哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

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	张瑞豪
学号	1180800811
班号	1803002
电子邮件	m17779349381@163.com
手机号码	17779349381

目录

1 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

2 实验环境配置

URL: https://github.com/ComputerScienceHIT/Lab4-1180800811.git

3 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！）。

3.1 Error and Exception Handling

3.1.1 处理输入文本中的三类错误

处理文本中三类错误的代码结构:

```
public void ReadFileCreatePlanningEntry(String file) throws Exception
```

负责读取文件，每次读取十三行信息并且调用 getFileFlightEntry(String s)方法来完成航班信息的提取并且创建航班。

```
public FlightEntry getFileFlightEntry(String s )
```

方法，

这个方法负责解析读取的十三行信息，同时也判断是否有各种异常，比如文件信息不符合格式要求，有相同的航班号等等。具体流程是:先调用 IsillegalRegular 方法判断是否有格式错误，如果没有则读取其中的信息构造一个航班，在调用 PanDuan 方法来判断构造的航班和原来的航班是否存在依赖关系错误，并且判断是否有完全相同的航班。

```
public void IsillegalRegular(String ff )
```

这个方法判断十三行组成的字符串是否存在格式错误，

```
public void PanDuan(FlightEntry flight)
```

判断新读取的航班和已经读取的航班集是否存在依赖关系错误或者是否具有完全相同的航班信息。

3.1.1.1 文件中的格式错误

我检查了所有可能出现的错误，检查的思路是：一行一行的读取，判断每一行的格式是否均符合 lab3 给的格式要求。

思路:由 lab3 可知，去除空行后，文件中每 13 行表示一个航班计划项，每一行都要符合 lab3 给出的格式要求。我使用 13 个正则表达式进行匹配，第一个正则表达式匹配第一行，如果第一行格式错误，直接抛出异常和提示信息，提醒用户换另一个文件进行读取。如果第一行格式正确，则使用第二个正则表达式匹配第一行和第二行，如果不匹配，那么肯定是第二行格式错误，因为我们已经知道第一行格式正确。依次类推，那么就需要 13 个正则表达式进行匹配。

为啥匹配的时候不是一次匹配一行呢：因为一次匹配一行，可能会出现这样的情况:13 行的每一行的信息都是符合格式要求的，但是每一行的顺序不是正确的，这种情况下每次匹配一行也能匹配到相应的信息，但是这显然不符合格式要求。

所有的错误：

- 第一行:①格式不符合 Flight:xxx,xxx

②航班日期格式不符合 xxxx-xx-xx
③ 航班号不符合格式要求，不是由两位大写字母和 2-4 位数字构成

- 第二行：①第二行不是‘{’
- 第三行：①格式不符合 DepartureAirport:xxx,比如 DepartureAirport 拼写错误
- 第四行： ①格式不符合 ArrivalAirport:xxx,比如 DepartureAirport 拼写错误
- 第五行： ①格式不符合 DepatureTime:xxxx,比如 DepatureTime 拼写错误
  ②出发时间格式不符合 xxxx-xx-xx yy:yy
- 第六行：①格式不符合 ArrivalTime:xxx,比如 ArrivalTime 拼写错误
  ②到达时间格式不符合 xxxx-xx-xx yy:yy
- 第七行：①不符合 Plane:xxx 格式，比如 Plane 拼写错误
  ②飞机编号格式不符合要求，不符合第一位为 N 或 B，后面是四位数字
- 第八行：①第八行不是‘{’
- 第九行：①第九行格式不符合 Type:xxx，比如 Type 拼写错误
  ②机型格式错误，不符合大小写字母或数字构成，不含有空格和其他符号
- 第十行：①第十行不符合 Seats:xxx，比如 Seats 拼写错误
  ②座位数不是整数
  ③座位数不在[50,600]
- 第十一行：①第十一行格式不符合 Age:xxx，比如 Age 拼写错误
  ②机龄不符合要求，比如范围不在[0,30]，小数位数超过 1 位
- 第十二行：①第十二行不是‘{’
- 第十三行：①第十三行不是‘{’

注释：我判断文件是否合法均是按照以上的标号顺序进行判断。(详细见代码)

自定义的异常类：illegalRegularException ，FileChooseException

```
package MyException;

public class illegalRegularException extends Exception{
    /**
     * 格式不符合要求的异常
     * @param message 提示信息
     */
    public illegalRegularException(String message) {
        super(message);
    }
}
```

注释：我把所有的和格式相关的异常都统一当做 illegalRegularException 异常，即格式不合法，因为我检测的格式不合法的异常实在太多了，感觉一一列举的话重复的工作很多。我会在抛出 illegalRegularException 异常的提示信息里面会区分到底是哪一行的格式不符合。客户端可以直接捕获异常获取异常信息。

具体的异常信息的捕获：

当出现格式错误，直接抛出 illegalRegularException 异常，捕获异常之后，再抛出 FileChooseException 异常给客户端代码，客户端代码获取异常之后，根据异常的提示信息来提示给用户，并进行日志记录。

```
            if(!(matcher = pattern11.matcher(ff)).find()) {
                t = "文件第" +cloum + "行输入不合法，航班的13行数据的第11行数据不符合要求";
                throw new illegalRegularException(t);
            }
            cloum ++;
            if(!(matcher = pattern12.matcher(ff)).find()) {
                t = "文件第" +cloum + "行输入不合法，航班的13行数据的第12行数据不符合要求";
                throw new illegalRegularException(t);
            }

        }catch(illegalRegularException e) {
            throw new fileChooseException( "FileChooseException:文件格式不合法：" +t + "，请重新选择文件");
        }
    }
```

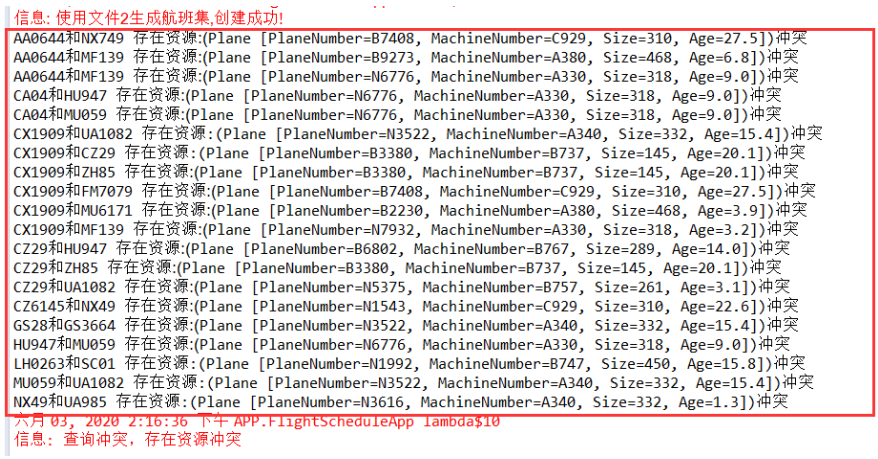
具体的异常信息

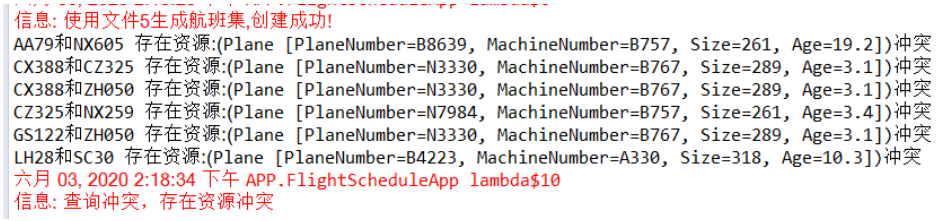
```
    try {
        schedule = new FlightSchedule("src/text/FlightSchedue1_1");
        JOptionPane.showMessageDialog(bu, "创建成功");
        logger.log(Level.INFO, "使用文件1生成航班集,创建成功!");
    } catch (Exception e1) {
        JOptionPane.showMessageDialog(bu, "创建失败,请选择其他的文件进行读取");
        logger.log(Level.SEVERE, "使用文件1生成航班集,创建失败!",e1);
        e1.printStackTrace();
    }
```

客户端读取文件创建航班出现异常则进行捕获，并提示用户重新选择文件

五个测试文件的结果：文件 1、3、4 的资源依赖关系错误，文件 2、5 存在资源冲突。

```
严重: 使用文件1生成航班集,创建失败!
MyException.fileChooseException: FileChooseException:文件不合法航班AA638 和航班 AA2784 使用相同的飞机，但是飞机的类型、座位数或机龄却不一致，请重新选择文件
```





3.1.1.2 输入文件中存在标签完全一样的元素

比如:存在多个航班计划项的“日期,航班号”信息完全一样。

思路：读取一个航班项之后，遍历之前已经读取到的航班集，判断是否存在和这个航班项完全相同的航班即可。详细见代码

自定义的异常类：sameLabelException

```
public class sameLabelException extends Exception{
    public sameLabelException() {
        super();
    };
```

```
    if(flightEntry.get(i).getResource().equals(flight.getResource()) ) {
        if(flightEntry.get(i).getBeginEndTime().equals(flight.getBeginEndTime())) {
            t = "输入文件存在标签完全一样的航班:" + flight.getName() ;
            throw new sameLabelException();
        }
    }
```

抛出异常

3.1.1.3 输入文件中各元素之间的依赖关系不正确

```
//在一个文件中不允许出现日期一样且航班号一样的两个计划项
//但允许出现“日期不同、航班号相同”的情况
//也允许出现“日期相同、航班号不同”的情况
//同一个航班号，虽然日期可以不同，但其出发和到达机场、
//出发和到达时间均应相同
//注意：CA0001、CA001、CA01 是相同的航班号
```

思路:每次读取了一个航班项之后，就遍历已经读取的航班项，判断是否存在上述依赖关系不正确。详细见代码

自定义的异常类:wrongDependenceException 类

```
public class wrongDependenceException extends Exception{
    /**
     * 错误的依赖关系的异常
     * @param message 提示信息
     */
    public wrongDependenceException(String message) {
        super(message);
    }
    public wrongDependenceException() {}
}
```

部分代码示例

```
for(int i = 0 ; i < flightEntry.size() ; i++) {
    String ss = flightEntry.get(i).getName().substring(0,2);//航班号前两个字符
    int xx = Integer.valueOf(flightEntry.get(i).getName().substring(2));//后面四个数字
    if(ss.equals(s) && xx == x) {//航班号相同
        String s11 = sdf.format(flightEntry.get(i).getTime1().getTime());//出发时间
        String s22 = sdf.format(flightEntry.get(i).getTime2().getTime());//终止时间
        if(!s11.equals(s1)) {
            t = "航班："+ flight.getName() + "的出发时间不一样!" ;
            throw new wrongDependenceException();
        }
        if(!s22.equals(s2)) {
            t = "航班："+ flight.getName() + "的到达时间不一样!" ;
            throw new wrongDependenceException();
        }
        if(!flight.getStartLocation().equals(flightEntry.get(i).getStartLocation())) {
            t = "航班："+ flight.getName() + "的出发机场不一样!" ;
            throw new wrongDependenceException(t);
        }
        if(!flight.getEndLocation().equals(flightEntry.get(i).getEndLocation())) {
            t = "航班："+ flight.getName() + "的到达机场不一样!";
            throw new wrongDependenceException();
        }
        if(flightEntry.get(i).getResource().equals(flight.getResource()) ) {
            if(flightEntry.get(i).getBeginEndTime().equals(flight.getBeginEndTime())) {
                t = "输入文件存在标签完全一样的航班:" + flight.getName() ;
                throw new sameLabelException();
            }
        }

    }else {
        if(flight.getResource().getPlaneNumber().equals(flightEntry.get(i).getResource().getPlaneNumber())) {
            if(!flight.getResource().equals(flightEntry.get(i).getResource())){
                t = "航班"+ flight.getName() + " 和航班 " + flightEntry.get(i).getName() + " 使用相同的飞机，但是飞机的类型、座位数或
                throw new wrongDependenceException();
            }
```

3.1.2 处理客户端操作时产生的异常

3.1.2.1 自定义的异常类

异常错误概要
类
cancelPlanningEntryException
changeLocationException
deleteLocationException
deleteResourceException
feipeiResourceException

以 cancelPlanningEntryException 异常为例，主要就是继承 Exception 类就行

```
package MyException;

import PlanningEntry.PlanningEntry;

public class cancelPlanningEntryException extends Exception{
    /**
     * 取消计划项异常
     * @param pl 待取消的计划项
     */
    public cancelPlanningEntryException(PlanningEntry pl) {
        super("取消计划项异常: 计划项" + pl.getName() + "当前的状态不允许取消");
    }
}
```

具体思路：客户端收到用户指令之后，然后调用相应的方法，如果出现异常，则相应的方法直接抛出相应的异常，客户端直接捕获异常之后就可以返回给用户提示信息。

```
try {
    t = schedule.FeiPeiResource(s4, time1, time2, s3);
} catch (feipeiResourceException e1) {
    Logger.log(Level.SEVERE, "给航班: "+s3+ "分配飞机 分配资源后存在资源独占冲突!",e1);
    JOptionPane.showMessageDialog(bu, "操作失败:" + e1.getMessage());
}
```

捕获异常

提示信息

3.2 Assertion and Defensive Programming

各项 Rep , RI , AF 在 lab3 都已经写的很详细了，这里不作过多的阐述，lab4 新增加的就是断言和 checkRep 方法，这里介绍一下。

3.2.1 checkRep()检查 rep invariants

不同的类的 checkRep 介绍

FlightEntry：

```
    public void checkRep() {
        assert loc != null ;//位置不为空
        assert res != null ;//资源不为空
        assert be != null ;//时间对不为空
    }
```

TrainEntry:

```
    public void checkRep() {
        assert mle != null ;//位置不为空
        assert be != null ;//时间对不为空
        assert msre != null ;//资源不为空
        Set<String> s = new HashSet<>();
        for(int i = 0 ; i < msre.getResource().size() ; i ++) {
            s.add(msre.getResource().get(i).getNumber());
        }
        assert s.size() == msre.getResource().size() ;//不存在重复的车厢编号

    }
```

CourseEntry:

```
    public void  checkRep() {
        assert loc != null ;//位置不为空
        assert be != null ;//时间对不为空
        assert res != null ;//资源不为空
    }
```

TimeSlot:

```
    public void checkRep() {
        assert date1 != null ;//起始时间不为空
        assert date2 != null ;//终止时间不为空
        assert date1.compareTo(date2) < 0 ;//起始时间早于终止时间
    }
```

MultipleLocationEntryImpl：

```
public void checkRep() {
```

```
        assert locations != null ;//位置列表不为空
        assert locations.size() != 0 ;//位置个数大于0
        Set<Location> l = new HashSet<>() ;
        for(Location loc : locations) {
            l.add(loc);
        }
        assert l.size() == locations.size() : "多个位置中不能有重复的位置" ;
    }
```

TwoLocationEntryImpl:

```
    public void checkRep() {
        assert start != null ;//起始位置不为空
        assert end != null ;//终止位置不为空
        assert !start.equals(end) : "起始位置和终止位置不能相同" ;
    }
```

MultipleSortedResourceEntryImpl:

```
    public void checkRep() {
        assert this.r != null : "资源不能为空";
        assert this.r.size() > 0 ;//资源数量大于0
    }
```

BlockableImpl:

```
    public void checkRep() {
        assert timeslots != null ; //时间对的列表不为空
        if(timeslots.size() > 1 ) {
            for(int i = 0 ; i < timeslots.size() - 1 ; i ++   ) {
assert timeslots.get(i).getdate2().compareTo(timeslots.get(i+1).getdate1())
< 0 ;//时间对递增
            }
        }
    }
```

UnBlockableImpl:

```
    public void checkRep() {
        assert timeslot != null ; //时间对不为空
        assert timeslot.getdate1().compareTo(timeslot.getdate2()) < 0 ;//时
间增序
    }
```

具体的 Board 类、application 类、APP 类的 checkRep 都是依靠基本的上述的 checkRep 来保证 RI，这里不一一阐述

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

思路：对于一些前置条件的判断，为了 fail fast ，我有的直接用断言检查前置条件是否符合，有的直接抛出异常。对于后置条件则是使用断言。由于很多的 getter 方法都是委托给其他的类来实现，所以大多数的后置条件的检查都是放在了一些基础的类里面，而基础的类的 getter 方法都是直接返回类的不变量，而类的不变量基本是由前置条件来保证，因此大多数的断言都是保证前置条件，对于一些不是直接返回类的属性的方法，需要使用断言保证后置条件。同时，对于自己实现的方法，为了保持 RI，同时用 checkRep 方法检测。

具体的详细见代码

```
public void setMachineNumber(String machineNumber) {
    assert machineNumber != null : "飞机编号不能为空";
    MachineNumber = machineNumber;
}
```

检查前置条件

```
public Timeslot(Calendar time1, Calendar time2) {
    if(time1.before(time2)) {
        date1 = time1;
        date2 = time2;
        checkRep();
    }
}
```

检测是否维持了RI

```
@Override
public Map<Timeslot,List<Location>> getTimeLocation(){
    Map<Timeslot,List<Location>> t = new HashMap<Timeslot,List<Location>>();
    for( int i = 0; i < getTimeslots().size(); i ++ ) {
        List<Location> loca = new ArrayList<Location>();
        loca.add(getLocation().get(i));
        loca.add(getLocation().get(i+1));
        t.put(getTimeslots().get(i), loca);
    }
    assert t != null;
    return t;
}
```

保持后置条件

3.2.3 你的代码的防御式策略概述

① 首先,类的属性都是使用 private 关键字来描述。这样保证了外部无法访问类的属性。

② 对于可变数据类型，当使用 getter 方法返回时，要进行防御式拷贝。典型的就是 Date 变量，还有就是返回 List 列表时候需要进行防御式拷贝

③ 同时，对于 setter 方法来说，如果参数也是可变的，同样不能直接用“=”来给属性赋值,需要创建一个新的变量,用这个新的变量来给类的属性赋值。

④ 对于客户端传进来的参数，通过断言和抛出异常来保证不变量，在 API 内，同时使用 checkRep 来检查不变量的正确性。断言也保证后置条件的正确性。

3.3 Logging

3.3.1 思路描述

① 自己声明一个日志类对象 logRecord，每一个日志类对象表示一条日志信息，日志类的属性有:异常发生的类名、方法名、时间、具体信息、日志等级、异常类型。从文件中读取的每一条日志用这个日志对象表示。

② 然后声明一个日志管理类 logKeeper，管理所有的日志信息，比如可以按照时间或者异常类型等等进行日志的查询。我们把 App 的日志信息输出到一个文件中，日志管理类 logKeeper 就可以读取文件信息创建多个日志类对象当客户端需要查询日志时，可以按照自己的格式来反馈给用户。

3.3.2 logRecord 类

功能:构造器是读取一个字符串，通过对字符串解析构造一个日志类对象。

```
public logRecord(String line) {
        String regex = "<record>\n\\<class\\>(.*?)\n\\<method\\>(.*?)\n\\<time\\>(.*?)\n\\<level\\>(.*
        Matcher matcher = Pattern.compile(regex).matcher(line);
        if(matcher.find()) {
            className = matcher.group(1);
            methodName = matcher.group(2);
            time = matcher.group(3);
            logType = matcher.group(4);
            message = matcher.group(5);
            exception = matcher.group(6); //异常类型
        }
    }
```

日志的属性

3.3.3 LogKeeper 类

功能:读取日志文件来构造多条日志的 List，并且按照时间对日志进行排序。

```
    public logKeeper(String ss) throws IOException, ParseException {
        BufferedReader in = new BufferedReader(new FileReader(new File(ss)));
        String fileline;
        StringBuffer s = new StringBuffer("") ;
        int cnt = 0 ;
        while ((fileline = in.readLine()) != null) {
            s.append(fileline + "\n") ;
            cnt++ ;
            if(cnt % 7 == 0) {
                logRecord record = new logRecord(s.toString());
                records.add(record);
                s = new StringBuffer("") ;
            }

        }
```

每次从文件读取七行信息就构造一个日志对象

LogKeeper 类实现了对日志的查询，可以根据时间范围，或者日志类型进行日志筛选。

```
/**
 * 查询在某个时间内的日志
 * @param time1 起始时间
 * @param time2 终止时间
 * @throws ParseException
 */
public void showRecordsTime(Date time1 , Date time2) throws ParseException {
    String[] columNames = {"序号" , "类名" , "方法名" ,"时间" , "日志等级" , "日志信息" , "异常类型"} ;//表格第一行信息
    List<logRecord> l = new ArrayList<logRecord>() ;//所有在指定时间内的日志
    for(int i = 0 ; i < records.size() ; i ++) {
        if(records.get(i).gettime().after(time1) && records.get(i).gettime().before(time2)) {
            l.add(records.get(i)) ;
        }
    }
    Object[][] rowData = new String[l.size()][] ;
    int i = 0 ;
    for(logRecord e : l) {
        rowData[i] = new String[] {String.valueOf(i+1) ,e.getClassName() , e.getMethodName() ,e.getTime() , e.getLogType() , e.getRea
        i ++ ;
    }
    SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
    JFrame jf =new JFrame("在时间"+ sdf.format(time1) + " - " + sdf.format(time2) + "之间的日志");
    Show.show(columNames, rowData, jf);
}
```

```
/**
 * 根据日志类型来查询日志
 * @param type 日志类型
 */
public void showRecordsType(String type) {
    String[] columNames = {"序号" , "类名" , "方法名" ,"时间" , "日志等级" , "日志信息" , "异常类型"} ;//表格第一行信息
    List<logRecord> l = new ArrayList<logRecord>() ;//所有的指定日志类型的日志
    for(int i = 0 ; i < records.size() ; i ++) {
        if(records.get(i).getLogType().equals(type)) {
            l.add(records.get(i)) ;
        }
    }
    Object[][] rowData = new String[l.size()][] ;
    int i = 0 ;
    for(logRecord e : l) {
        rowData[i] = new String[] {String.valueOf(i+1) ,e.getClassName() , e.getMethodName() ,e.getTime() , e.getLogType() , e.getReason()
        i ++ ;
    }
    JFrame jf =new JFrame("所有的日志类型为" + type + "的日志");
    Show.show(columNames, rowData, jf);
}
```

3.3.4 日志格式

记录的日志有:异常和用户操作的信息。对于异常来说，当抛出异常时，不仅提示用户相应的异常信息，同时通过日志记录相应的信息。每一条用户选择的指令都需要通过日志记录。

在 APP 类，定义一个日志对象，对日志记录的信息和格式进行设置。

```
private static Logger logger = Logger.getLogger(FlightScheduleApp.class.getName());//记录日志
private Panel cardPannel;//第二层容器
public void init() {//初始化
    FileHandler fileHandler;
    try {
        fileHandler = new FileHandler("src/text/logger");
        fileHandler.setFormatter(new Formatter() {

            @Override public String format(LogRecord record) {
                String string = "<record>\n";
                string += "<class>" + record.getSourceClassName() + "\n";
                string += "<method>" + record.getSourceMethodName() + "\n";
                string += "<time>" + new SimpleDateFormat("yyyy-MM-dd hh:mm:ss").format(new Date()) + "\n";
                string += "<level>" + record.getLevel() + "\n";
                string += "<message>" + record.getMessage() + "\n";
                string +="<Exception>" + record.getThrown() + "\n";
                return string;
            }
        });
        logger.addHandler(fileHandler);
        logger.setLevel(Level.INFO);
    }catch (Exception e) {
        e.printStackTrace();
        System.exit(-1);
    }
```

日志格式

```
<record>
<class>APP.FlightScheduleApp
<method>lambda$0
<time>2020-06-02 08:04:20
<level>INFO
<message>读入文件创建初始航班集
<Exception>null
```

日志的格式

注释：日志等级我分为三种：INFO、WARNING、SEVERE

INFO:用户的操作成功步骤

WARNING：用户操作失败的步骤，比如用户输入的时间格式错误等等。这种步骤不涉及 ADT 抛出的异常。

SEVERE：记录 ADT 内部抛出的异常，不如删除位置异常、删除资源异常、文件读取格式错误等等。

3.3.5 异常处理的日志功能

当出现异常时，需要记录异常的出现的类名、方法名、时间、提示信息、异常类、日志等级。

文件读取异常

```
try {
    schedule = new FlightSchedule("src/text/FlightScheduel_2");
    JOptionPane.showMessageDialog(bu, "创建成功");
    logger.log(Level.INFO, "使用文件2生成航班集,创建成功!");
} catch (Exception e1) {
    JOptionPane.showMessageDialog(bu, "创建失败,请选择其他的文件进行读取");
    logger.log(Level.SEVERE, "使用文件2生成航班集,创建失败!",e1);
    e1.printStackTrace();
}
```

所有的日志类型为SEVERE的日志

序号	类名	方法名	时间	日志级别	日志信息	异常类型
1	APP.FlightScheduleApp	lambda$0	2020-06-02 08:04:24	SEVERE	使用文件1生成航班集,创建失败!	MyException.fileChooseException: FileChoo...

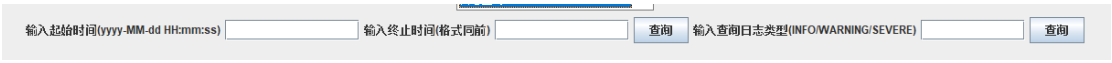
3.3.6 应用层操作的日志功能

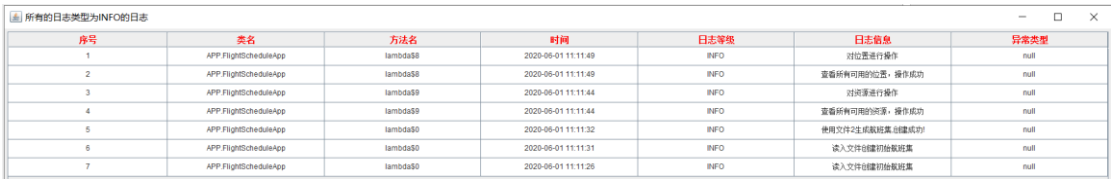
对于用户的所有操作，均记录在日志内。

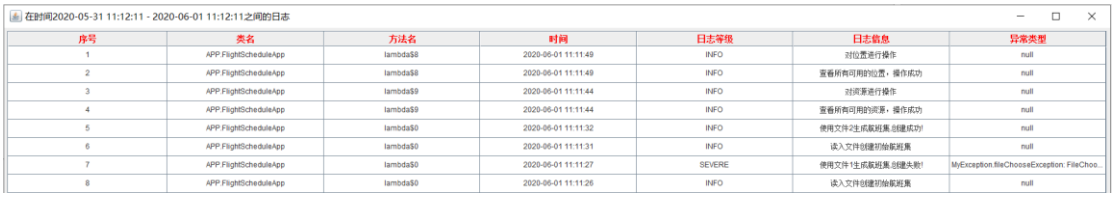
```
bu.addActionListener((e)->{//事件监听器
    logger.log(Level.INFO, "展示某个位置的信息板");
    String s1 = test1.getText();
    String s2 = test2.getText();
    String s3 = test3.getText();
    SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm");
    Calendar time = Calendar.getInstance();
    try {
        time.setTime(sdf.parse(s1));
        logger.log(Level.INFO, "查询位置: "+s2 +"的信息板,查询成功!");
        JOptionPane.showMessageDialog(bu, "操作成功");
        schedule.board(new Location(s2), time, Integer.valueOf(s3));
    } catch (ParseException e2) {
        JOptionPane.showMessageDialog(bu, "时间输入格式错误");
        logger.log(Level.SEVERE, "查询位置: "+s2 +"的信息板,时间输入格式错误!" , e2);
        e2.printStackTrace();
    }
```

3.3.7 日志查询功能

我设定了两种日志查询的功能，一个是根据时间范围进行日志查询，另一个是根据日志类型来进行日志查询。







具体的实现，在上面已经阐述。

3.4 Testing for Robustness and Correctness

3.4.1 Testing strategy

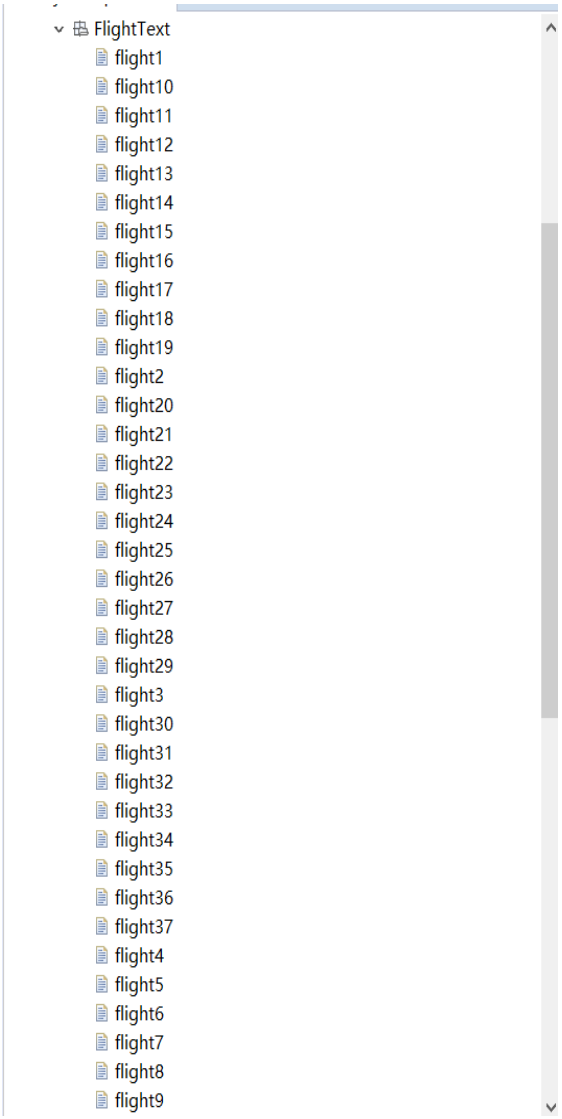
使用等价类和边界值的测试思想，为各 ADT 添加 testing strategy。

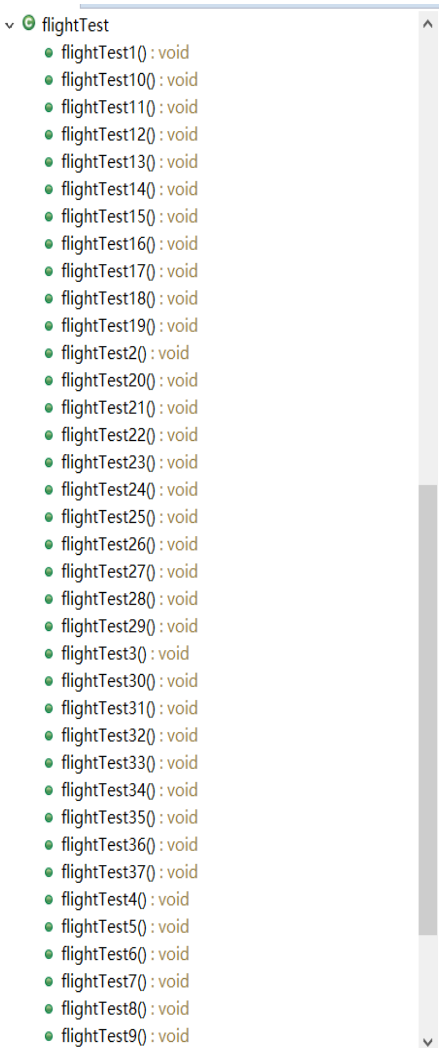
以 PlanningEntryAPIS 类为例:

```
package APIs;

import static org.junit.Assert.assertFalse;
public class PlanningEntryAPIsTest {

    /*
     * Testing strategy for checkLocationConflict()
     * Partition for PlanningEntry: CourseEntry
     * Partition for output :    there is Location Conflict between PlanningEntrys
     *                           there is not Location Conflict between PlanningEntrys
     *
     * Testing strategy for checkLocationConflict()
     * Partition for PlanningEntry: CourseEntry , FlightEntry , TrainEntry
     * Partition for output :    there is Resource Conflict between PlanningEntrys
     *                           there is not Resource Conflict between PlanningEntrys
     *
     * Testing strategy for findPreEntryPerResource(R r, PlanningEntry<R> e, List< ? extends PlanningEntry<R>> entries)
     * Partition for PlanningEntry: CourseEntry, FlightEntry , TrainEntry
     * Partition for output : PlanningEntry == null , PlanningEntry != null
     */
```

3.4.2 测试用例设计

对于所有的涉及文件读取的异常，我都专门写了测试文件来进行测试。



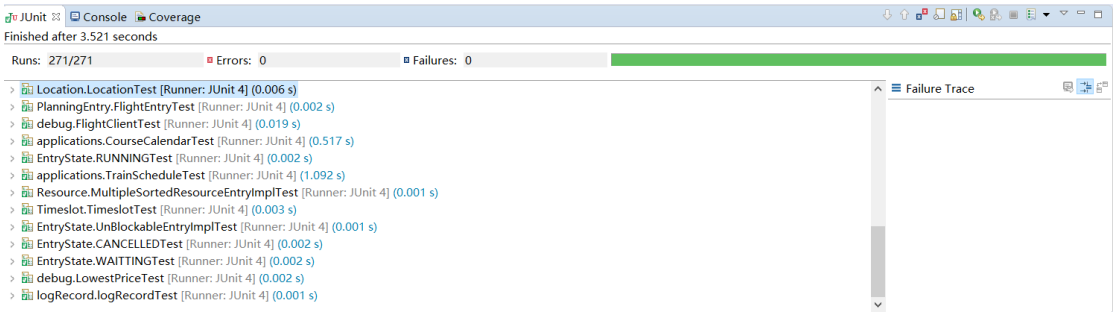


具体的每个文件的测试异常种类详细见代码

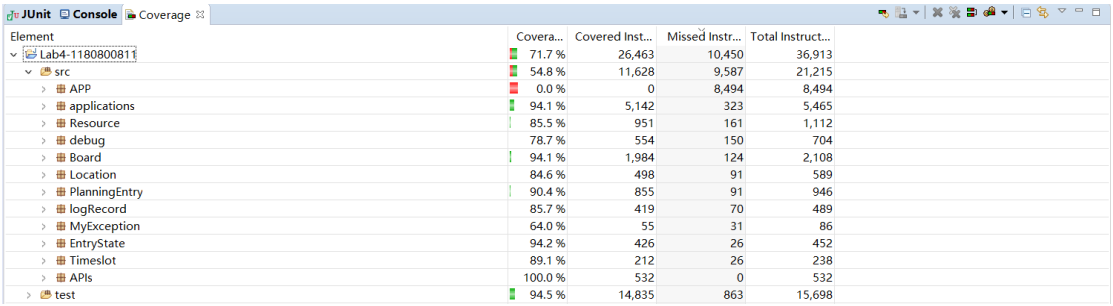
对于判断读取文件是否出现异常的四个方法，我的覆盖率为:

IsillegalRegular(String)	98.2 %	112	2	114
getFileFlightEntry(String)	97.9 %	46	1	47
PanDuan(FlightEntry)	100.0 %	37	0	37
ReadFileCreatePlanningEntry(String)	100.0 %	16	0	16

3.4.3 测试运行结果与 EclEmma 覆盖度报告

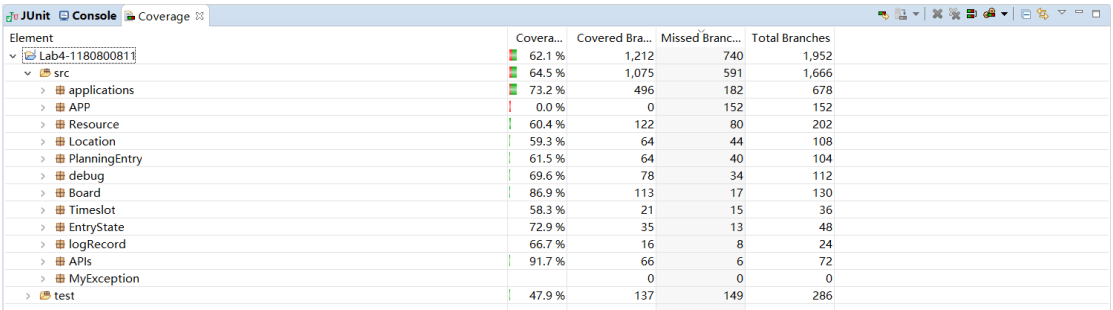


语句覆盖度:



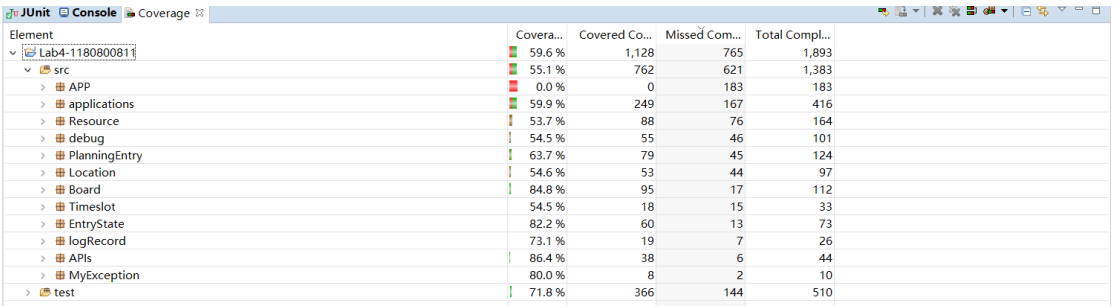
Element	Covera...	Covered Inst...	Missed Instr...	Total Instruct...
Lab4-1180800811	71.7 %	26,463	10,450	36,913
src	54.8 %	11,628	9,587	21,215
APP	0.0 %	0	8,494	8,494
applications	94.1 %	5,142	323	5,465
Resource	85.5 %	951	161	1,112
debug	78.7 %	554	150	704
Board	94.1 %	1,984	124	2,108
Location	84.6 %	498	91	589
PlanningEntry	90.4 %	855	91	946
logRecord	85.7 %	419	70	489
MyException	64.0 %	55	31	86
EntryState	94.2 %	426	26	452
Timeslot	89.1 %	212	26	238
APIs	100.0 %	532	0	532
test	94.5 %	14,835	863	15,698

分支覆盖度



Element	Covera...	Covered Bra...	Missed Branc...	Total Branches
Lab4-1180800811	62.1 %	1,212	740	1,952
src	64.5 %	1,075	591	1,666
applications	73.2 %	496	182	678
APP	0.0 %	0	152	152
Resource	60.4 %	122	80	202
Location	59.3 %	64	44	108
PlanningEntry	61.5 %	64	40	104
debug	69.6 %	78	34	112
Board	86.9 %	113	17	130
Timeslot	58.3 %	21	15	36
EntryState	72.9 %	35	13	48
logRecord	66.7 %	16	8	24
APIs	91.7 %	66	6	72
MyException		0	0	0
test	47.9 %	137	149	286

路径覆盖度



Element	Covera...	Covered Co...	Missed Com...	Total Compl...
Lab4-1180800811	59.6 %	1,128	765	1,893
src	55.1 %	762	621	1,383
APP	0.0 %	0	183	183
applications	59.9 %	249	167	416
Resource	53.7 %	88	76	164
debug	54.5 %	55	46	101
PlanningEntry	63.7 %	79	45	124
Location	54.6 %	53	44	97
Board	84.8 %	95	17	112
Timeslot	54.5 %	18	15	33
EntryState	82.2 %	60	13	73
logRecord	73.1 %	19	7	26
APIs	86.4 %	38	6	44
MyException	80.0 %	8	2	10
test	71.8 %	366	144	510

3.5 SpotBugs tool

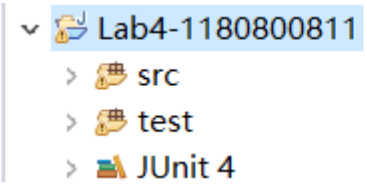
① 一开始粘贴 3.6 节老师给的 debug 代码的时候， 出现的 bug 是判断相等用的是==，而不是 equals 方法

```
5 »   »   if (this.getPlaneNo()==(ap.getPlaneNo()))¶
7 »   »   »   return true;¶
```

```
15 »   »   Plane ap = (Plane) another;¶
16 Comparison of String objects using == or != in debug.Plane.equals(Object) [Troubling(11), Normal confidence]
17 »   »   »   return true;¶
18 »   »   return false;¶
```

改为 equals 之后 bug 自然消失。

除了这个 bug 之外我没有其他的 bug



3.6 Debugging

3.6.1 EventManager 程序

功能:给定许多个事件的起止时间，需要找出某一时刻同时进行的事件的最大数量。

3.6.1.1 理解待调试程序的代码思想

属性是类型为 TreeMap 的变量 temp，temp 的键存储的是事件在时间轴上的时间，值对应的是在该时间点上面同时发生的时间数量的最大值。所以只需要循环 temp 的所有的值，找出最大的值即可。

使用 TreeMap 的好处:TreeMap 已经是按照键的大小排序，所以可以直接遍历 temp，表示从时间轴向后遍历所有的事件，而不需要对键重新进行排序。

```
»   »   for (int d : temp.values()) {¶
»   »   »   active += d;¶
»   »   »   if (active >= ans)¶
»   »   »   »   ans = active;¶
»   »   }¶
```

初始化:每次加入一个事件，就把这个事件对应的时间点的原来发生的时间数量加一。

3.6.1.2 发现并定位错误的过程

需要修改的程序有以下几点错误:

① 没有检查前置条件是否符合。

② 初始化错误，一是需要存储的键没有考虑 day 这个属性，二是需要判断待加入的事件开始和结束的时间点在 temp 的键中是否存在，如果不存在，则开始时间对应的值应该置 1，结束时间置-1。

3.6.1.3 你如何修正错误

① 增加前置条件

```
if(!(day >= 1 && day <365)) {
    throw new Exception("day的范围应该在[0,365)");
}
if(!(start >= 0 && start <24)) {
    throw new Exception("start的范围应该在[0,24)");
}
if(!(end > 0 && end <=24)) {
    throw new Exception("end的范围应该在(0,24]");
}
if(!(start < end)) {
    throw new Exception("start应该小于end");
}
```

② 初始化:键要考虑 day 这个属性。而且需要判断待加入的事件开始和结束的时间点在 temp 的键中是否存在

```
if(temp.keySet().contains(day*24 + start)) {
    temp.put(day*24 + start, temp.get(day*24 + start) + 1 );
}else {
    temp.put(day*24 + start,1 );
}
if(temp.keySet().contains(day*24 + end)) {
    temp.put(day*24 + end, temp.get(day*24 + end) - 1 );
}else {
    temp.put(day*24 + end, -1 );
}
```

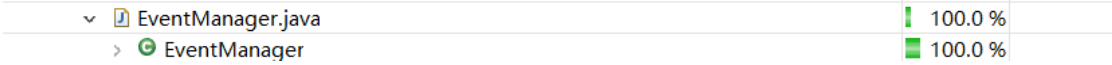
③ 后置条件

```
assert ans >= 1;//判断后置条件是否符合要求
```

3.6.1.4 修复之后的测试结果

测试策略:

```
//Partition :
// 前置条件的等价类划分: day < 1 || day > 365 || day >= 1 && day <= 365
//      start : start < 0 || start >= 24 || start >= 0 && start <= 24
//      end : end <=0 || end > 24 || end > 0 && end <= 24
//      start = end || start > end || start < end
// 前置条件正确后输入的划分：
//      k = 1, k > 1
// ---------all of the day is same, the day is not all same
//      there exists two events that have the same start/end and the day, there don't exist two events that have the same start/end and the day
```



3.6.2 LowestPrice 程序

程序功能:给定一组优惠，并且给出我们所需要的各个物品的数量，找出需要购买这些物品的最低价格。

3.6.2.1 理解待调试程序的代码思想

程序思想:首先求出按照商品的单价直接购买商品所需要的价格 res。然后，遍历所有的优惠，如果这个优惠的需要的商品的数量不超过我们需要的数量，则递归的求出使用优惠所需要的价格。最低价格就是 res 和使用优惠的最小值。

```
res = Math.min(res, s.get(j) + shopping(price, special, clone));
```

3.6.2.2 发现并定位错误的过程

错误 1:循环终止条件错误，如果优惠的需要的某种物品数量大于我们需要的这种物品的数量，则直接跳过这个优惠，选取下一个优惠，而不是继续判断另一种物品的数量是否满足要求。
错误 2:循环次数错误，List 最大下标是 size-1，而不是 size。
错误 3:判断优惠是否符合错误，diff= `clone.get(j) - s.get(j)`，`diff = 0` 也符合要求。就是优惠需要的数量和我们需要的数量恰好相等。
错误 4:没有判断前置条件是否符合

3.6.2.3 你如何修正错误

错误 1：循环终止条件错误，直接将 continue 改成 break。

```
for (j = 0; j < needs.size(); j++) {
    int diff = clone.get(j) - s.get(j);
    if (diff < 0)
        break;
    clone.set(j, diff);
}
```

将continue改成break

错误 2：循环次数错误，将终止条件 size 变成 size-1

```
for (j = 0; j < needs.size(); j++) {
    int diff = clone.get(j) - s.get(j);
    if (diff < 0)
        break ;        <= 变成 <
    clone.set(j, diff);
}
```

错误 3：判断优惠是否符合错误,将 diff <= 0 改成 diff < 0 ;

```
    int diff = clone.get(j) - s.get(j);
    if (diff < 0)
        break ;    < = 0 改成 < 0
    clone.set(j, diff);
```

错误 4：判断前置条件是否符合

```
if(price.size() > 6) {
    throw new Exception("商品数量最多为6");
}
if(special.size() > 100 ) {
    throw new Exception("优惠方案最多为100种");
}

for (int i = 0 ; i < needs.size() ; i ++) {
    if(needs.get(i) > 6) {
        throw new Exception("每件商品的购买数量最多为6");
    }
}
```

3.6.2.4 修复之后的测试结果

```
// Partition for shoppingOffers()方法
// 物品所需的数量不能通过优惠来购买， 物品的所需的数量一部分可以通过优惠购买，物品所需的数量全部可以通过优惠购买。
//
// 测试异常:* 1.最多有6种商品，特别优惠100种。
//          * 2.对于每件物品，您最多需要购买6件。
//
```

LowestPrice	100.0 %	24	0	24
dot(List<Integer>, List<Integer>)	100.0 %	4	0	4
shopping(List<Integer>, List<List<Integer>>, List<Integer>)	100.0 %	11	0	11
shoppingOffers(List<Integer>, List<List<Integer>>, List<Integer>)	100.0 %	8	0	

3.6.3 FlightClient/Flight/Plane 程序

功能:给定一组航班和一组飞机，问能否在飞机不冲突的情况下给每个航班都安排一架飞机。

3.6.3.1 理解待调试程序的代码思想

首先，对航班按照出发时间进行排序，然后随机选取一架飞机，判断如果给这个航班安排飞机是否存在冲突，如果存在冲突选取别的飞机，遍历所有的飞机如果仍然不能安排飞机，则返回 false。

3.6.3.2 发现并定位错误的过程

错误 1:对航班的出发时间排序错误。
错误 2:终止错误，没有给出当不能给某个航班分配飞机时候返回 false
错误 3:时间比较错误
错误 4:随机分配飞机错误，无法保证随机的分配飞机会覆盖到所有的飞机。也没有判断随机选择的飞机之前是否已经选择。

3.6.3.3 你如何修正错误

错误 1：对航班的出发时间排序错误。

```
Collections.sort(flights ,new Comparator<Flight>() {//对航班按照出发时间进行增序排序
    @Override
    public int compare(Flight o1, Flight o2) {
        if(o1.getDepartTime().before(o2.getDepartTime())) {
            return -1 ;
        }else if(o1.getDepartTime().after(o2.getDepartTime())) {
            return 1 ;
        }else {
            return 0 ;
        }
    }
});
```

错误 2：终止错误，没有给出当不能给某个航班分配飞机时候返回 false

```
if(bAllocated == false)
    return false ;
```

错误 3：时间比较错误

```
if ((fStart.compareTo(tStart) >= 0 && fStart.compareTo(tEnd) <= 0) || (tStart.compareTo(fStart) >= 0 && tStart.compareTo(fEnd) <= 0)) {
    bConflict = true;
    break ;
}
```

错误 4：随机分配飞机错误

```
Plane p = planes.get(r.nextInt(planes.size()));//随机分配 飞机
if(pp.contains(p)) {//判断这个飞机之前是否判断过
    continue ;
}
pp.add(p) ;

if(pp.size() == planes.size()) {//判断是否遍历了所有的飞机
    break ;
}
```

3.6.3.4 修复之后的测试结果

```
//·Partition·for·planeAllocation方法¤¶
//»  »  航班数量 > 飞机数量，  航班数量 = 飞机数量，  航班数量 < 飞机数量¤¶
//»  »  航班时间存在交叉，航班时间不存在交叉¤¶
//»  »  存在冲突，  不存在冲突¤¶
//·····测试异常:航班出发时间晚于到达时间¤¶
```

FlightClient.java	92.7 %	38	3	41
FlightClient	97.2 %	35	1	36
planeAllocation(List<Plane>, List<Flight>)	97.1 %	34	1	35

4 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
2020-05-24	20:30-11:00	完成 3.1	未完成
2020-05-24	8:00-10:00	完成 3.1	提前完成
2020-05-24	14:00-18:00	完成 3.2	未完成
2020-05-24	20:00-22:00	完成 3.2 测试类 bug	未完成
2020-05-25	全天	完成 3.4logging	完成
2020-05-26	全天	完成所有类的测试和规约撰写等等	完成
2020-05-31	21:00-23:00	完成 debug	未完成
2020-06-01	07:30-09:30	完成 debug	未完成
2020-06-01	14:00-16:30	完成 debug	完成
2020-06-02	07:40-09:40	完成实验报告	完成

5 实验过程中遇到的困难与解决途径

遇到的难点	解决途径
日志不会	上 b 站学

6 实验过程中收获的经验、教训、感想

6.1 实验过程中收获的经验和教训

6.2 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？

正确性只需要考虑保持不变量的不变，即保证 ADT 内部的正确性。

健壮性则是要考虑外部的错误对内部正确性的影响，考虑更多的是如果出现错误了怎么处理等等。

(2) 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？）

划算，我们应该考虑更多的异常，把困难留给开发人员，保证用户端的使用正确性。如果什么错误都考虑类似于在民航飞机上安装降落伞，我觉得这样没人敢坐飞机了。美国十几年前也是因为一个小小的错误导致火箭发射失败，这个代价可想而知。

(3) “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？

前者对客户的要求不高，客户的操作空间很大，考虑了客户的需求。后者对客户的限制比较多，把规范留给了客户。

(4) 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？

在我们构建代码时候，过分的防御是好的，这样可以帮助我们更快的发现错误，当代码构建完成时，将要发布给用户之后，过多的防御不必要，这样会影响性能。

(5) 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？

很明显，Assertion 和 log 会帮助我们更快的定位错误。Lab3 我遗留了一些错误，lab4 使用断言才发现。

(6) 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？

(7) 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。

一点也不快乐，没有注释没有文档，我得先猜出这个错误的程序的实现思路，然后再去改正。

(8) 关于本实验的工作量、难度、deadline。

工作量小，难度较低，deadline 合适。

(9) 到目前为止你对《软件构造》课程的评价和建议。

完美的课程，让我有了想要学软件工程的冲动。

(10)期末考试临近，你对占成绩 60%的闭卷考试有什么预期？

希望老师最后几个题目给的 ADT 不要太难，太难的话大家都没时间做完，根本没机会检验水平。要检验水平是检验不出来的，太简单的话，涉及的知识太浅，也检验不出来。中等偏上的题目就很容易检验出我们学到了多少东西，哪里没掌握等等。嘿嘿