

**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 | 郭茁宁 |
| 学号 | 1183710109 |
| 班号 | 1837101 |
| 电子邮件 | gzn00417@foxmail.com |
| 手机号码 | 13905082373 |

# 目录

[0 目录 2](#_Toc40970663)

[1 实验目标概述 1](#_Toc40970664)

[2 实验环境配置 1](#_Toc40970665)

[3 实验过程 2](#_Toc40970666)

[3.1 Error and Exception Handling 2](#_Toc40970667)

[3.1.1 处理输入文本中的三类错误 2](#_Toc40970668)

[3.1.1.1 DataPatternException 3](#_Toc40970669)

[3.1.1.2 EntryNumberFormatException 3](#_Toc40970670)

[3.1.1.3 SameAirportException 4](#_Toc40970671)

[3.1.1.4 TimeOrderException 4](#_Toc40970672)

[3.1.1.5 PlaneNumberFormatException 5](#_Toc40970673)

[3.1.1.6 PlaneTypeException 6](#_Toc40970674)

[3.1.1.7 PlaneSeatRangeException 6](#_Toc40970675)

[3.1.1.8 PlaneAgeFormatException 7](#_Toc40970676)

[3.1.1.9 SameEntryException 7](#_Toc40970677)

[3.1.1.10 HugeTimeGapException 8](#_Toc40970678)

[3.1.1.11 EntryInconsistentInfoException 8](#_Toc40970679)

[3.1.1.12 PlaneInconsistentInfoException 9](#_Toc40970680)

[3.1.1.13 SameEntrySameDayException 10](#_Toc40970681)

[3.1.2 处理客户端操作时产生的异常 10](#_Toc40970682)

[3.1.2.1 DeleteAllocatedResourceException 10](#_Toc40970683)

[3.1.2.2 DeleteOccupiedLocationException 11](#_Toc40970684)

[3.1.2.3 UnableCancelException 12](#_Toc40970685)

[3.1.2.4 ResourceSharedException 12](#_Toc40970686)

[3.1.2.5 LocationSharedException 13](#_Toc40970687)

[3.2 Assertion and Defensive Programming 14](#_Toc40970688)

[3.2.1 checkRep()检查rep invariants 14](#_Toc40970689)

[3.2.1.1 TimeSlot 14](#_Toc40970690)

[3.2.1.2 Location 15](#_Toc40970691)

[3.2.1.3 EntryState 15](#_Toc40970692)

[3.2.1.4 Resource 16](#_Toc40970693)

[3.2.1.5 PlanningEntry 17](#_Toc40970694)

[3.2.2 Assertion/异常机制来保障pre-/post-condition 17](#_Toc40970695)

[3.2.2.1 EntryState 17](#_Toc40970696)

[3.2.2.2 PlanningEntry 18](#_Toc40970697)

[3.2.2.3 PlanningEntryCollection 19](#_Toc40970698)

[3.2.3 你的代码的防御式策略概述 19](#_Toc40970699)

[3.2.3.1 Client-->API前置条件防御 19](#_Toc40970700)

[3.2.3.2 Client-->API后置条件防御 20](#_Toc40970701)

[3.2.3.3 API-->ADT前置条件防御 20](#_Toc40970702)

[3.2.3.4 API-->ADT后置条件防御 20](#_Toc40970703)

[3.3 Logging 20](#_Toc40970704)

[3.3.1 异常处理的日志功能 21](#_Toc40970705)

[3.3.1.1 需要终止当前操作的异常 21](#_Toc40970706)

[3.3.1.2 不需要终止当前操作的异常 22](#_Toc40970707)

[3.3.1.3 Assertion error 22](#_Toc40970708)

[3.3.2 应用层操作的日志功能 22](#_Toc40970709)

[3.3.3 日志查询功能 23](#_Toc40970710)

[3.4 Testing for Robustness and Correctness 24](#_Toc40970711)

[3.4.1 Testing strategy 24](#_Toc40970712)

[3.4.2 测试用例设计 25](#_Toc40970713)

[3.4.3 测试运行结果与EclEmma覆盖度报告 26](#_Toc40970714)

[3.5 SpotBugs tool 27](#_Toc40970715)

[3.6 Debugging 28](#_Toc40970716)

[3.6.1 EventManager程序 28](#_Toc40970717)

[3.6.2 LowestPrice程序 29](#_Toc40970718)

[3.6.3 FlightClient/Flight/Plane程序 30](#_Toc40970719)

[4 实验进度记录 32](#_Toc40970720)

[5 实验过程中遇到的困难与解决途径 33](#_Toc40970721)

[6 实验过程中收获的经验、教训、感想 33](#_Toc40970722)

[6.1 实验过程中收获的经验和教训 33](#_Toc40970723)

[6.2 针对以下方面的感受 33](#_Toc40970724)

# 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

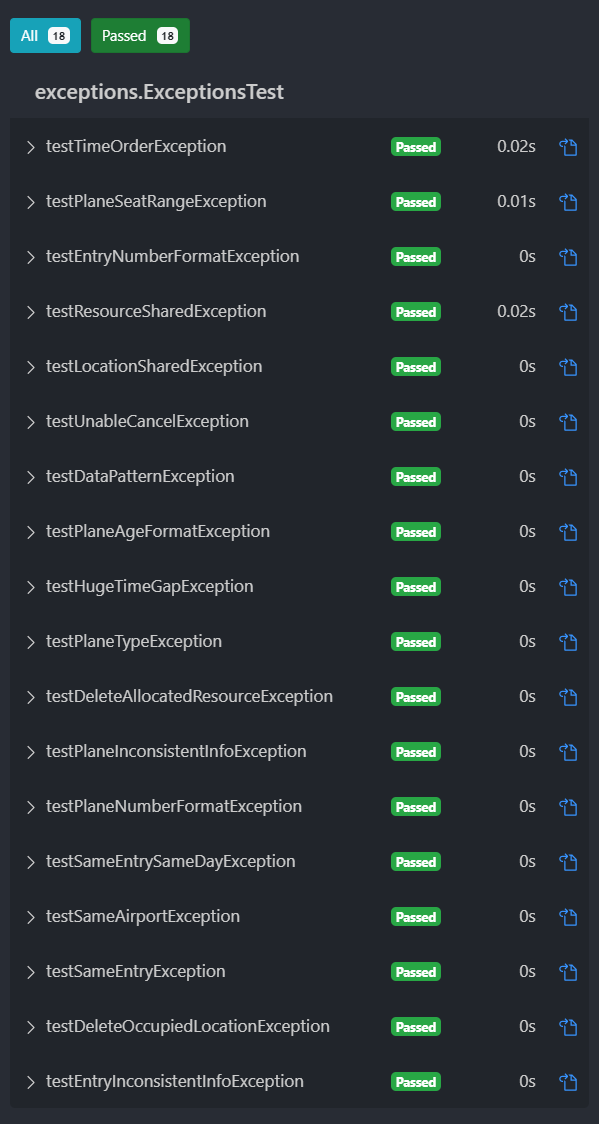
# 实验环境配置

<https://github.com/ComputerScienceHIT/Lab4-1183710109>

# 实验过程

## Error and Exception Handling

在data/Exceptions/中构造了错误数据，并在ExceptionTest.java中测试了这些错误。



### 处理输入文本中的三类错误

第1-8个为不符合语法规则错误，第9个为元素相同错误，第10-13个为依赖关系不正确错误。

处理方法为：

try {

……

throw new ……Exception();

} catch (……Exception *e1*) {

logger.log(Level.WARNING, e1.getMessage(), e1);

……

}

#### DataPatternException

原因：由于数据的常量错误而没有匹配到单个元素。

抛出异常方法：在正则表达式匹配时，若没有匹配到则抛出该错误。

if (!matcher.find()) {

throw new DataPatternException("Data: " + stringInfo + " mismatch Pattern.");

}

#### EntryNumberFormatException

原因：计划项编号不符合规则。

抛出异常方法：检查是否符合“前两个字符为大写字母，后2-4个字符为数字”。

    /\*\*

     \* check entry number

     \* @param *planningEntryNumber*

     \* @throws EntryNumberFormatException

     \*/

    public static void checkEntryNumber(String *planningEntryNumber*) throws EntryNumberFormatException {

        if (Character.isUpperCase(planningEntryNumber.charAt(0))

                && Character.isUpperCase(planningEntryNumber.charAt(1))) {

            for (int i = 2; i < planningEntryNumber.length(); i++) {

                if (!Character.isDigit(planningEntryNumber.charAt(i)))

                    throw new EntryNumberFormatException(planningEntryNumber + " has incorrect format.");

            }

        } else

            throw new EntryNumberFormatException(planningEntryNumber + " has incorrect format.");

    }

#### SameAirportException

原因：起飞和到达机场相同引起的错误。

抛出异常方法：对比两个机场字符串是否相等。

    /\*\*

     \* check airports are different

     \* @param *departureAirport*

     \* @param *arrivalAirport*

     \* @throws SameAirportException

     \*/

    public static void checkDiffAirport(String *departureAirport*, String *arrivalAirport*) throws SameAirportException {

        if (departureAirport.equals(arrivalAirport))

            throw new SameAirportException(departureAirport + " is the same with " + arrivalAirport + " .");

    }

#### TimeOrderException

原因：起飞时间应该在到达时间之前（不能相等）。

抛出异常方法：首先try时间能否被parse，若不行则抛出DateTimeParseException；否则在finally中使用LocalDateTime.isBefore()方法比较时间先后。

    /\*\*

     \* check time format and departure is before arrival

     \* @param *departureTime*

     \* @param *arrivalTime*

     \* @throws TimeOrderException

     \* @throws DateTimeParseException

     \*/

    public static void checkTime(String *departureTime*, String *arrivalTime*)

            throws TimeOrderException, DateTimeParseException {

        LocalDateTime dt = null, at = null;

        try {

            dt = LocalDateTime.parse(departureTime, DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm"));

            at = LocalDateTime.parse(arrivalTime, DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm"));

        } catch (Exception *e*) {

            throw new DateTimeParseException("The date time is not matched.", departureTime + arrivalTime, 0);

        } finally {

            if (dt != null && at != null) {

                if (!dt.isBefore(at))

                    throw new TimeOrderException(

                            "Departure time " + departureTime + " is not before arrival time " + arrivalTime + " .");

            }

        }

    }

#### PlaneNumberFormatException

原因：飞机编号不符合格式。

抛出异常方法：检查字符串长度以及首字母、后4位数字。

    /\*\*

     \* check plane number

     \* @param *planeNumber*

     \* @throws PlaneNumberFormatException

     \*/

    public static void checkPlaneNumber(String *planeNumber*) throws PlaneNumberFormatException {

        if (planeNumber.length() == 5 && (planeNumber.charAt(0) == 'N' || planeNumber.charAt(0) == 'B')) {

            for (int i = 1; i < planeNumber.length(); i++) {

                if (!Character.isDigit(planeNumber.charAt(i)))

                    throw new PlaneNumberFormatException(planeNumber + " has incorrect format.");

            }

        } else

            throw new PlaneNumberFormatException(planeNumber + " has incorrect format.");

    }

#### PlaneTypeException

原因：飞机类型不符合格式。

抛出异常方法：检查是否由字母和数字构成。

    /\*\*

     \* check plane type

     \* @param *strType*

     \* @throws PlaneTypeException

     \*/

    public static void checkPlaneType(String *strType*) throws PlaneTypeException {

        for (int i = 0; i < strType.length(); i++) {

            char ch = strType.charAt(i);

            if (!(Character.isAlphabetic(ch) || Character.isDigit(ch)))

                throw new PlaneTypeException(strType + " has incorrect format.");

        }

    }

#### PlaneSeatRangeException

原因：飞机座位数范围错误。

抛出异常方法：转换为整数比较范围。

    /\*\*

     \* check plane seat range

     \* @param *strSeats*

     \* @throws PlaneSeatRangeException

     \*/

    public static void checkPlaneSeat(String *strSeats*) throws PlaneSeatRangeException {

        int intSeats = Integer.valueOf(strSeats);

        if (intSeats < 50 || intSeats > 600)

            throw new PlaneSeatRangeException(intSeats + " is not in [50, 600].");

    }

#### PlaneAgeFormatException

原因：飞机年龄非一位小数或整数，且介于0-30之间

抛出异常方法：查找小数点的位置，与字符串长度比较，得出几位小数，并查找区间。

    /\*\*

     \* check plane age format

     \* @param *strAge*

     \* @throws PlaneAgeFormatException

     \*/

    public static void checkPlaneAge(String *strAge*) throws PlaneAgeFormatException {

        double age = Double.valueOf(strAge);

        if (strAge.indexOf(".") < strAge.length() - 2 || age < 0 || age > 30)

            throw new PlaneAgeFormatException();

    }

#### SameEntryException

原因：存在两个航班，飞机和航班号都相等。

抛出异常方法：遍历所有计划项，两两比较是否存在上述条件。

    /\*\*

     \* check dates and numbers conflict

     \* @throws SameEntryException

     \*/

    public void checkDateNumberConflict() throws SameEntryException {

        List<PlanningEntry<Resource>> entries = this.getAllPlanningEntries();

        int n = entries.size();

        for (int i = 0; i < n - 1; i++) {

            for (int j = i + 1; j < n; j++) {

                if (i != j) {

                    PlanningEntry<Resource> e1 = entries.get(i), e2 = entries.get(j);

                    if (e1.getPlanningEntryNumber().equals(e2.getPlanningEntryNumber())) {

                        if (((FlightSchedule<Resource>) e1).getResource()

                                .equals(((FlightSchedule<Resource>) e2).getResource()))

                            throw new SameEntryException(e1.getPlanningEntryNumber() + " and "

                                    + e2.getPlanningEntryNumber() + " are the same entries.");

                    }

                }

            }

        }

    }

#### HugeTimeGapException

原因：起飞时间和到达时间超过一天。

抛出异常方法：判断每个计划项的起飞时间晚1d是否比到达时间晚。

    /\*\*

     \* check gap between leaving and arrival

     \* @throws HugeTimeGapException

     \*/

    public void checkTimeGap() throws HugeTimeGapException {

        List<PlanningEntry<Resource>> entries = this.getAllPlanningEntries();

        int n = entries.size();

        for (int i = 0; i < n - 1; i++) {

            FlightSchedule<Resource> e = (FlightSchedule<Resource>) entries.get(i);

            LocalDateTime t1 = e.getTimeLeaving(), t2 = e.getTimeArrival();

            if (t1.plusDays(1).isBefore(t2))

                throw new HugeTimeGapException(t1.toString() + " is to early than " + t2.toString());

        }

    }

#### EntryInconsistentInfoException

原因：相同航班号的航班信息（起降地点/时间）不一致。

抛出异常方法：检查每一对计划项，得到其时间和地点对象。

    /\*\*

     \* check entry information consistent

     \* @throws EntryInconsistentInfoException

     \*/

    public void checkEntryConsistentInfo() throws EntryInconsistentInfoException {

        List<PlanningEntry<Resource>> entries = this.getAllPlanningEntries();

        int n = entries.size();

        for (int i = 0; i < n - 1; i++) {

            for (int j = i + 1; j < n; j++) {

                if (i != j) {

                    FlightSchedule<Resource> e1 = (FlightSchedule<Resource>) entries.get(i),

                            e2 = (FlightSchedule<Resource>) entries.get(j);

                    if (e1.getPlanningEntryNumber().equals(e2.getPlanningEntryNumber())) {

                        LocalTime t11 = e1.getTimeLeaving().toLocalTime(), t12 = e1.getTimeArrival().toLocalTime(),

                                t21 = e2.getTimeLeaving().toLocalTime(), t22 = e2.getTimeArrival().toLocalTime();

                        if (!(t11.equals(t21) && t12.equals(t22)) || !e1.getLocation().equals(e2.getLocation()))

                            throw new EntryInconsistentInfoException(e1.getPlanningEntryNumber() + " and " e2.getPlanningEntryNumber() + " is inconsistent.");

                    }

                }

            }

        }

    }

#### PlaneInconsistentInfoException

原因：不同的航班中出现相同的飞机。

抛出异常方法：遍历每一对飞机，若飞机号相同，但内容不相同，则出现不一致信息。

    /\*\*

     \* check plane information consistent

     \* @throws PlaneInconsistentInfoException

     \*/

    public void checkPlaneConsistentInfo() throws PlaneInconsistentInfoException {

        Set<Resource> planes = this.getAllResource();

        for (Resource r1 : planes) {

            for (Resource r2 : planes) {

                if (r1 != r2) {

                    Plane p1 = (Plane) r1, p2 = (Plane) r2;

                    if (p1.getNumber().equals(p2.getNumber()) && !p1.equals(p2))

                        throw new PlaneInconsistentInfoException(p1.getNumber() + " has inconsistent information.");

                }

            }

        }

    }

#### SameEntrySameDayException

原因：相同航班号的航班在同一天。

抛出异常方法：遍历比较

    /\*\*

     \* check same entry in different days

     \* @throws SameEntrySameDayException

     \*/

    public void checkSameEntryDiffDay() throws SameEntrySameDayException {

        List<PlanningEntry<Resource>> entries = this.getAllPlanningEntries();

        int n = entries.size();

        for (int i = 0; i < n - 1; i++) {

            for (int j = i + 1; j < n; j++) {

                if (i != j) {

                    PlanningEntry<Resource> e1 = entries.get(i), e2 = entries.get(j);

                    if (e1.getPlanningEntryNumber().equals(e2.getPlanningEntryNumber())) {

                        if (((CommonPlanningEntry<Resource>) e1).getPlanningDate()

                                .isEqual(((CommonPlanningEntry<Resource>) e2).getPlanningDate()))

                            throw new SameEntrySameDayException();

                    }

                }

            }

        }

    }

### 处理客户端操作时产生的异常

在App中遇到客户端操作异常时，抛出异常后使用Logger记录，并取消该操作。

#### DeleteAllocatedResourceException

原因：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。

抛出异常方法：遍历计划项，对于多个使用该资源的计划项，均检查计划项状态。捕获到异常后将“允许删除标签”设为false，最后显示弹窗声明删除失败。

Resource deletingResource = allResourceList.get(num);

boolean flag = true;

try {

checkResourceAllocated(flightScheduleCollection, deletingResource);

} catch (DeleteAllocatedResourceException *e1*) {

logger.log(Level.WARNING, e1.getMessage(), e1);

flag = false;

}

flag &= flightScheduleCollection.deleteResource(deletingResource);

JOptionPane.showMessageDialog(resourceFrame, flag ? "Successful" : "Failed", "Deleting Resource", JOptionPane.PLAIN\_MESSAGE);

#### DeleteOccupiedLocationException

原因：在删除某位置的时候，如果有尚未结束的计划项正在该位置执行。

抛出异常方法：遍历计划项，对于多个使用该位置的计划项，均检查计划项状态。

（与上DeleteAllocatedResourceException同理）

Check方法Spec如下：

    /\*\*

     \* check location occupied

     \* @param *flightScheduleCollection0*

     \* @param *location*

     \* @throws DeleteOccupiedLocationException

     \*/

    public static void checkLocationOccupied(FlightScheduleCollection *flightScheduleCollection0*, String *location*)

            throws DeleteOccupiedLocationException {

        List<PlanningEntry<Resource>> planningEntries = flightScheduleCollection0.getAllPlanningEntries();

        for (PlanningEntry<Resource> planningEntry : planningEntries) {

            FlightSchedule<Resource> flightSchedule = (FlightSchedule<Resource>) planningEntry;

            if (flightSchedule.getLocationOrigin().equals(location)

                    || flightSchedule.getLocationTerminal().equals(location))

                if (planningEntry.getState().getState().equals(EntryStateEnum.ALLOCATED)

                        || planningEntry.getState().getState().equals(EntryStateEnum.BLOCKED)

                        || planningEntry.getState().getState().equals(EntryStateEnum.RUNNING))

                    throw new DeleteOccupiedLocationException(location + " is occupied");

        }

    }

#### UnableCancelException

原因：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。

抛出异常方法：通过cancelPlanningEntry()🡪setNewState()返回的Boolean来判断是否可取消。

operationFlag = flightScheduleCollection.cancelPlanningEntry(planningEntryNumber);

if (!operationFlag)

try {

throw new UnableCancelException();

} catch (UnableCancelException *e1*) {

logger.log(Level.WARNING, e1.getMessage(), e1);

}

#### ResourceSharedException

原因：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”。

抛出异常方法：与DeleteAllocatedResourceException类似，与其不同的是在分配资源是遍历查找。

boolean flag = true;

try {

checkResourceShared(flightScheduleCollection, flightScheduleCollection.getPlaneOfNumber(strResourceNumber));

} catch (ResourceSharedException *e1*) {

logger.log(Level.WARNING, e1.getMessage(), e1);

flag = false;

}

if (flag) flightScheduleCollection.allocateResource(strPlanningEntryNumber, strResourceNumber);

JOptionPane.showMessageDialog(allocateResourceFrame, flag ? "Successfully" : "Failed", "Allocate Resource", JOptionPane.PLAIN\_MESSAGE);

#### LocationSharedException

原因：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”。

抛出异常方法：与ResourceSharedException同理。该功能在Activity Calendar App中。

    /\*\*

     \* check location modifiable

     \* @param *flightScheduleCollection0*

     \* @param *location*

     \* @throws LocationSharedException

     \*/

    public static void checkLocationModifiable(FlightScheduleCollection *flightScheduleCollection0*, String *location*)

            throws LocationSharedException {

        List<PlanningEntry<Resource>> planningEntries = flightScheduleCollection0.getAllPlanningEntries();

        for (PlanningEntry<Resource> planningEntry : planningEntries) {

            ActivityCalendar<Resource> activityCalendar = (ActivityCalendar<Resource>) planningEntry;

            if (activityCalendar.getLocation() != null && activityCalendar.getLocation().equals(location))

                throw new LocationSharedException(location + " is shared.");

        }

    }

## Assertion and Defensive Programming

### checkRep()检查rep invariants

#### TimeSlot

TimeSlot的AF、RI如下：

    /\*

     \* AF:

     \* arrival[i] represent the time it arrives locations[i]

     \* leaving[i] represent the time it leaves locations[i]

     \*

     \* when Flight Schedule:

     \* length == 2, arrival[0] == leaving[0], arrival[1] == leaving[1]

     \*

     \* when Activity Schedule:

     \* length == 1, arrival[0] is ending time, leaving[0] is beginning time

     \*

     \* RI:

     \* the length of arrival and leaving should be equal

     \* leaving[i] should be later than arrival[i]

     \* when i<length arrival[i] and leaving[i] should be non-null

     \*

     \* Safety:

     \* do not provide mutator

     \*/

由此可以设计checkRep()方法：

    /\*\*

     \* check Rep

     \*/

    private void checkRep() {

        assert (arrival.size() == leaving.size());

        for (int i = 0; i < arrival.size(); i++) {

            assert (arrival.get(i) != null);

            assert (leaving.get(i) != null);

        }

    }

#### Location

Location的AF、RI如下：

    /\*

     \* AF:

     \* locations represent the locations in the plan

     \*

     \* RI:

     \* locations should be as long as arrival and leaving in class TimeSlot

     \*

     \* Safety:

     \* do not provide mutator

     \*/

Location的Representation可以保证包括航班和高铁在内的“任意两个站不相同”。该checkRep()如下：

    /\*\*

     \* check Rep

     \*/

    private void checkRep() {

        for (String strLocation1 : locations) {

            assert (strLocation1.length() > 0);

            for (String strLocation2 : locations) {

                if (strLocation1 != strLocation2)

                    assert (!strLocation1.equals(strLocation2));

            }

        }

    }

#### EntryState

EntryState的AF、RI如下：

    /\*

     \* AF:

     \* the state enum's name represents the state

     \* RI:

     \* state must be in enums

     \* Safety:

     \* it's a mutable object, but do not let the outside modify state directly

     \*/

该checkRep()非常容易，略。

#### Resource

Resource的3个实现类均是immutable类型ADT，存储一定信息，因此其checkRep就是保证信息存储的变量符合格式，检查方法与抛出异常方法类似，因此对抛出异常的方法进行复用。以Plane为例：

    /\*\*

     \* check Rep

     \*/

    private void checkRep() {

        try {

            FlightScheduleCollection.checkPlaneNumber(number);

        } catch (PlaneNumberFormatException *e*) {

            assert false;

        }

        try {

            FlightScheduleCollection.checkPlaneType(strType);

        } catch (PlaneTypeException *e*) {

            assert false;

        }

        try {

            FlightScheduleCollection.checkPlaneSeat(String.valueOf(intSeats));

        } catch (PlaneSeatRangeException *e*) {

            assert false;

        }

        try {

            FlightScheduleCollection.checkPlaneAge(Double.toString(age));

        } catch (PlaneAgeFormatException *e*) {

            assert false;

        }

    }

#### PlanningEntry

在新建计划项时，资源、位置、时间、状态均被检查过，因此只要检查4者不为空，且标签正确即可。

    private void checkRep() {

        assert (strPlanningEntryType.equals("FlightSchedule"));

        assert (location != null);

        assert (timeSlot != null);

        assert (state != null);

        assert (resource != null);

    }

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

Assertion主要针对mutable对象的mutator。

#### EntryState

在修改状态时，前置条件和后置条件均为：当前状态合法。除了类型为高铁，否则不能为blocked。因此判断两次状态的合法性。

    /\*\*

     \* set the new state

     \* @param *strPlanningEntryType* in {"FlightSchedule", "TrainSchedule", "ActivityCalendar"}

     \* @param *strNewState*

     \* @return true if the setting is successful, false if not

     \*/

    public Boolean setNewState(String *strPlanningEntryType*, String *strNewState*) {

        assert (strPlanningEntryType.toLowerCase().contains("train")

                || !this.getStrState().toLowerCase().equals("blocked"));

        if (this.setAvailability(strPlanningEntryType, strNewState.toUpperCase())){

            this.state = EntryStateEnum.valueOf(strNewState.toUpperCase());

            assert (strPlanningEntryType.toLowerCase().contains("train")

                    || !this.getStrState().toLowerCase().equals("blocked"));

            return true;

        }

        return false;

    }

#### PlanningEntry

计划项的mutator在于分配资源和更改状态。

分配资源时，前置条件为：被分配的资源不能为空。以ActivityCalendar为例：

    /\*\*

     \* allocate the resource to the flight schedule

     \* set the state as ALLOCATED

     \* @param *resource*

     \* @param *intResourceNumber*

     \* @return true if the resource is set and state is ALLOCATED

     \*/

    public Boolean allocateResource(R *resource*, int *intResourceNumber*) {

        assert (resource != null && intResourceNumber > 0);

        super.resource = resource;

        this.intResourceNumber = intResourceNumber;

        return this.state.setNewState(strPlanningEntryType, "Allocated");

    }

更改状态时，后置条件为：更改后的状态不能为空且为某一合法状态。以CommonPlanningEntry.start()为例：

    @Override

    public Boolean start() {

        boolean flag = this.state.setNewState(strPlanningEntryType, "Running");

        assert (this.state != null && this.state.getState() != null);

        return flag;

    }

其中，PlanningEntry中的TrainSchedule有操作“取第i个车厢”，对于该i的前置条件为：不能查询第1个站的到达时间且不能查询最后一个站的出发时间。以查询出发时间为例：

    /\*\*

     \* get the LocalDateTime of leaving time of No.indexLocation Location

     \* @param *indexLocation*

     \* @return the LocalDateTime of leaving time of No.indexLocation Location

     \*/

    public LocalDateTime getLeavingTimeOfIndex(int *indexLocation*) {

        assert (indexLocation != TERMINAL);

        return super.getTimeSlot().getLeaving().get(indexLocation);

    }

#### PlanningEntryCollection

在计划项集合类中，有许多关联到计划项编号的操作，前置条件要求计划项编号参数不能为blank。同理，所有有关查询操作的参数均不能为空白。

    /\*\*

     \* search for a planning entry whose number matches the given

     \* @param *planningEntryNumber*

     \* @return the planning entry

     \*/

    public PlanningEntry<Resource> getPlanningEntryByStrNumber(String *planningEntryNumber*) {

        assert (!planningEntryNumber.isBlank());

        for (PlanningEntry<Resource> planningEntry : planningEntries)

            if (planningEntry.getPlanningEntryNumber().equals(planningEntryNumber))

                return planningEntry;

        return null;

    }

### 你的代码的防御式策略概述

代码的“错误传递”发生在客户端到API、API到ADT之间，因此在这两种传递过程的起始和完成阶段，都应该进行防御。



#### Client-->API前置条件防御

客户端和API之间，需要基于用户输入参数进行功能控制，因此用户输入的内容正确性决定了API功能实现的正确性。客户端的输入方法或API的方法起始阶段需要对用户输入进行检查。

例如FlightScheduleCollection.addPlanningEntry()中需要读入一段数据，在方法中进行了对各项参数的检查，错误则抛出包括EntryNumberFormatException在内的相应异常；而在查询指定计划项信息时，则是在FlightScheduleApp中先对该编号正确性进行检查（该操作委派给了FlightScheduleCollection）然后才获取指定信息。

#### Client-->API后置条件防御

在API操作完成之后，在客户端或API中需要对结果进行正确性的大致检查，避免一下明显错误情况；若API操作不当，可能在程序中引入隐式错误。

例如在启动计划项时，FlightScheduleApp在完成操作后弹窗显示操作结果；在暂停计划项之后，会检查该计划项类型是否为可暂停的计划项对象类型。

#### API-->ADT前置条件防御

在API的操作会对ADT进行影响，若ADT为可变的，则要求Setter()参数正确。检查参数正确可以在API的方法中，也可以在ADT的方法中。

例如API在获得某计划项的资源时，会判断该ADT的资源是否为空；在API需要获得高铁的第i站的到达时间，在ADT的方法中会对i的取值进行断言（不能为0）。

#### API-->ADT后置条件防御

在修改ADT的内容之后，需要确认修改后的ADT符合RI。此时，可以调用ADT私有方法checkRep()进行校验。在各个ADT中均有checkRep()，出现在构造器（immutable对象），也会出现在mutator（mutable对象）。

## Logging

本实验中，日志功能的实现调用了 Java 的库 java.util.logging。并且日志功能仅支持在应用中实现，不支持在测试时使用。

private final static Logger logger = Logger.getLogger("Flight Schedule Log");

在后续的日志创建中，如果 Logger.getLogger(String name)的 name 参数和上述三个之一相同，那么日志变量将指向同一块内存区域，即共享同一 logger。然后对日志配置。

首先配置日志的输出语言为英文。

Locale.setDefault(new Locale("en", "EN"));

然后设置日志显示信息的最低级别（即包括了 INFO, WARNING 和 SEVERE）。

logger.setLevel(Level.INFO);

接着对三个 logger 配置加入相应的文件管理，使得日志可以写入到文件中。

FileHandler fileHandler = new FileHandler("log/FlightScheduleLog.txt", true);

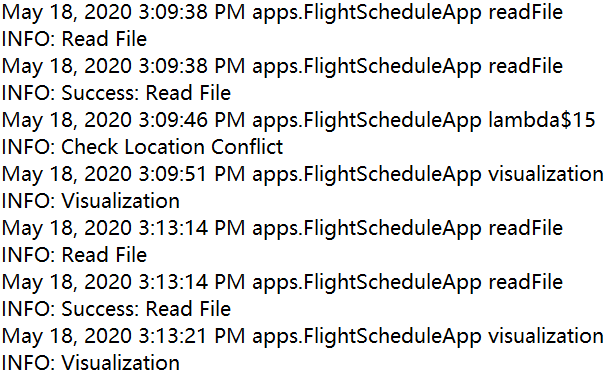
对文件管理handler配置好文件写入的格式，采用SimpleFormatter格式文件，可读性高，但在检索时可能比较麻烦。

fileHandler.setFormatter(new SimpleFormatter());

logger.addHandler(fileHandler);

这样日志的配置就做好了。在应用类中，只需要加一个全局静态变量，调用已经创建的相关的 logger，再设置无需从控制台输出日志内容即可。除了应用类，凡是应用类调用的函数如果有异常被 catch 的话，其所在的类中，也应该有这个 logger 管理变量。

文件整体效果：



### 异常处理的日志功能

#### 需要终止当前操作的异常

如果遇到需要终止当前操作的异常，在 catch 结束前，应该记录 SEVERE级别的日志信息，如：

try {

flightSchedule = flightScheduleCollection.addPlanningEntry(stringInfo.toString());

} catch (DataPatternException *e*) {

logger.log(Level.SEVERE, e.getMessage(), e);

break;

}

#### 不需要终止当前操作的异常

如果遇到不需要终止当前操作的异常，在 catch 结束前，应该记录WARNING 级别的日志信息，如：

try {

checkResourceShared(flightScheduleCollection, flightScheduleCollection.getPlaneOfNumber(strResourceNumber));

} catch (ResourceSharedException *e1*) {

logger.log(Level.WARNING, e1.getMessage(), e1);

flag = false;

}

#### Assertion error

对于应用中遇到的 Assertion error，应该记录下 SEVERE 级别的信息。

### 应用层操作的日志功能

应用中使用功能在应用中使用的任何功能，都应该在调用之后马上生成 INFO 调用信息，在功能成功结束后，生成 INFO 成功信息。如：

switch (strOperation) {

case "Start":

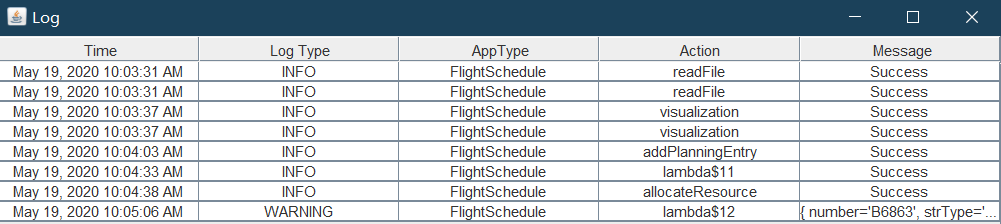
logger.log(Level.INFO, "Start Planning Entry");

operationFlag = flightScheduleCollection.startPlanningEntry(planningEntryNumber);

break;

### 日志查询功能

首先处理字符串，并能够将所有的日志显示出来。通过委派复用Board功能，可视化JTable。以下，将增加方法参数（时间、事件类型和App类型），对日志项进行筛选，以下图形式进行显示。



通过正则表达式读入日志。

Pattern pattern1 = Pattern.compile("(.\*?) apps\\.(.\*?)App (.\*?)\\.");

Pattern pattern2 = Pattern.compile("([A-Z]\*?): (.\*?)\\.+");

匹配后得出所需信息。

String time = matcher1.group(1);

String logType1 = matcher2.group(1);

String appType1 = matcher1.group(2);

String action = matcher1.group(3);

String message = matcher2.group(2);

转换时间后，检测时间是否匹配（设定：在WITHIN\_MINUTE分钟以内）：

if (!askedTime.isBlank()) {

askingTime = LocalDateTime.parse(askedTime, DateTimeFormatter.ofPattern("yyyy-MM-dd hh:mm"));

}

LocalDateTime timeFormatted = LocalDateTime.parse(timeString,

DateTimeFormatter.ofPattern("MMM dd, yyyy HH:mm:ss a"));

if (askingTime == null ||

(askingTime.plusMinutes(WITHIN\_MINUTE).isAfter(timeFormatted)

&& askingTime.minusMinutes(WITHIN\_MINUTE).isBefore(timeFormatted))) {

……

}

同理，检测事件类型和App类型：

if (logType.isBlank() || logType.equals(logType1)) {

if (appType.isBlank() || appType.equals(appType1)) {

……

}

}

方法Spec：

    /\*\*

     \* show logs

     \* @param *askedTime*

     \* @param *logType*

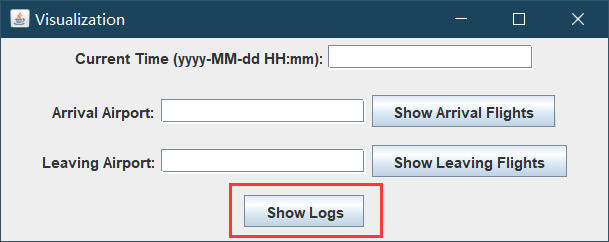
     \* @param *appType*

     \* @throws IOException

     \*/

    public void showLog(String *askedTime*, String *logType*, String *appType*) throws IOException {}

最后通过GUI显示出选择日志，方法的按钮在visualization内的“Show Logs”。



## Testing for Robustness and Correctness

### Testing strategy

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

        /\*

         \* Test strategy

         \*

         \* Location Conflict:

         \* Add entries from without conflict to with conflict.

         \* call method when they're added.

         \*

         \* Resource conflict:

         \* Add entries from without conflict to with conflict.

         \* call method when they're added.

         \*

         \* Find Pre Entry:

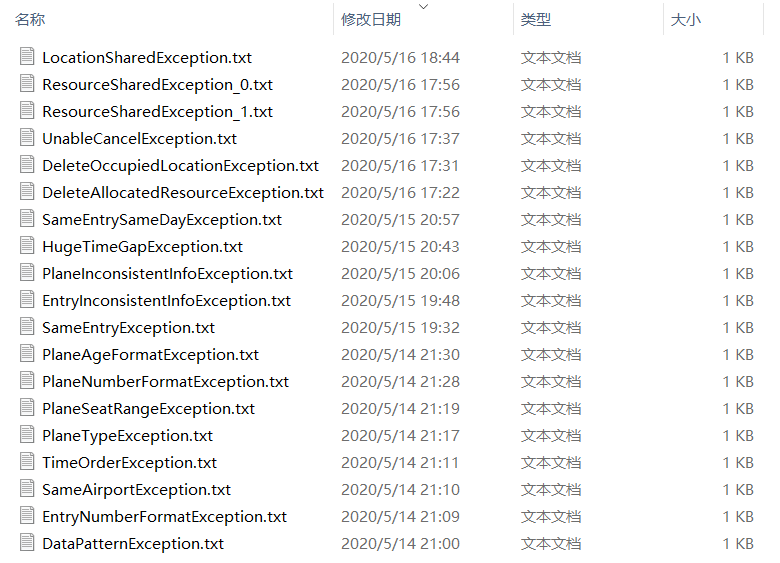
         \* Add entries from non-pre-entry to has it, and to new pre-entry.

         \* call method when they're added, and update the new entry closer to the asking.

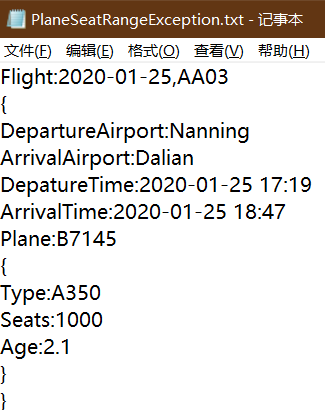
         \*/

### 测试用例设计

为每种Exception设计测试用例，保存在data/Exceptions/中。



主要有两种数据：最少需要一组数据的、最少需要两组数据的。当测试计划项数据文本内容各种Exception、以及资源相关Exception最少只需要一组数据。以PlaneSeatRangeException为例：



测试用例代码：

    @Test

    public void testPlaneSeatRangeException() throws Exception {

        exception.expect(PlaneSeatRangeException.class);

        exception.expectMessage(1000 + " is not in [50, 600].");

        String data = getOneData("data/Exceptions/PlaneSeatRangeException.txt");

        FlightScheduleCollection flightScheduleCollection = new FlightScheduleCollection();

        FlightSchedule<Resource> flightSchedule = flightScheduleCollection.addPlanningEntry(data);

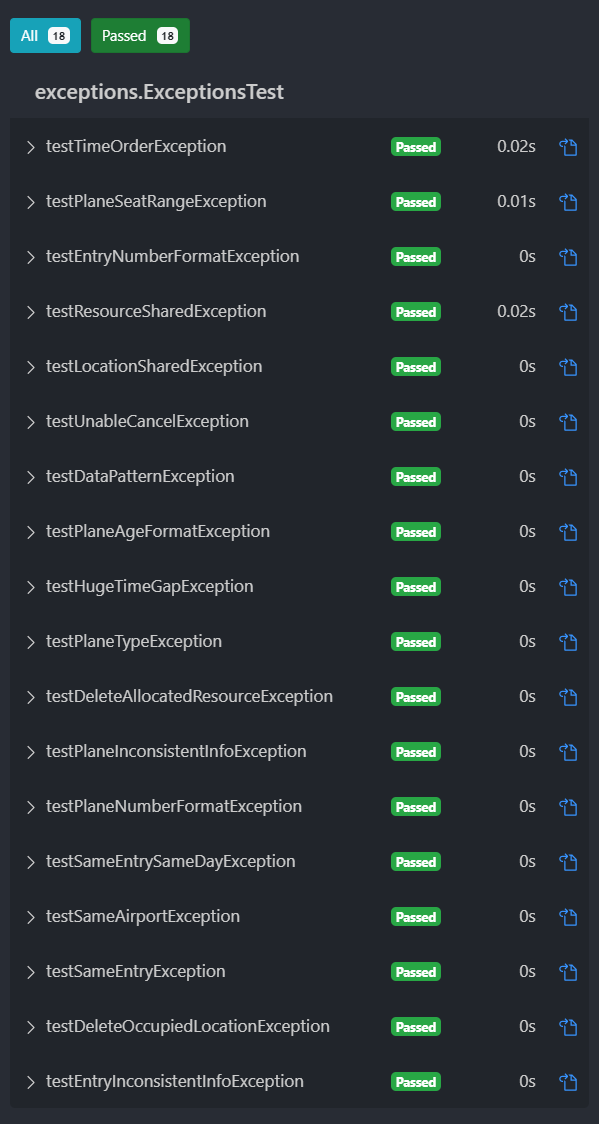
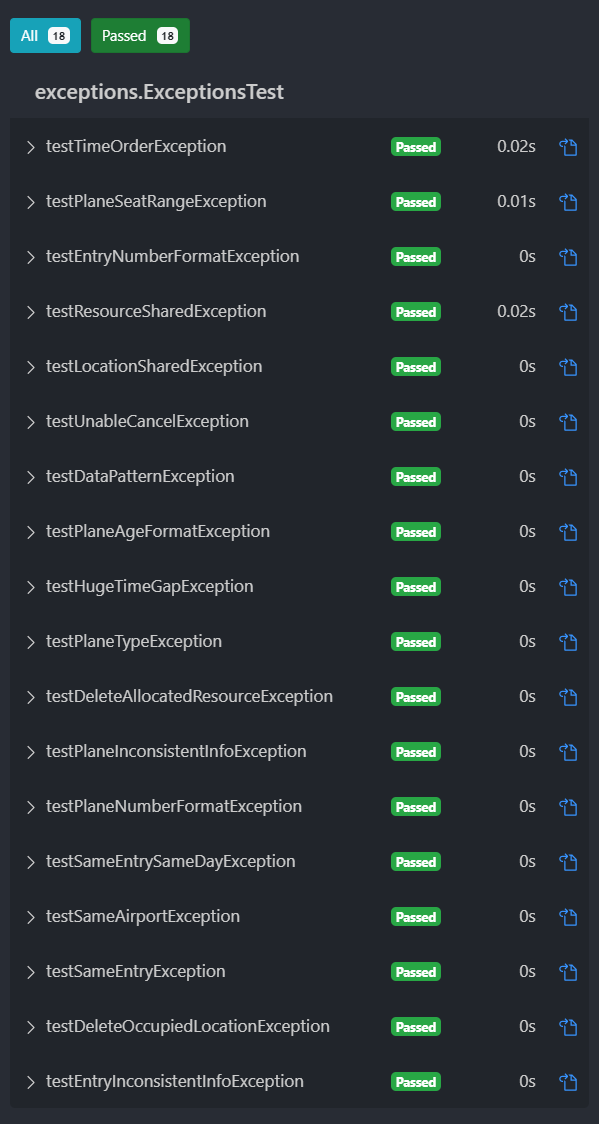
        flightScheduleCollection.allocatePlanningEntry(flightSchedule.getPlanningEntryNumber(), data);

    }

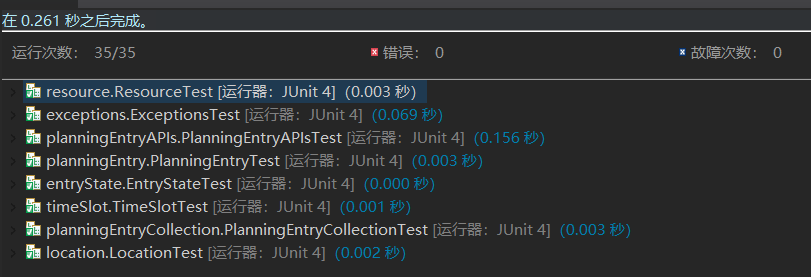
如此设计多种非法文件，促使程序出错，提高健壮性和正确性。

### 测试运行结果与EclEmma覆盖度报告

ExceptionTest测试运行结果：

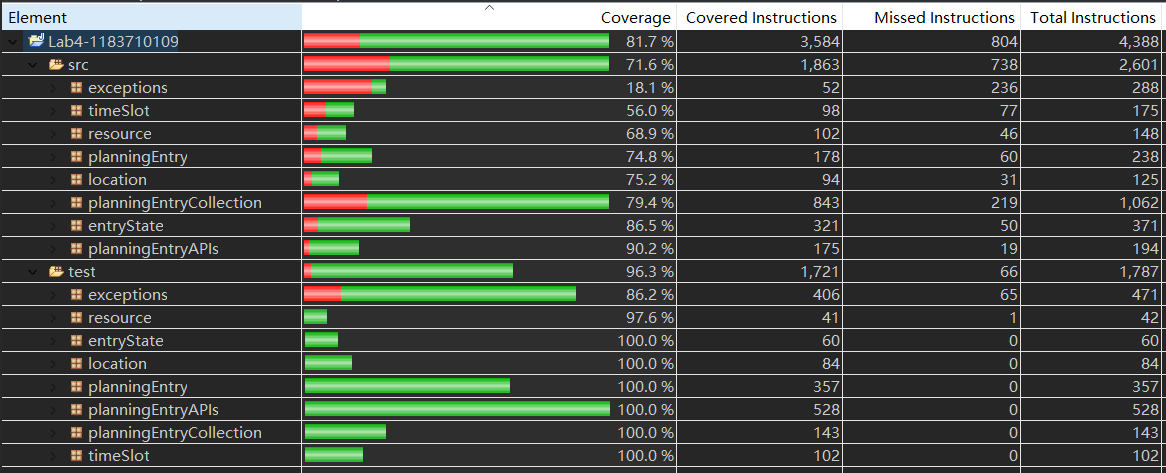


所有测试运行结果：

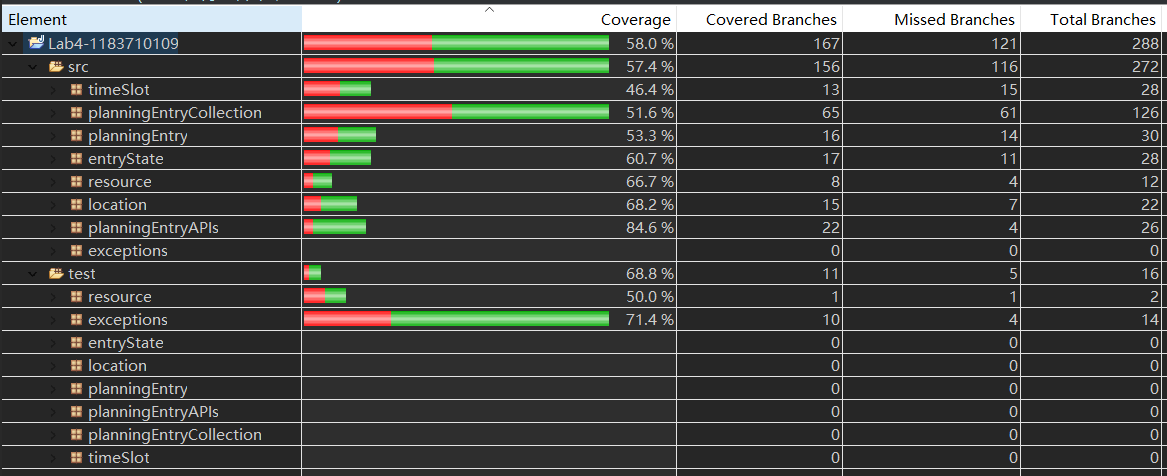


EclEmma覆盖度测试：

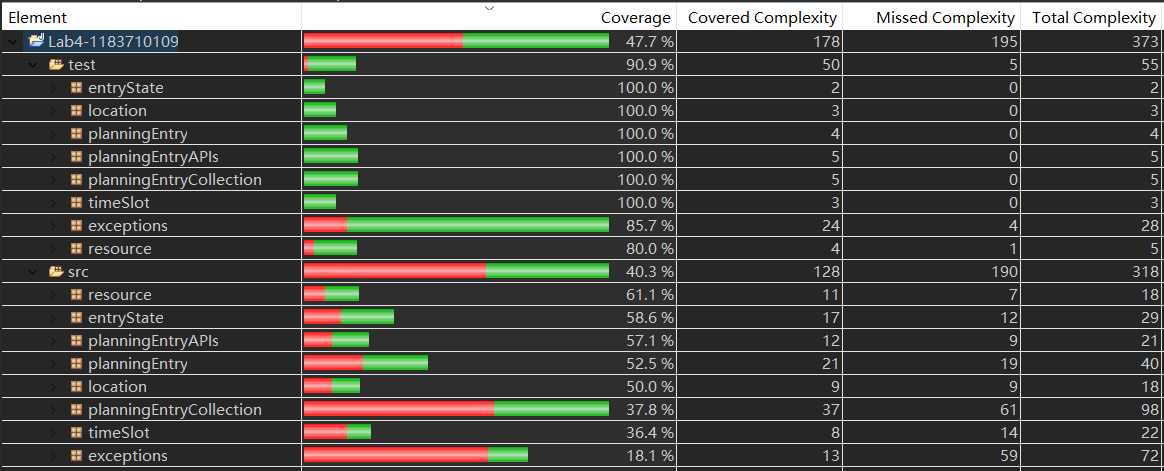
语句覆盖度：



分支覆盖度：



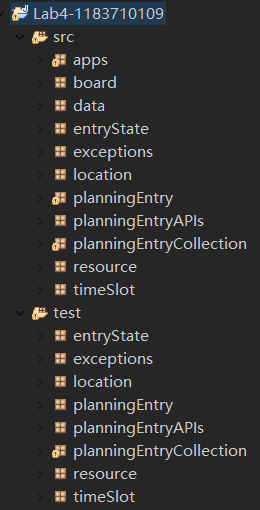
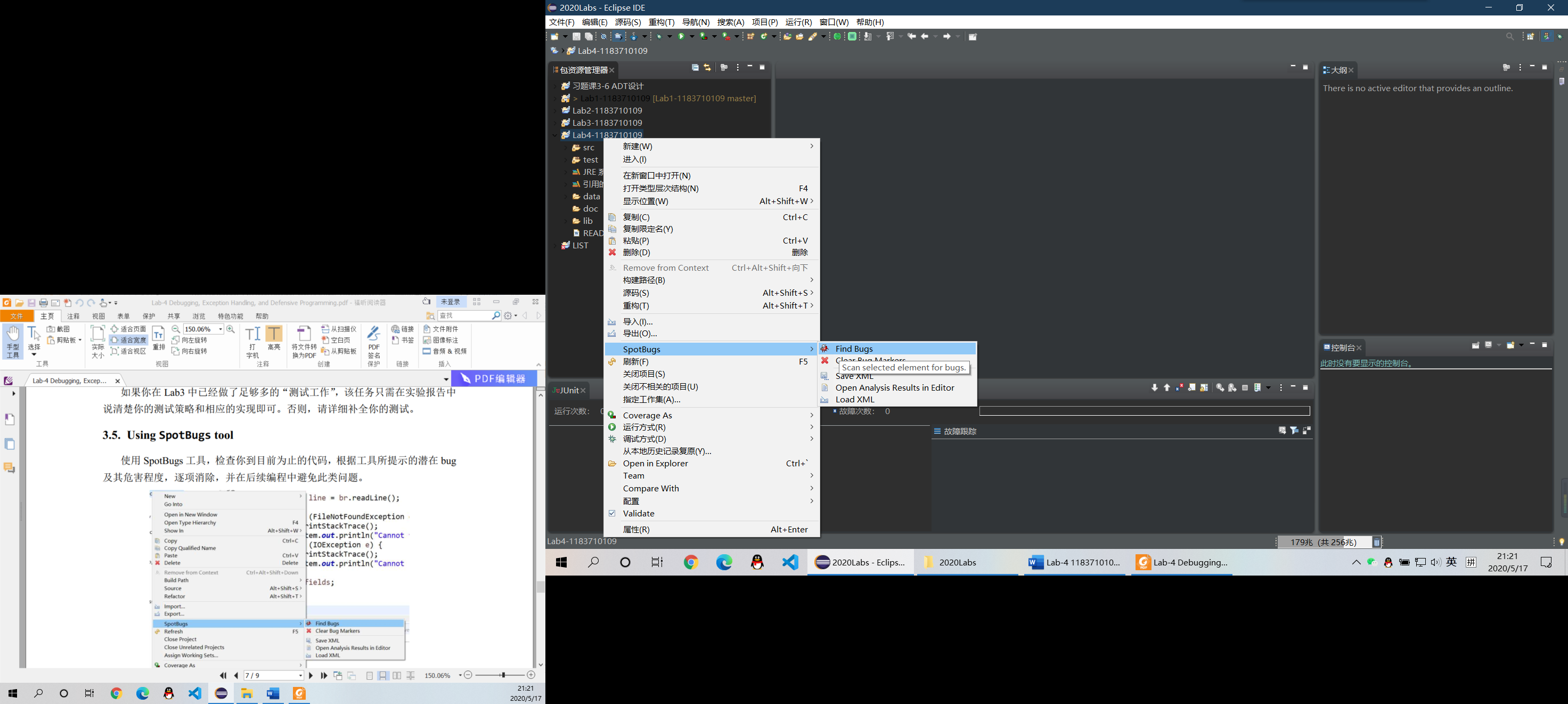
复杂度覆盖度：



## SpotBugs tool

没有Spot到Bugs。

之前有用到过，检测出实际返回类型和方法返回类型不一致的错误，还有可能为Null Pointer的错误。



## Debugging

### EventManager程序

理解待调试程序的代码思想：

该程序要求若干时间区间交集数量的最大值。start代表从某个点开始的日程数目（+start），end代表到某个点结束的日程（-end），统计某个点日程的重复数目，就是将这个点之前的start和end求和。

发现错误并修改：

* 方法开始要有限制条件，非法情况不能添加；

if ((day >= 1 && day <= 365 && start >= 0 && start < 24 && end > 0 && end <= 24 && start < end)) {

……

}

int active = 0, ans = 0;

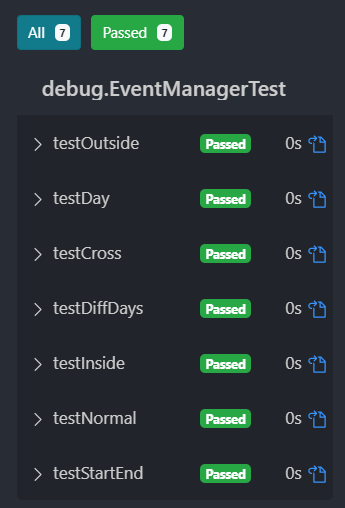
for (int d : temp.values()) {

……

}

* Map取值时可能为空，所以需要判断：if (temp.containsKey(key))
* 要将day考虑进去，所以key应该为start+day\*25或end+day\*25（可以为比25更大的整数值）；

修复之后的测试结果：



### LowestPrice程序

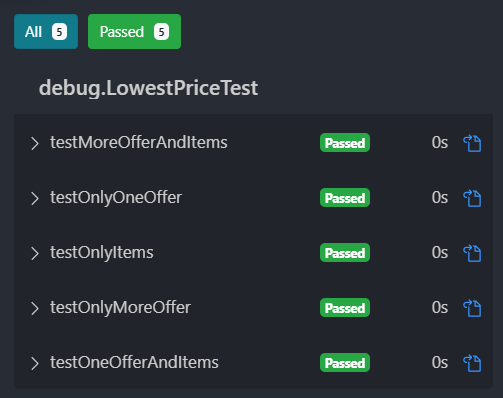
理解待调试程序的代码思想：

该程序想要求在有special offer的情况下最优价格。算法思想是贪心。首先假设最低代价为所有均用零售，然后每次将一个special offer加入“购物车”，更新需求，再用新需求迭代求解。

发现错误并修改：

* Needs没有进行深拷贝，使用addAll；
* 循环等场景下出现list下标等于size()情况，修改为needs.size()-1；
* s.get()参数错误，该j与special无关，修改为s.size()-1；
* 当循环存在差值小于0则break，应改为diff<0；
* 被使用过的special offer还有可能被使用，应该拷贝后去除；

修复之后的测试结果：



### FlightClient/Flight/Plane程序

理解待调试程序的代码思想：

该程序通过枚举每个航班，尝试安排飞机，确保没有与其他已经分配的航班冲突，最后确认是否能所有同时分配成功。

发现并修正错误：

* Collections.sort()参数缺失，应该添加Comparator，比较时间（我好像没觉得有什么作用）；

Comparator<Flight> comparator = new Comparator<Flight>() {

@Override

public int compare(Flight *o1*, Flight *o2*) {

return o1.getDepartTime().before(o2.getDepartTime()) ? -1 : 1;

}

};

Collections.sort(flights, comparator);

* While循环不退出，要用for顺序遍历所有plane后退出。
* 判断是否以分配，否则不比较：if (q == null || !q.equals(p))
* 在得出两个航班的起止时间后，比较错误，应该调用API改为：

if (fStart.before(tEnd) && fEnd.after(tStart)) {

bConflict = true;

break;

}

* 在判断是否能分配资源，应该是改航班不与任何航班冲突，若有一个冲突则不能分配。因此，要在确认所有航班均不冲突后（即bConflict==false），分配飞机并将bAllocated=true（添加在if中）；

if (!bConflict) {

f.setPlane(p);

bAllocated = true;

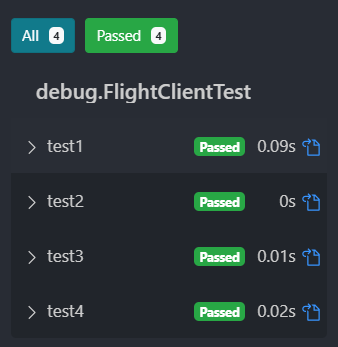
break;

}

* 未返回bFeasible，若存在航班bAllocated=false（即被分配），则bFeasible=false；

bFeasible &= bAllocated;

修复之后的测试结果：



# 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 |
| 2020-05-13 | 下午 | 初始化仓库 | 完成 |
| 2020-05-13 | 下午 | 完成3.1.1中8种语法错误 | 完成 |
| 2020-05-13 | 晚上 | 完成3.1.1中第9-11种错误 | 完成 |
| 2020-05-14 | 晚上 | 完成3.1.1中第12-13种错误；测试这13种错误 | 完成 |
| 2020-05-15 | 晚上 | 完成3.1.2中第1-3种错误 | 完成 |
| 2020-05-16 | 上午 | 完成3.1.2中第4-5种错误 | 完成 |
| 2020-05-16 | 晚上 | 完成3.1.2的5种错误测试 | 完成 |
| 2020-05-17 | 下午 | 完成3.2断言和checkRep() | 完成 |
| 2020-05-17 | 晚上 | 通过所有测试 | 完成 |
| 2020-05-18 | 晚上 | 设计日志功能 | 初步完成 |
| 2020-05-19 | 上午 | 完成3.3日志记录和查询功能 | 完成 |
| 2020-05-19 | 下午 | 3.4-3.5补充测试策略、修改Bug | 完成 |
| 2020-05-19 | 晚上 | 完成3.6.2和3.6.3 | 完成 |
| 2020-05-21 | 下午 | 完成3.6.1 | 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 |
| Exception在APP内无法测试 | 设置check静态方法，调用静态方法测试 |
| APP捕获异常无法恢复/修复数据 | 直接抛出异常或保存先前状态 |
| EventManager希望能start到end都标记，改动太大 | Start++，end++，在区间内统计最大值 |

# 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

* 程序里一定有很多的潜在Bug；
* 需要提前预期错误，设置Exception和防御措施来保证程序的正确性和健壮性；
* 测试的代码量可能很大，取决于对正确性和健壮性的追求

## 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？
   1. 健壮性，希望能应对所有的错误，因此会考虑每一步的防御措施；
   2. 正确性，希望在任何时候都正确；因此会追求结果的正确性；
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？）
   1. 要估计成本于期望风险的价值关系；
   2. 民航飞机不安装降落伞，是因为大部分人不会用，机组有机会跑了就没法全力拯救飞机，而且概率小，成本太高。
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？
   1. 前者用户爽，~~（我会在用户给了很多很多钱的情况下遵循）~~，我会在程序失败后代价极高的场景下使用；
   2. 后者我爽，我会在用户（也许是我）非常清楚使用方法，或是出错概率不大的情况下使用；
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？
   1. 考虑开发成本和风险期望；
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？
   1. 有；
   2. 会；
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？
   1. 覆盖到每一个等价类；
   2. 语句覆盖度100%不一定，复杂度100%意味着接近100%充分；
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。
   1. 有乐趣，前提是我找的出来，而且找出来的了有所收获；
8. 关于本实验的工作量、难度、deadline。
   1. 适中、适中、较长；
9. 到目前为止你对《软件构造》课程的评价和建议。
   1. 课程很好；
   2. 实验分数实在是太低了，我们马原100个字的作业值7分，在这门课100行代码都不值1分……我抗议，实验分数太低，建议提高；
10. 期末考试临近，你对占成绩60%的闭卷考试有什么预期？
    1. 线下考；
    2. 不要执着于一些概念性的东西，尤其是细节；
    3. 考一些我们能从实验里学到的东西（实验花的时间太多，分数又少），比如实验中的细节、自己的思路、实验挖坑拓展、新需求、新风险……不要让实验只有可怜的30分（考试里考实验还能惩罚实验作弊的同学）；