3.1 Error and Exception Handling 4](#_Toc39518188)

[3.1.1 处理输入文本中的三类错误 4](#_Toc39518189)

[3.1.2 处理客户端操作时产生的异常 8](#_Toc39518190)

[3.2 Assertion and Defensive Programming 9](#_Toc39518191)

[3.2.1 checkRep()检查rep invariants 9](#_Toc39518192)

[3.2.2 Assertion/异常机制来保障pre-/post-condition 10](#_Toc39518193)

[3.2.3 你的代码的防御式策略概述 11](#_Toc39518194)

[3.3 Logging 12](#_Toc39518195)

[3.3.1 异常处理的日志功能 12](#_Toc39518196)

[3.3.2 应用层操作的日志功能 13](#_Toc39518197)

[3.3.3 日志查询功能 14](#_Toc39518198)

[3.4 Testing for Robustness and Correctness 15](#_Toc39518199)

[3.4.1 Testing strategy 15](#_Toc39518200)

[3.4.2 测试用例设计 17](#_Toc39518201)

[3.4.3 测试运行结果与EclEmma覆盖度报告 19](#_Toc39518202)

[3.5 SpotBugs tool 21](#_Toc39518203)

[3.6 Debugging 21](#_Toc39518204)

[3.6.1 EventManager程序 21](#_Toc39518205)

[3.6.2 LowestPrice程序 21](#_Toc39518206)

[3.6.3 FlightClient/Flight/Plane程序 21](#_Toc39518207)

[4 实验进度记录 22](#_Toc39518208)

[5 实验过程中遇到的困难与解决途径 22](#_Toc39518209)

[6 实验过程中收获的经验、教训、感想 22](#_Toc39518210)

[6.1 实验过程中收获的经验和教训 22](#_Toc39518211)

[6.2 针对以下方面的感受 22](#_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

此次实验需要额外安装一个Eclipse插件SpotBugs用于Java代码的静态分析。与EclEmma类似，我可以在Eclipse Marketplace中安装。

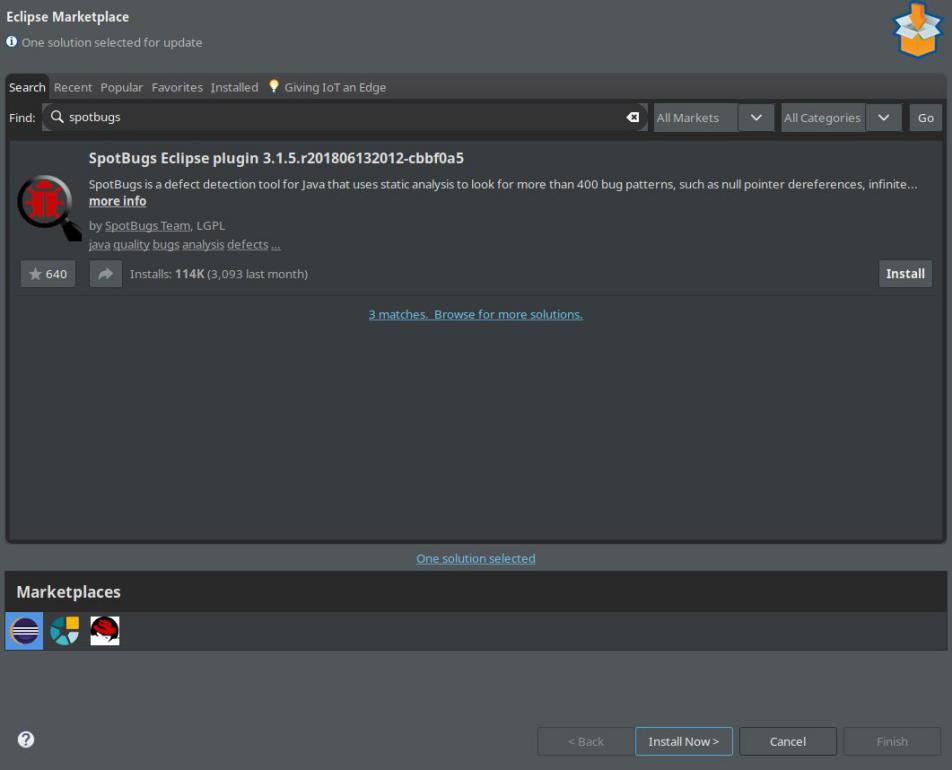


图 2-1Eclipse Marketplace中搜索SpotBugs

我的GitHub Lab4仓库的URL地址：

[https://github.com/ComputerScienceHIT/Lab4-xxxxxxxxxx](https://github.com/ComputerScienceHIT/Lab4-1180100217)

# 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**

在本节中，我需要实现一些自定义的checked exception类型，用于表示程序中可能出现的一些常见错误类型,为代码可能发生的问题提供新的含义，区分代码运行时可能出现的相似问题，或给出程序中一组共性错误的特殊含义。

对于各个场景，我需要为可能出现的错误设计出对应的Exception并且在调用这些可以抛出异常的函数的地方使用try-catch-finally机制来优雅地处理这些异常。

### **处理输入文本中的三类错误**

在Lab3中，我就已经为parse函数设计了一个通用的Exception类型：FileParseException用于表示在解析文件的过程中出现的不符合语法的异常。在此次实验中我为FileParseException派生了一些子类用于表示描述更加精细化的异常以应对输入文本中的三类错误。

第一类错误：输入文件中存在不符合语法规则的语句。Lab3的实验文档提供了输入文件的标准语法。参考语法我可以设想出许多异常情况。在此次实验中，我对于10种可能的“不符合语法规则的情况”设计了对应的异常类。它们都是FileParseException的子类。而对于其他的解析过程中出现的异常，我仍然使用FileParseException来抛出异常。

|  |  |
| --- | --- |
| “不符合语法规则”的情况 | 对应的异常类 |
| 某个元素定义的标签非法 | MismatchedKeyException |
| 元素定义的内容格式与语法规范不一致 | ValueFormatException |
| 元素定义的内容不存在 | ValueNotFoundException |
| 元素定义中分量的数目与规范不符 | WrongNumberOfValuesException |
| 元素定义中分量的次序与规范不符 | ValuesOutOfOrderException |
| 左括号缺失 | LeftBracketNotFoundException |
| 右括号缺失 | RightBracketNotFoundException |
| 标签或值的大小写不匹配 | CaseNotMatchException |
| 日期的格式错误 | DateFormatException |
| 日期时间的格式错误 | DateTimeFormatException |
| 数值超出了规定的范围（主要是seat和age标签） | WrongNumberOfValuesException |

每一个异常会提示错误发生在哪个文件的第几行以及期望的内容是什么。例如在MismatchedKeyException中，它的构造函数中就包含了三个参数：file来表示解析的是哪个文件，line用来表示错误发生在哪一行，expected用来表示期望的标签是什么。

1. **public** **class** MismatchedKeyException **extends** FileParseException {
3. **private** **static** **final** **long** serialVersionUID = 5454864838101114543L;
5. /\*\*
6. \* Constructor of {@link MismatchedKeyException}.
7. \*
8. \* @param file     the file where failed
9. \* @param line     the line where failed
10. \* @param expected the expected key
11. \*/
12. **public** MismatchedKeyException(File file, **int** line, String expected) {
13. **super**("Parse file " + file.getName() + " failed on line " + line + ". " + "Expected Key: " + expected);
14. }
15. }

为了在正确的位置抛出对应的异常，我需要对实验三中的代码重构。在实验三中，我只用考虑抛出一种异常FileParseException，而现在我又多了10种更为精细的异常类型。所以原本的粗糙的过于庞大的parse函数（在实验三中单个函数达到了302行）就不合适了，如果直接在原本的函数中添加新的逻辑会使得代码的可读性大大下降，同时代码出错的概率也会大大增加。我需要对parse进行拆分，将它更改为若干个小函数和一个parse来调用它们。

我使用parseValueTokens函数来解析一个键值对，它会返回一个值序列（因为部分标签对应的值不止一个）。它首先会解析出标签和值，之后判断对应的标签是否与期望的是否相同。如果不同还会判断是否为大小写错误，之后抛出对应的异常；如果相同那么会解析出对应的值序列。如果值序列为空，那么还需要抛出元素对应的内容不存在的异常。对应的流程图如下：

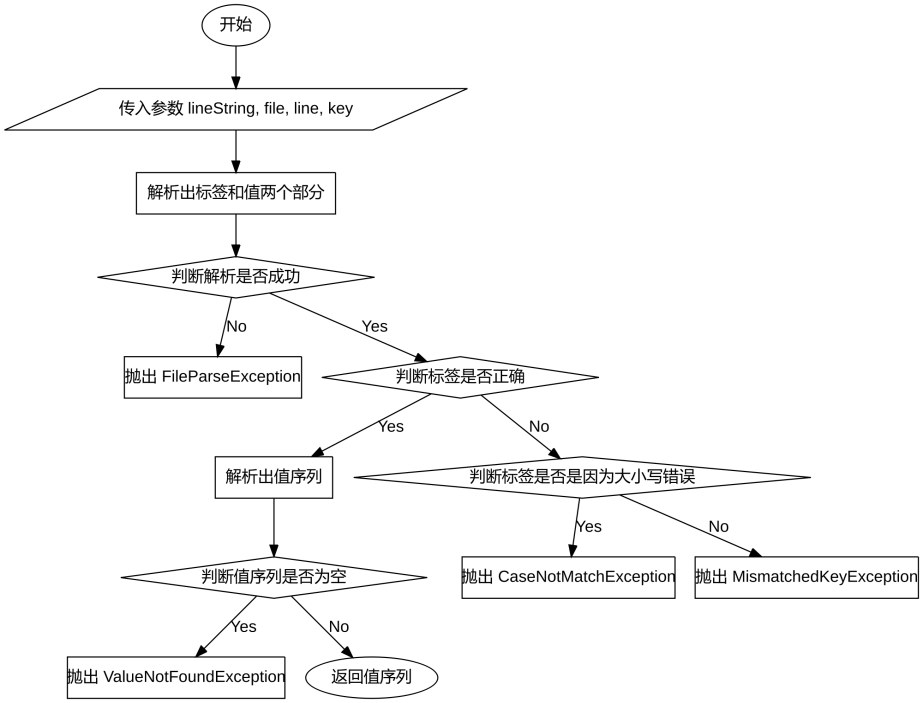


图 3-1parseValueTokens函数的流程图

而对于其他的函数，它们也抛出可能遇到的各种异常。如getNonemptyLine函数会抛出FileParseException异常；parseFlight函数会抛出WrongNumberOfValuesException、CaseNotMatchException、ValueFormatException、ValuesOutOfOrderException和DateFormatException异常；parseBracket会抛出LeftBracketNotFoundException和RightBracketNotFoundException异常；parseAirport函数会抛出WrongNumberOfValuesException和ValueNotFoundException异常；parseDateTime函数会抛出WrongNumberOfValuesException和DateTimeFormatException异常；parsePlaneNumber函数会抛出WrongNumberOfValuesException、CaseNotMatchException和ValueFormatException异常；parsePlaneType函数会抛出WrongNumberOfValuesException和ValueFormatException异常；parsePlaneSeats函数会抛出WrongNumberOfValuesException、ValueFormatException和NumberOutOfRangeException异常；parsePlaneAge函数会抛出WrongNumberOfValuesException、ValueFormatException和NumberOutOfRangeException异常。

第二类异常：输入文件中存在标签完全一样的元素。这类异常就是存在重复的航班计划项。为了描述此类错误，我设计了一个新的FileParseException的子异常类SameElementException。当存在两个日期和编号完全相同的航班项时，我的程序就会抛出一个SameElementException异常。

1. String dateFlightNumberPair = date + "," + number;
2. **if** (dateFlightNumberLineMap.containsKey(dateFlightNumberPair)) {
3. **throw** **new** SameElementException(file, dateFlightNumberLineMap.get(dateFlightNumberPair), flightLine);
4. }

第三类异常：输入文件中各元素之间的依赖关系不正确。此类异常是指输入文件的内容存在逻辑错误。这类错误没有语法错误，但是存在语义错误，如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；降落时间中的日期与航班日期差距大于1天等。为此我设计了4中对应的异常。

|  |  |
| --- | --- |
| 异常对应的情况 | 异常类 |
| 航班的出发时间未早于降落时间 | DepartureTimeIsNotSmallerThanArrivalTimeException |
| 降落时间中的日期与航班日期差距大于1天 | TimeGapTooLargeException |
| 存在两架飞机编号相同但是属性不同 | PlaneAttributeConflictException |
| 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 | MismatchedDateException |

在解析的过程中，我解析出一个元素后就马上判断是否存在异常的情况，使错误尽早地暴露。具体的代码如下：

1. if (!date.equals(departureTime.substring(0, 10))) {
2. // 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
3. throw new MismatchedDateException(file, line[0]);
4. }
5. ...
6. Timeslot timeslot;
7. try {
8. timeslot = Timeslot.parse(departureTime, arrivalTime);
9. } catch (AssertionError e) {
10. // 出发时间 >= 降落时间，利用 {@link Timeslot#checkRep()}
11. throw new DepartureTimeIsNotSmallerThanArrivalTimeException(file, line[0]);
12. }
13. Duration duration = Duration.between(timeslot.getStartTime(), timeslot.getEndTime());
14. if (duration.getSeconds() > SECONDS\_PER\_DAY) {
15. // 降落时间中的日期与航班日期差距大于1天
16. throw new TimeGapTooLargeException(file, line[0]);
17. }
18. ...
19. Plane plane = new Plane(planeNumber, planeType, seats, age);
20. if (planeMap.containsKey(planeNumber)) {
21. Plane anotherPlane = planeMap.get(planeNumber);
22. if (!plane.equals(anotherPlane)) {
23. int anotherPlaneLine = planeLineMap.get(planeNumber);
24. throw new PlaneAttributeConflictException(file, planeLine, anotherPlaneLine);
25. }
26. } else {
27. planeMap.put(planeNumber, plane);
28. planeLineMap.put(planeNumber, planeLine);
29. }

在调用parse函数时客户端需要处理这些异常。由于这些错误不方便在自己的代码中处理，所以我在FlightScheduleApp中使用的处理方法是：当检测到这些异常时，弹出一个对话框提示错误的具体信息，提示用户修改文件内容或者选择其他的文件。

为了测试这些异常能否正确抛出，我自己书写了FlightSchedule\_6.txt到FlightSchedule\_23.txt这18个测试文件，用来测试每个异常能否正确抛出。

### **处理客户端操作时产生的异常**

接下来,我考虑了3.12节各种client操作时可能出现的以下“异常”操作,采用自定义checked exception和相应的异常处理机制改造我原本的代码,提高健壮性：

* 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。为此我设计了一个异常类：UnfinishedEntryUsingTheResourceException。
* 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。为此我设计了一个异常类：UnfinishedEntryMayReachThisLocationException。值得注意的是只有当计划项可能到达这个位置时才会抛出对应的异常，如果已经离开了位置就不做考虑。
* 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。这个异常在实验三中我就设计一个异常类IllegalStateTransitionException。在实现的状态转移的中如果出现非法的状态转移，那么就会抛出这个异常。当客户端捕获这个异常后，程序会显示出一个对话框来提示用户这个操作非法。
* 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”。我为这一异常设计了异常类ResourceConflictException。在每次分配资源后程序会将资源分配到对应的计划项，之后检测是否存在异常。如果存在异常，那么会取消资源分配，同时抛出异常。对应的代码如下：

1. entry.setResource(plane);
2. for (FlightPlanningEntry anotherEntry : board.getEntries()) {
3. if (entry == anotherEntry) {
4. continue;
5. }
6. if (anotherEntry.conflictResourceWith(entry)) {
7. logger.error("The entry: " + entry + "and entry: " + anotherEntry + "conflict on plane: " + plane);
8. throw new ResourceConflictException(entry, anotherEntry, plane);
9. }
10. }
11. entry.allocated();
12. logger.info("Allocate entry " + entry + " with resource " + plane);

* 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”。我设计了异常类LocationConflictException来描述这种异常。它的实现以及应用代码与ResourceConflictException类似。

除了上述的异常，代码中还存在其他可能的异常，如修改FlightEntry的位置。所以我在代码的中还抛出了其他内置的异常类，如对于使用无法修改的内容，我在修饰器中抛出了UnsupportedOperationException类。在client中对于这些异常类，我采用了一种通用的方法将捕获的异常信息提示给用户。

## **Assertion and Defensive Programming**

基于我在实验三里为各个ADT所撰写的rep invariants (RI)、AbstractionFunction (AF),以及它们的每个方法的spec(pre-condition 和 post-condition),对代码进行防御式改造。

### **checkRep()检查rep invariants**

在实验三中，我就为每个类编写了checkRep方法。按照对应的RI，我撰写了checkRep并且在可能修改对应的实例的方法（即mutater）的最后调用checkREP方法。对于绝大多数ADT，其表示不变量为其拥有的变量（除开基础类型）不为空，如State类，CommonLocation类，MyRegExp类，PairLocationsEntryImpl类，SingleLocationEntryImpl类，三个App客户端等。还有一部分类的表示不变量一直为真，即不需要调用checkRep方法，如一系列Exception类（因为自身其实没有存储变量，变量由父类存储，其对应的checkRep由父类实现）等。而对于一些特别的类，我们需要为它们设计客制化的表示不变量并编写单独的checkRep方法。下面是一些表示不变量特别的类。

LocationWithLatitudeAndLongitude类：其名称不是null，经纬度要么同时为NaN（如果暂时不知道位置的经纬度），要么维度在闭区间[-90, 90]这个范围（正负用来表示南北）、经度在闭区间[-180, 180]这个范围（正负用于表示东西）。

CommonLocation类：其名称不是null。

MultipleSortedResourceEntryImpl类：其资源列表不是null，且其存储的资源对象也不是null。

MultipleTimeslotEntryImpl类：其时间对列表不是null，且其存储的时间对也不是null。

AbstractCommonPlanningEntry<R>类：其名称非空。

Timeslot类：开始时间要早于结束时间。

MultipleLocationsEntryImpl类：其位置列表不是null，且其存储的位置也不是null。

FlightEntry类：满足父类的表示不变量，其三个接口的impl类不是null，起点和终点不同，其位置均可以共享。

Plane类：其编号id和类型type不是null，座位数seats为正数，机龄age非负。

CourseEntry类：满足父类的表示不变量，各个变量不是null，位置不可共享。

TrainEntry类：满足父类的表示不变量，其三个接口的impl类不是null，名称非空，时间对列表非空，位置列表的大小比时间对列表的大小大一。所有的位置均可以共享且各不相同，所有的车厢编号不同。

TrainCabin类：编号id为正数，类型type不是null，座位数seats也是正数。

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

每一个方法，我需要为每个方法检测前置约束，所以我在每个方法的开头调用了断言来检测传入的参数是否满足前置条件。如对于train.Factory的工厂方法createPlanningEntry，我提前检测了传入的参数是否合法。

1. public class Factory implements base.Factory<TrainCabin> {
2. public static final Factory FACTORY = new Factory();
4. /\*\*
5. \* Get an instance of {@link TrainPlanningEntry}
6. \*
7. \* @param name      the name of the plan, not empty
8. \* @param timeslots the timeslots of the plan, not null
9. \* @param locations the locations of the plan, not null
10. \* @return a {@link TrainPlanningEntry} with given params
11. \*/
12. @Override
13. public TrainPlanningEntry createPlanningEntry(String name, List<Timeslot> timeslots, List<Location> locations) {
14. assert !timeslots.isEmpty() && timeslots.size() == locations.size() - 1;
15. return TrainPlanningEntry.factoryMethod(name, timeslots, locations);
16. }
17. }

对于每一个setter，我会提前检测设置的新的值是否满足前置条件。如对于MultipleSortedResourceEntryImpl的setResources函数，我提前利用断言检测了对于resources列表。

1. @Override
2. **public** **void** setResources(List<R> resources) {
3. **assert** resources != **null**;
4. **assert** resources.parallelStream().allMatch(x -> x != **null**);
5. **this**.resources = **new** ArrayList<>(resources);
6. checkRep();
7. }

同样对于setResource、setLocations、setLocation也具有类似的断言用于检测。

而对于构造函数的前置条件的检测，可以使用在完成构造后使用checkRep来进行检测。在另一个方面，它也是对于后置条件进行检测。

对于一部分getter，我也检测了后置条件，如对于PairLocationsEntryImpl的getLocations方法，我设计了如下的后置条件检测。

1. @Override
2. **public** List<Location> getLocations() {
3. List<Location> list = Arrays.asList(startLocation, targetLocation);
4. **assert** list.size() == 2;
5. **return** list;
6. }

对于GUI等系统提供的函数，我无法在函数的内部进行检测，所以我对于函数的返回值检测。如对于JComboBox的getSelectedItem函数，我就检测了返回值是否为null。

1. CoursePlanningEntry entry = (CoursePlanningEntry) entryComboBox.getSelectedItem();
2. **assert** entry != **null**;

### **你的代码的防御式策略概述**

对于State类的每次函数调用，我都会检测State类对应的计划项是否为正确。对于一个RunningState的end方法，我就设计了下面的代码：

1. @Override
2. **public** **void** end() **throws** IllegalStateTransitionException {
3. **assert** entry.getState() == **this**;
4. entry.setState(**new** EndedState(entry));
5. }

这样就不会让客户端拿着一个已经“过时”的State类来改变“现在”的计划项。

除此之外，对于所有的具有List类等mutable类的返回值的函数，为了防止客户端通过利用修改返回值的方式来间接修改这个类，我返回时都进行了防御性复制。如MultipleSortedResourceEntryImpl的getter，我放回的就不是私有变量，而是对于私有变量的复制。

1. @Override
2. public List<R> getResources() {
3. return new ArrayList<>(resources);
4. }

对于部分的switch语句，我都枚举了所有可能的情况，所以不应该进入default语句对应的情况。为了尽早的抛出错误，我在这些switch语句中的default后面接的语句都是assert false，进而抛出错误。

1. **private** **static** **void** parseBracket(String lineString, File file, **int** line, **char** c)
2. **throws** LeftBracketNotFoundException, RightBracketNotFoundException {
3. **switch** (c) {
4. **case** '{':
5. **if** ("{".equals(lineString)) {
6. **return**;
7. } **else** {
8. **throw** **new** LeftBracketNotFoundException(file, line);
9. }
10. **case** '}':
11. **if** ("}".equals(lineString)) {
12. **return**;
13. } **else** {
14. **throw** **new** RightBracketNotFoundException(file, line);
15. }
16. **default**:
17. **assert** **false**;
18. }
19. }

## **Logging**

日志是用于可以用来记录程序的操作过程，它具有可以设置输出级别，禁止某些级别输出，如，只输出错误日志；可以被重定向到文件，这样可以在程序运行结束后查看日志；可以按包名控制日志级别，只输出某些包打的日志等功能。

在此次实验中我使用的是Log4j这个第三方包来实现日志功能。通过在pom.xml文件中添加如下代码来添加Log4j这个第三方包。

1. <dependencies>
2. <dependency>
3. <groupId>log4j</groupId>
4. <artifactId>log4j</artifactId>
5. <version>1.2.17</version>
6. </dependency>
7. </dependencies>

我设计了两个Appender来处理日志：一个是将日志输出到终端的Log4j自带的org.apache.log4j.ConsoleAppender，另一个是将日志存储下来的LogManager，用来提供后面的日志查询的功能。

### **异常处理的日志功能**

对于程序中抛出的异常，我都会打印一个ERROR级别的日志。大部分的日志是在抛出的位置打印的。如SingleLocationEntryUnmodifiableDecorator的setter方法会抛出异常UnsupportedOperationException以阻止客户端修改成员变量。现在我在抛出异常之前会打印一个日志：

1. **public** **class** SingleLocationEntryUnmodifiableDecorator **implements** SingleLocationEntry {
2. ...
3. @Override
4. **public** **void** setLocation(Location location) {
5. Logger.getRootLogger().error("Try to mutate an immutable class.");
6. **throw** **new** UnsupportedOperationException();
7. }
8. }

而剩下异常由于会由客户端捕获，所以我将日志打印的工作交给了应用类。

### **应用层操作的日志功能**

对于应用层的操作，我都将成功的操作以INFO级别的日志来记录，而对于捕获到的异常，我则是使用ERRO/FATAL级别的日志来记录。如在航班管理应用中，如果解析文件成功，那么程序会记录一个成功的INFO级别的日志；如果失败，那么会记录一个ERROR级别的日志；如果文件不存在，那么会记录一个FATAL级别的日志。

1. JFileChooser fileChooser = new JFileChooser();
2. fileChooser.setCurrentDirectory(new File(System.getProperty("user.home")));
3. int result = fileChooser.showOpenDialog(this);
4. if (result == JFileChooser.APPROVE\_OPTION) {
5. File selectedFile = fileChooser.getSelectedFile();
6. logger.info("Try to parse " + selectedFile.getAbsolutePath());
7. try {
8. board = new Board(new ArrayList<>(FlightEntry.parse(selectedFile)));
9. } catch (FileParseException exception) {
10. logger.error("Fail to parse " + selectedFile.getAbsolutePath());
11. JOptionPane.showMessageDialog(this, "文件解析失败，请选择其他文件\n" + exception.getMessage(), "错误", ERROR\_MESSAGE);
12. } catch (FileNotFoundException exception) {
13. logger.fatal("Fail to open " + selectedFile.getAbsolutePath());
14. JOptionPane.showMessageDialog(this, "无法打开文件", "错误", ERROR\_MESSAGE);
15. } finally {
16. Set<Plane> planeSet = new HashSet<>();
17. Set<Location> locationSet = new HashSet<>();
18. board.getEntries().forEach((entry) -> {
19. planeSet.add(entry.getResource());
20. locationSet.addAll(entry.getLocations());
21. });
22. planes = new ArrayList<>(planeSet);
23. locations = new ArrayList<>(locationSet);
24. logger.info("Parsed " + selectedFile.getAbsolutePath() + " successfully");
25. refreshTable();
26. }
27. }

类似的，我为添加资源、删除资源、添加位置、删除位置、更换计划项的位置、添加一个新的计划项、取消机会项、为计划项分配资源、启动计划项、结束计划项、阻塞机会项、检测冲突、查询计划项、更新信息板等函数添加了日志记录。当然不是所有的应用程序都拥有所有对对应的功能。

### **日志查询功能**

最早我使用的是一个类似守护进程的方案来将Log4j的日志消息利用管道的方式来记录所有的日志，但是由于对于多线程编程的不了解，所有编写出的程序在线程管理时存在一定的问题（为此我浪费了半周、哭）。

在日志查询的过程中，我使用了正则表达式来解析出一部分信息，所以需要设计一下日志格式。我使用的是自己设计的一个Appender类LogManager来存储所有的日志。对于它接收的日志，信息的格式是

[ Level:%p ] [ Method:%M ] [ Path:%l ] [ Message:%m ]

其中%p是日志信息的等级、%M是日志信息在哪个方法中记录、%l：输出日志事件的发生位置，相当于%c.%M(%F:%L)的组合，包括类全名、方法、文件名以及在代码中的行数、%m是输出代码中指定的具体日志信息、。

日志信息的过滤包含四个参数：开始时间、终止时间、操作和关联个计划项。为此LogManager中包含四个私有变量：startTime、endTime、method和entry。当日志信息的记录时间早于开始时间或晚于终止时间、或者与过滤参数的方法和计划项无关、那么在客户端获取消息时就不会返回这条信息。如果对应的参数没有限制，那么对应的参数就为null。具体的过滤方法为：

1. private boolean matchFilter(LogRecord log) {
2. if (startTime != null && log.getTime().compareTo(startTime) < 0) {
3. return false;
4. }
5. if (endTime != null && log.getTime().compareTo(endTime) > 0) {
6. return false;
7. }
8. if (method != null && !log.getMethod().equals(method)) {
9. return false;
10. }
11. return entry == null || log.getMessage().contains(entry);
12. }

而当客户端需要获取信息时，那么它需要调用方法getFilteredLogs。

1. **public** Object[][] getFilteredLogs() {
2. **synchronized** (LOG\_RECORDS) {
3. **return** LOG\_RECORDS.stream().filter(**this**::matchFilter).map(r -> {
4. Object time = r.getTime().format(DATE\_TIME\_FORMATTER);
5. **return** **new** Object[]{time, r.getLevel(), r.getPath(), r.getMessage()};
6. }).toArray(Object[][]::**new**);
7. }
8. }

而对于显示日志的部分，我使用了JTextArea类来显示过滤出的新对话框。其显示效果如下：



图 3-2设置过滤条件

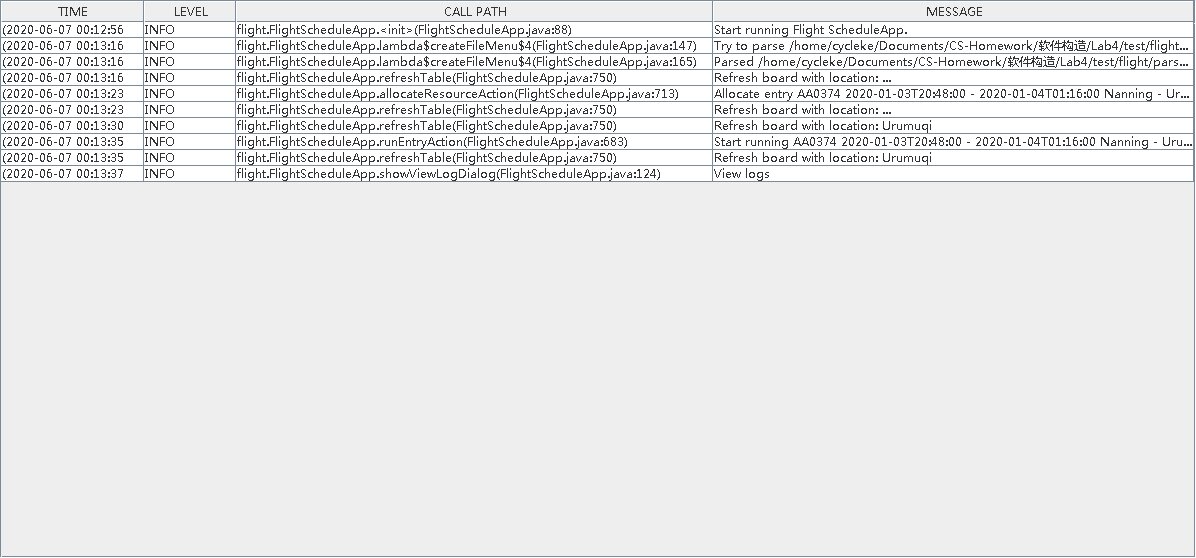


图 3-3查看日志对话框

## **Testing for Robustness and Correctness**

在本节中，我为每个类设计了对应的测试类用于测试。我使用等价类和边界值的测试思想,为各ADT设计testing strategy并以此添加了对应的测试用例。

### **Testing strategy**

对于每个方法，我都设计了对应的测试策略。对于一些简单的getters和setters函数，我的测试策略没有显式地在代码中表现出来。而对于那些情况较多的方法，我则在代码中以注释的形式将测试策略写出来。如果报告中介绍所有的测试策略，那么整个报告化显得过于冗长，所以我打算只介绍几个比较重要的具有典型特点的测试策略：

对于TrainEntry的cancel方法，我按照到达CALCELLED STATE的路径将输入划分为三个等价类：

* 由等待状态直接到达取消状态；
* 由等待状态分配完资源，到达已分配状态，之后在进入取消状态；
* 由等待状态分配完资源后到达已分配状态，之后进行运行然后阻塞，在其中一个阻塞状态中取消到达取消状态；

对于Timeslot的intersect方法，我按照两个时间对之间的相互关系对输入进行了等价类划分：

按照一个实例的起始时间的位于另一个时间对中位置的关系，我将输入划分为了5类（以左小右大的方式画出时间轴）：

* 起始时间位于另一个时间对的左方；
* 起始时间位于另一个时间对的左段点；
* 起始时间位于另一个时间对的内部；
* 起始时间位于另一个时间对的右端点；
* 起始时间位于另一个时间对的右方；

按照一个实例的终止时间的位于另一个时间对中位置的关系，我将输入划分为了5类（以左小右大的方式画出时间轴）：

* 终止时间位于另一个时间对的左方；
* 终止时间位于另一个时间对的左段点；
* 终止时间位于另一个时间对的内部；
* 终止时间位于另一个时间对的右端点；
* 终止时间位于另一个时间对的右方；

对于FlightEntry的parse方法，我按照可能抛出的异常对于输入文件进行划分：

* 文件合法，不会抛出异常；
* 文件不存在；
* 文件存在两个飞机的编号相同，而属性不相同；
* 文件中存在一个标签缺失；
* 文件中存在值的大小写错误；
* 文件中的日期格式错误；
* 文件中存在一个计划项的起飞时间晚于降落时间；
* 文件中存在一个左括号"{"缺失；
* 文件中存在一个右括号"}"缺失；
* 文件中存在一个标签，其值为空；
* 文件中存在一个飞机的位置数在规定范围外；
* 文件中存在一个飞机的机龄在规定的范围外；
* 文件中存在两个日期和编号相同的计划项；
* 文件中存在一个计划项其飞行时间超过一天；
* 文件中存在一个标签对应的内容格式不对；
* 文件中存在一个计划项的时间和编号顺序错误；
* 文件中存在一个标签的的值数目错误；

对于LocationWithLatitudeAndLongitude的构造函数，我将输入的各个参数进行了等价类划分：

* 对于参数名称name：长度为0，1和大于1三种情况
* 对于可否分享shareable：真值和假值
* 对于维度latitude：-90，-90到-1，-1，0，1，1到90，90
* 对于经度longitude：-180，-180到-1，-1，0，1，1到180，180

### **测试用例设计**

对于TrainEntry的cancel方法，为它设计了三个测试函数，即testCancel\_1，testCancel\_2和testCancel\_3。它们覆盖了三种到达CANCELLED STATE的路径，它们对应的代码如下：

1. @Test
2. **public** **void** testCancel\_1() **throws** IllegalStateTransitionException {
3. TrainPlanningEntry entry = singleton();
4. entry.cancel();
5. assertTrue(entry.getState() **instanceof** CancelledState);
6. }
7. @Test
8. **public** **void** testCancel\_2() **throws** IllegalStateTransitionException {
9. TrainPlanningEntry entry = singleton();
10. entry.setResources(Arrays.asList(**new** TrainCabin(1, TrainCabinType.BUSINESS, 10, 2020),
11. **new** TrainCabin(2, TrainCabinType.FIRST\_CLASS, 20, 2020)));
12. entry.allocated();
13. entry.cancel();
14. assertTrue(entry.getState() **instanceof** CancelledState);
15. }
17. @Test
18. **public** **void** testCancel\_3() **throws** IllegalStateTransitionException {
19. TrainPlanningEntry entry = singleton();
20. entry.setResources(Arrays.asList(**new** TrainCabin(1, TrainCabinType.BUSINESS, 10, 2020),
21. **new** TrainCabin(2, TrainCabinType.FIRST\_CLASS, 20, 2020)));
22. entry.allocated();
23. entry.run();
24. entry.block();
25. entry.cancel();
26. assertTrue(entry.getState() **instanceof** CancelledState);
27. }

对于Timeslot的intersect方法，我将所有合法的方案（即起始时间早于终止时间）组合都进行了测试。为此我设计了七个时间点，利用七个时间点的组合来进行测试。具体的代码如下：

1. @Test
2. **public** **void** testIntersect() {
3. LocalDateTime time0 = LocalDateTime.parse("2020-03-01T08:00");
4. LocalDateTime time1 = LocalDateTime.parse("2020-03-01T09:00");
5. LocalDateTime time2 = LocalDateTime.parse("2020-03-01T10:00");
6. LocalDateTime time3 = LocalDateTime.parse("2020-03-01T11:00");
7. LocalDateTime time4 = LocalDateTime.parse("2020-03-01T12:00");
8. LocalDateTime time5 = LocalDateTime.parse("2020-03-01T13:00");
9. LocalDateTime time6 = LocalDateTime.parse("2020-03-01T14:00");
10. LocalDateTime time7 = LocalDateTime.parse("2020-03-01T15:00");
12. Timeslot timeslot = Timeslot.of(time2, time5);
13. assertTrue(timeslot.intersect(timeslot));
15. Timeslot timeslot0 = Timeslot.of(time0, time1);
16. assertFalse(timeslot.intersect(timeslot0));
18. Timeslot timeslot1 = Timeslot.of(time0, time2);
19. assertFalse(timeslot.intersect(timeslot1));
21. Timeslot timeslot2 = Timeslot.of(time0, time4);
22. assertTrue(timeslot.intersect(timeslot2));
24. Timeslot timeslot3 = Timeslot.of(time0, time5);
25. assertTrue(timeslot.intersect(timeslot3));
27. Timeslot timeslot4 = Timeslot.of(time0, time7);
28. assertTrue(timeslot.intersect(timeslot4));
30. Timeslot timeslot5 = Timeslot.of(time2, time4);
31. assertTrue(timeslot.intersect(timeslot5));
33. Timeslot timeslot6 = Timeslot.of(time2, time5);
34. assertTrue(timeslot.intersect(timeslot6));
36. Timeslot timeslot7 = Timeslot.of(time2, time7);
37. assertTrue(timeslot.intersect(timeslot7));
39. Timeslot timeslot8 = Timeslot.of(time3, time4);
40. assertTrue(timeslot.intersect(timeslot8));
42. Timeslot timeslot9 = Timeslot.of(time3, time5);
43. assertTrue(timeslot.intersect(timeslot9));
45. Timeslot timeslot10 = Timeslot.of(time3, time7);
46. assertTrue(timeslot.intersect(timeslot10));
48. Timeslot timeslot11 = Timeslot.of(time5, time7);
49. assertFalse(timeslot.intersect(timeslot11));
51. Timeslot timeslot12 = Timeslot.of(time6, time7);
52. assertFalse(timeslot.intersect(timeslot12));
53. }

对于FlightEntry的parse方法，除了实验三中原本使用的五个测试文件外，我还额外设计了十八个文本文件用来测试每个异常能否正确抛出。我设计的所有测试文件中保证了每种异常至少被抛出一次。

对于LocationWithLatitudeAndLongitude的构造函数，由于四个参数之间相互独立，所以我使用了四个for迭代语句来覆盖所有可能的等价类组合：

1. @Test
2. **public** **void** testConstructor() {
3. **super**.testConstructor();
4. String[] names = **new** String[]{"", "a", "New York"};
5. **boolean**[] shareables = **new** **boolean**[]{**true**, **false**};
6. **double**[] latitudes = **new** **double**[]{-90, -45.23, -1, 0, 1, 78.90, 80, 90};
7. **double**[] longitudes = **new** **double**[]{-180, -3.2222, -1, 0, 1, 22, 34.0, 180};
8. **for** (String name : names)
9. **for** (**boolean** shareable : shareables)
10. **for** (**double** latitude : latitudes)
11. **for** (**double** longitude : longitudes)
12. assertNotNull(constructor(name, shareable, latitude, longitude));
13. }

### **测试运行结果与EclEmma覆盖度报告**

在Eclipse中测试的结果为所有的测试都成功通过，无错误和异常。

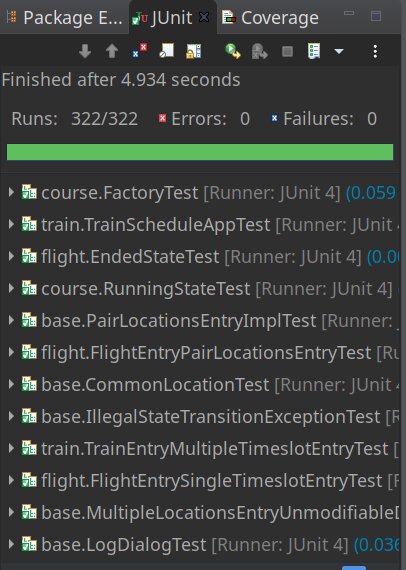


图 3-4使用JUnit测试后的结果

而对于测试的覆盖率，通过使用EclEmma插件，我的测试覆盖率结果如图3-5所示。整体代码的覆盖率达到了64.6%。测试覆盖率较低的主要原因是各种GUI相关的代码难以测试。除开测试率较低的GUI类，剩下的类测试覆盖率普遍超过了70%，部分达到了90%。

通过分析EclEmma生成的报告，我发现大部分指令起始都有执行，但是部分路径无法覆盖。其中还存在部分语句执行了，但是存在部分指令未执行（如大量使用断言检测前置条件和后置条件的语句）的情况。我认为，主要是因为断言可能会抛出异常，而我的测试输入中存在大量满足了前置条件的语句，而且在期望中后置条件的测试应当总是成立还有部分断言语句不应当执行（如switch语句中的assert false语句）。

对比三种语句覆盖度（Instruction Counters）、分支覆盖度（Branch Counters）和路径覆盖度（Complexity）这三种用与描述测试效果的指标，我发现总是语句覆盖度大于分支覆盖度大于路径覆盖率。语句覆盖度是一个最弱的覆盖率指标，因为只要语句有执行，那么它就会被统计在覆盖率中。分支覆盖度要强于语句覆盖率，它是要求对于所有的判定语句的所有结果都要求执行一边。用形象的语言描述即统计每个分叉路口有多少个路口我们的测试输入有经过。而路径覆盖度比上面两者都要更强，它是统计我们走过了多少可能的情况。可以认为路径就是分支的组合。这样就不难理解为什么总是覆盖度总是大于分支覆盖率，分支覆盖度大于路径覆盖度。

值得一提的是，就算三个覆盖度均达到100%，这仍无法证明代码是正确的。但是测试数据越强，测试覆盖度越高，那么这个程序正确的可能性越大，我们对于这个程序“越有信心”。

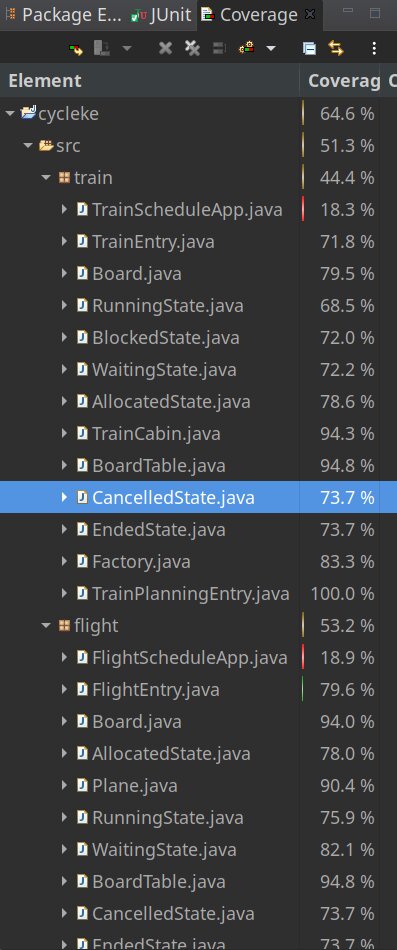


图 3-5使用EclEmma插件查看代码测试的覆盖率

## **SpotBugs tool**

在此节中我使用SpotBugs这个Eclipse插件来检测可能出现的BUG。

SpotBugs提示我的代码中有五个可能的错误，其中两个在主代码中，三个在测试代码中。

主代码中提示的错误均出现在LocationWithLatitudeAndLongitude类中，SpotBugs提示两个变量latitude和longitude可能未初始化，

SpotBugs_1

图 3-6SpotBugs对于LocationWithLatitudeAndLongitude的报错

对应的解决方法是在声明时就初始化这两个私有变量。

而测试代码中的错误是传入null值作为参数，而期望的是一个不是null的数组，对应的修改方法是将null值改为一个临时数组。

## Debugging

### EventManager程序

此程序要求我们实现EventManager类的book函数，其功能是添加一个时间，其参数为day，start和end。待添加的新事件发生在day，这是一个整数，表示一年里的第day天，start表示事件的起始时间，为该day天的第start小时，end表示该事件的结束时间，为该day天的第end小时。参数的约束条件：1<=day<=365（无需考虑闰年之类的问题），0<=start<end<=24。而此函数的返回值为在添加完该事件后在第day天的最大重叠数（依据piazza的解释）。

首先我直接使用注释中使用的实例代码进行测试，发现一个显然的错误：代码中直接使用get，没有考虑在原本的Map中可能不存在对应的键值对。而且代码只是单纯的将获取到的值又重新设置到Map中，而没有进行任何修改。所以我将代码修改的头两句语句改为：

1. temp.put(start, temp.getOrDefault(start, 0) + 1);
2. temp.put(end, temp.getOrDefault(end, 0) - 1);

这样的代码可以通过原本的测试样例。但是如果出现两天及以上的操作，那么这份代码就会出错，因为在这份代码中所有的365天都使用同一个Map，那么数据会出现冲突。由于每一天的数据是独立的，我设计了一个day到对应的Map的映射，使得每天拥有独立的Map变量。最终的代码如下：

1. **static** Map<Integer, Map<Integer, Integer>> dayMap = **new** TreeMap<>();
3. /\*\*
4. \* Calculate the max number of concurrent events in the same hour on the day given.
5. \*
6. \* @param day   the day of the event to be added, should be in [1, 365]
7. \* @param start start time of the event to be added, should be in [0, 24)
8. \* @param end   end time of the event to be added, should be in (0, 24], bigger than start
9. \* @return the max number of concurrent events in the same hour
10. \*/
11. **public** **static** **int** book(**int** day, **int** start, **int** end) {
12. **assert** day >= 1 && day <= 365;
13. **assert** start >= 0;
14. **assert** start < end;
15. **assert** end <= 24;
17. Map<Integer, Integer> hourCount;
18. **if** (dayMap.containsKey(day)) {
19. hourCount = dayMap.get(day);
20. } **else** {
21. hourCount = **new** TreeMap<>();
22. dayMap.put(day, hourCount);
23. }
25. hourCount.put(start, hourCount.getOrDefault(start, 0) + 1);
26. hourCount.put(end, hourCount.getOrDefault(end, 0) - 1);
28. **int** active = 0, ans = 0;
29. **for** (**int** d : hourCount.values()) {
30. active += d;
31. **if** (active >= ans) {
32. ans = active;
33. }
34. }
35. **return** ans;
36. }

修复之后的代码成功通过了测试代码：

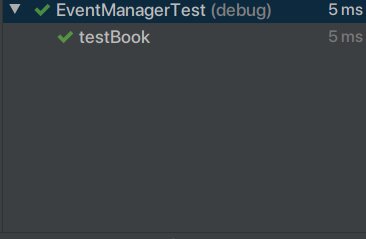


图 3-7成功通过测试

### LowestPrice程序

实验要求实现一个LowestPrice类，它的主要功能是实现找到一个最优的购买方法并返回最小划分的代价。LowestPrice类的shoppingOffers用来计算最小购买价格。其参数price是各个物品的单价，special是几种特价方案，needs是每个物品需要的数目。原本的代码使用的是深度优先搜索的算法，来通过枚举来计算出最优方案。

我通过肉眼观察代码就可以发现一些十分简单的错误：在访问needs数组时，我发现j可以到达needs.size()，这样会访问到无效下标；diff为零也是合法的，不应该放弃；特价的价格在s中的下标为needs.size()，所以计算答案时应当访问s.get(j)。在修改完这些错误后我尝试运行了一下程序，发现程序会抛出栈溢出的错误。思考代码的逻辑不难想出，当抛弃一种选择时应当break跳出循环，而不是continue来继续。这样就将代码的错误修正。

除了原本代码注释中的两个测试样例，我还构造了两个测试样例：一个需要使用多个特价方案，一个不使用任何特价方案。改正后的代码成功通过了测试。

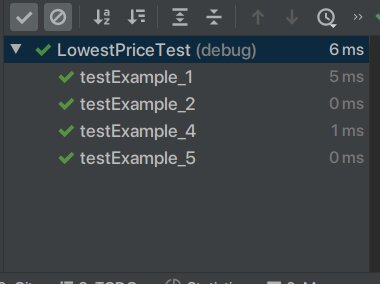


图 3-8成功通过测试

### FlightClient/Flight/Plane程序

此部分中Flight类和Plane类其实是对于多个数据的包装，实现了对应的getters和setters。此部分的重点为FlightClient，它具有一个方法planeAllocation用来检测是否存在一个分配方案使得所有航班都用对应的飞机，而不存在一架飞机会同时属于两个航班。

首先我们需要解决静态检测提示的语法错误。Flight类不是一个Comparable的类，所以无法直接直接调用sort，根据代码的随机化算法，我将所有的航班按照结束时间来排序，所以更改后的代码为为：

1. flights.sort(Comparator.comparing(Flight::getArrivalTime));

此外Calendar类无法直接使用大于/小于符号来进行比较，而需要更改为使用compareTo函数。而且原本的代码没有考虑起飞时间和降落时间相等的情况，需要添加这个考虑。

之后仔细观察代码，容易发现代码似乎可能无法终止，而且不会返回false值，所以我需要对于代码的算法进行一些修改。如果当前航班尝试所有的可能分配方案后均失败，那么我们需要返回false值。所有我在程序中添加了一个数组来记录哪些方案已经使用过，同时添加一个对应的计数器。在实现这些修改后，我在测试过程中发现航班当前的飞机可能为null，所以判断p和q是否相等时要提前判断q不为null。

我为此方法设计了两个个对应的测试样例：一个存在合法的方案，另一个不存在可行的方案。代码通过了这两个测试。



图 3-9成功通过测试

# 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。

|  |  |  |  |
| --- | --- | --- | --- |
| 日期 | 时间段 | 计划任务 | 实际完成情况 |
| 2020-5-23 | 22:00-24:00 | 完成flight.FlightEntry.parse的异常处理 | 完成 |
| 2020-5-28 | 19:00-20:30 | 完成航班管理App的异常 | 延长十分钟完成 |
| 2020-5-28 | 22:00-23:00 | 完成其他客户端的异常处理 | 完成 |
| 2020-5-29 | 13:00-14:30 | 完成3.2节 | 完成 |
| 2020-5-30 | 20:00-23:00 | 完成3.3节 | 失败，程序的BUG无法修复，需要使用其他方法 |
| 2020-6-3 | 9:00-11:45 | 完成3.3节 | 完成 |
| 2020-6-3 | 20:00-22:30 | 完成3.4节 | 完成 |
| 2020-6-6 | 20:00-20:30 | 完成3.5节 | 完成 |
| 2020-6-7 | 13:45-17:00 | 完成3.6节 | 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 |
| Logging工具的不熟悉 | 上网搜索Log4j的相关资料 |
| 日志的GUI显示 | 利用多线程和管道获取日志信息 |
| 多线程出现未查明的错误 | 更换技术方案，使用Appender来获取日志 |

# 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

在此次实验过程中，我学习了如何使用异常以及断言来增强代码的鲁棒性和正确性，同时也学会了如何使用日志方便以后代码的查错和调试。

同时我还获取的一些深刻的教训。每个函数应当有清晰的pre condition和post condition，便于添加断言和编写checkRep。即使单个ADT没有出现问题，在进行整合使用时也可能出现问题。个人的精力是有限的，使用插件等自动化工具可以有效提高DEBUG的效率。

## 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？

在保证正确性时，我如果发现错误第一时间就是报错（如抛出异常）。而如果要保证代码的健壮性，那么我就不能粗暴地报错，而需要考虑更多的处理，使得程序仍可以稳定运行。这是一种更全面的思考，要考虑应对不可预期的使用情况的措施。

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？）

显然这与代码的使用场景有关。如果代码是使用在一些对于代码的稳定性要求极高的场景，如火箭发射控制、银行账户管理等应用对于程序稳定性的要求是极高的，因为出错的代价极大。而如果程序的价值较小（如自己平时写的小程序，一些普通的应用程序）或者修改代码的代价较大（如时间成本），那么我认为可以暂时可以不处理或者简单处理这些错误或异常，而在后续的代码中可以通过补丁的方式来修正错误。

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？

二者的差异在于程序员在编写代码时对于客户端的用户差异。如果面向的是一般的用户，因为他们的对于程序的了解不够，即使直接了解到错误也无法修复，那么我会让自己的程序应对更多的异常。而如果面向的是较专业的用户，他们对于异常可以有自己的理解，那么我会让客户端/程序的用户承担确保正确性的职责。此外如果对于程序的稳健性要求较高，出错的代价较高，那么我会在程序内部处理异常和错误。

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？

没有必要。我们不能单一地注重程序的效率或者安全性，而是需要根据程序的应用场景来平衡。如果场景要求程序的性能极高，如高频交易、大规模科学计算等场景，那么我们可以适当舍弃一些过于谨慎的防御。

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？

利用系统抛出的异常信息中的栈信息可以快速定位出错的地方。在编写代码的过程中总有部分细节出错的概率较高，我在查错时会优先在这些细节附近检查。Assertion可以让错误更早地暴露，而log可以让我不用开始单步调试就可以获取一些运行时变量的信息，更快的进行分析。

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？

充分的测试意味着考虑到几乎所有的等价类和边界情况，可以应对诸如 模糊测试之类的形式化方法。

代码覆盖率100%并不意味着测试充分。覆盖率达到100%只是意味这代码所有可能的运行步骤都测试了，但是仍可能存在部分等价类没有覆盖。

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。

在Debug成功时有一定的乐趣，而失败时就没那么有趣了。

如果程序无注释，那么我需要提前分析代码，猜测代码的意图。这使得修改无注释、无文档的程序难度提升了不止一个难度级别。

1. 关于本实验的工作量、难度、deadline。

难度和deadline适中，如果平时对于防御性编程而意识较低，那么3.2，3.3的工作量较大，而且对应的报告较多。

1. 到目前为止你对《软件构造》课程的评价和建议。

希望以后实验的描述更加清晰，不要中途改需求。。。

1. 期末考试临近，你对占成绩60%的闭卷考试有什么预期？

希望可以更多的考察知识性的内容，不要考察细枝末节的单纯考察记忆的知识点。希望考试的内容可以与实验相结合，毕竟更多的同学在实验上花费的时间更多。