

**2020年春季学期  
计算机学院《软件构造》课程**

**Lab 3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](#_Toc39577195)

[1 实验目标概述 1](#_Toc39577196)

[2 实验环境配置 1](#_Toc39577197)

[3 实验过程 2](#_Toc39577198)

[3.1 待开发的三个应用场景 2](#_Toc39577199)

[3.2 面向可复用性和可维护性的设计：PlanningEntry<R> 2](#_Toc39577200)

[3.2.1 PlanningEntry<R>的共性操作 3](#_Toc39577201)

[3.2.2 局部共性特征的设计方案 4](#_Toc39577202)

[3.2.3 面向各应用的PlanningEntry子类型设计（个性化特征的设计方案） 4](#_Toc39577203)

[3.3 面向复用的设计：R 7](#_Toc39577204)

[3.4 面向复用的设计：Location 8](#_Toc39577205)

[3.5 面向复用的设计：Timeslot 9](#_Toc39577206)

[3.6 面向复用的设计：EntryState及State设计模式 10](#_Toc39577207)

[3.7 面向应用的设计：Board 13](#_Toc39577208)

[3.8 Board的可视化：外部API的复用 15](#_Toc39577209)

[3.9 PlanningEntryCollection的设计 17](#_Toc39577210)

[3.10 可复用API设计及Façade设计模式 21](#_Toc39577211)

[3.10.1 检测一组计划项之间是否存在位置独占冲突 21](#_Toc39577212)

[3.10.2 检测一组计划项之间是否存在资源独占冲突 22](#_Toc39577213)

[3.10.3 提取面向特定资源的前序计划项 23](#_Toc39577214)

[3.11 设计模式应用 24](#_Toc39577215)

[3.11.1 Factory Method 24](#_Toc39577216)

[3.11.2 Iterator 25](#_Toc39577217)

[3.11.3 Strategy 25](#_Toc39577218)

[3.12 应用设计与开发 26](#_Toc39577219)

[3.12.1 航班应用 26](#_Toc39577220)

[3.12.1.1 初始化数据 26](#_Toc39577221)

[3.12.1.2 起始界面 27](#_Toc39577222)

[3.12.1.3 可视化 28](#_Toc39577223)

[3.12.1.4 新建计划项 28](#_Toc39577224)

[3.12.1.5 分配资源 29](#_Toc39577225)

[3.12.1.6 询问状态 30](#_Toc39577226)

[3.12.1.7 计划项操作（启动、取消、暂停、结束） 30](#_Toc39577227)

[3.12.1.8 API：检查冲突、查找前置计划项 30](#_Toc39577228)

[3.12.1.9 管理（增加/删除）资源 31](#_Toc39577229)

[3.12.1.10 管理（增加/删除）位置 32](#_Toc39577230)

[3.12.1.11 同一资源的计划项 32](#_Toc39577231)

[3.12.2 高铁应用 33](#_Toc39577232)

[3.12.3 学习活动应用 33](#_Toc39577233)

[3.13 基于语法的数据读入 34](#_Toc39577234)

[3.13.1.1 航班 34](#_Toc39577235)

[3.13.1.2 高铁 34](#_Toc39577236)

[3.13.1.3 学习活动 35](#_Toc39577237)

[3.14 应对面临的新变化 36](#_Toc39577238)

[3.14.1 变化1：航班 36](#_Toc39577239)

[3.14.2 变化2：高铁 37](#_Toc39577240)

[3.14.3 变化3：学习活动 37](#_Toc39577241)

[3.15 Git仓库结构 37](#_Toc39577242)

[4 实验进度记录 38](#_Toc39577243)

[5 实验过程中遇到的困难与解决途径 39](#_Toc39577244)

[6 实验过程中收获的经验、教训、感想 40](#_Toc39577245)

[6.1 实验过程中收获的经验和教训 40](#_Toc39577246)

[6.2 针对以下方面的感受 40](#_Toc39577247)

# 实验目标概述

本次实验覆盖课程第 3、4、5 章的内容，目标是编写具有可复用性和可维护性的软件，主要使用以下软件构造技术：

* 子类型、泛型、多态、重写、重载
* 继承、代理、组合
* 常见的 OO 设计模式
* 语法驱动的编程、正则表达式
* 基于状态的编程
* API 设计、API 复用

本次实验给定了五个具体应用（高铁车次管理、航班管理、操作系统进程管理、大学课表管理、学习活动日程管理），学生不是直接针对五个应用分别编程实现，而是通过 ADT 和泛型等抽象技术，开发一套可复用的 ADT 及其实现，充分考虑这些应用之间的相似性和差异性，使 ADT 有更大程度的复用（可复用性）和更容易面向各种变化（可维护性）。

# 实验环境配置

<https://github.com/ComputerScienceHIT/Lab3-1183710109>

# 实验过程

## 待开发的三个应用场景

列出你所选定的三个应用。

分析三个应用场景的异同，理解需求：它们在哪些方面有共性、哪些方面有差异。

我选择的三个应用场景：

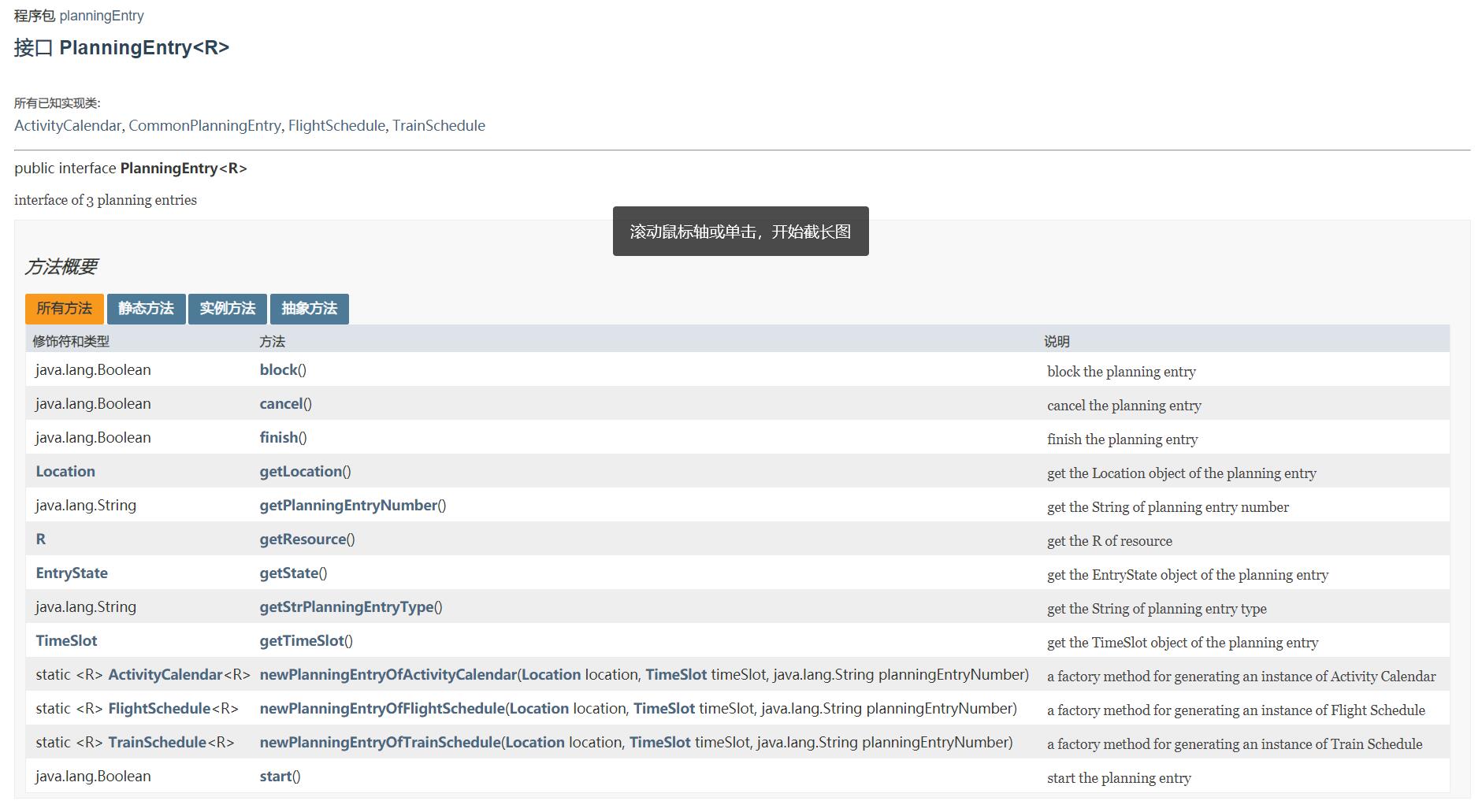
* 航班管理
* 高铁车次管理
* 学习日程管理

这三个场景的异同点：

* 位置的数量：分别为1个、2个和多个
* 仅有学习日程的位置可更改
* 航班为单个资源，高铁为有序多个资源，学习日程为无序多个资源
* 仅有高铁车次可阻塞
* 时间均在创建时设定

## 面向可复用性和可维护性的设计：PlanningEntry<R>

该节是本实验的核心部分。

计划项是一个状态可变的ADT，它保存有一个计划项的时间、地点、资源等有效信息。PlanningEntry在我的设计中是一个接口，设计有各种计划项均要实现的方法以及工厂方法。

### PlanningEntry<R>的共性操作

3个工厂方法分别能够返回指定类型的PlanningEntry实现类，以Flight Schedule为例：

    /\*\*

     \* a factory method for generating an instance of Flight Schedule

     \* @param *<R>*

     \* @param *location*

     \* @param *timeSlot*

     \* @param *planningEntryNumber*

     \* @return an empty instance of planning entry of flight schedule

     \*/

    public static <R> FlightSchedule<R> newPlanningEntryOfFlightSchedule(Location *location*, TimeSlot *timeSlot*,

            String *planningEntryNumber*) {

        return new FlightSchedule<R>(location, timeSlot, planningEntryNumber);

    }

状态的转换，以目标状态进行分类，能够将状态转换为RUNNING、BLOCKED、CANCELLED、ENDED四种之一（其中转换为ALLOCATED是个性化设计），分别用4个方法来实现。以start()为例：

    /\*\*

     \* start the planning entry

     \* @return true if the entry is started

     \*/

    public Boolean start();

Getter()包括了获取Location、TimeSlot、State、Type、Number、Type一些共有的信息对象。

### 局部共性特征的设计方案

CommonPlanningEntry类实现了PlanningEntry接口中共性方法，包括了状态转换和Getter方法。

状态转换，以Start()为例：将状态转换委派给state对象的Setter操作，通过常量来进行目标状态的区分。在state对象中，首先判断该转换是否合法（访问EntryStateEnum静态常量进行判断），然后在进行状态覆盖，最后返回操作成功与否的标识。

    @Override

    public Boolean start() {

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

    }

Getter操作访问CommonPlanningEntry中定义的共性成员变量，包括Location、Resource等等。以getLocation()为例：

    @Override

    public Location getLocation() {

        return this.location;

    }

此外，设计抽象方法getPlanningDate()等Spec相同的方法。

    /\*\*

     \* get the planning date

     \* @return LocalDate of planning date

     \*/

    public abstract LocalDate getPlanningDate();

### 面向各应用的PlanningEntry子类型设计（个性化特征的设计方案）

3个子类型的不同主要在于两方面：Location、TimeSlot、Resource等信息的存储模式和信息的修改。

存储模式：在我的设计中，信息存储的差异统一合并到“信息对象”的内部中，通过不同子类型的不同Getter来得到相应的信息细节。信息对象具体设计在3.3-3.6说明。

以Flight Schedule为例，getLocationOrigin()、getLocationTerminal()方法获得了起飞、降落机场；而在Activity Calendar中则用getStrLocation()获得活动地点。

    /\*\*

     \* get the origin location object

     \* @return the origin location object

     \*/

    public String getLocationOrigin() {

        return super.getLocation().getLocations().get(ORIGIN);

    }

    /\*\*

     \* get the terminal location object

     \* @return the terminal location object

     \*/

    public String getLocationTerminal() {

        return super.getLocation().getLocations().get(TERMINAL);

}

由于Location存储为List，因此List大小分别为1、2、n，根据不同的计划项特点设计Getter即可实现不同的特性。

信息修改：根据不同计划项信息的修改特点进行设计。例如allocateResource()，飞机只能分配一个资源，而高铁为多个有序资源（用List<R>存储），活动为多个无序资源。以高铁的分配资源为例：

    /\*\*

     \* allocate the resource to the flight schedule

     \* set the state as ALLOCATED

     \* @param *resources*

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

     \*/

    public Boolean allocateResource(R... *resources*) {

        this.resources.addAll(Arrays.asList(resources));

        this.ORIGIN = 0;

        this.LENGTH = this.resources.size();

        this.TERMINAL = this.resources.size() - 1;

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

    }

通过不定项的资源作为参数，然后保存到list中，获取长度、起终点标识，并设置状态。

此外，Activity Calendar可以在开始前设置新地点，则再该计划项子类中添加对应的Setter方法。

    /\*\*

     \* set a new location

     \* @param *strNewLocation*

     \*/

    public void setNewLocation(String *strNewLocation*) {

        if (this.getState().getStrState().equals("ALLOCATED"))

            this.location = new Location(strNewLocation);

    }

最后，重写不同的equals、hashcode、toString方法。以Activity为例：

    @Override

    public String toString() {

        return "{" + " intResourceNumber='" + getIntResourceNumber() + "'" + "}";

    }

    @Override

    public boolean equals(Object *o*) {

        if (o == this)

            return true;

        if (!(o instanceof ActivityCalendar)) {

            return false;

        }

        ActivityCalendar<R> activityCalendar = (ActivityCalendar<R>) o;

        return intResourceNumber == activityCalendar.intResourceNumber;

    }

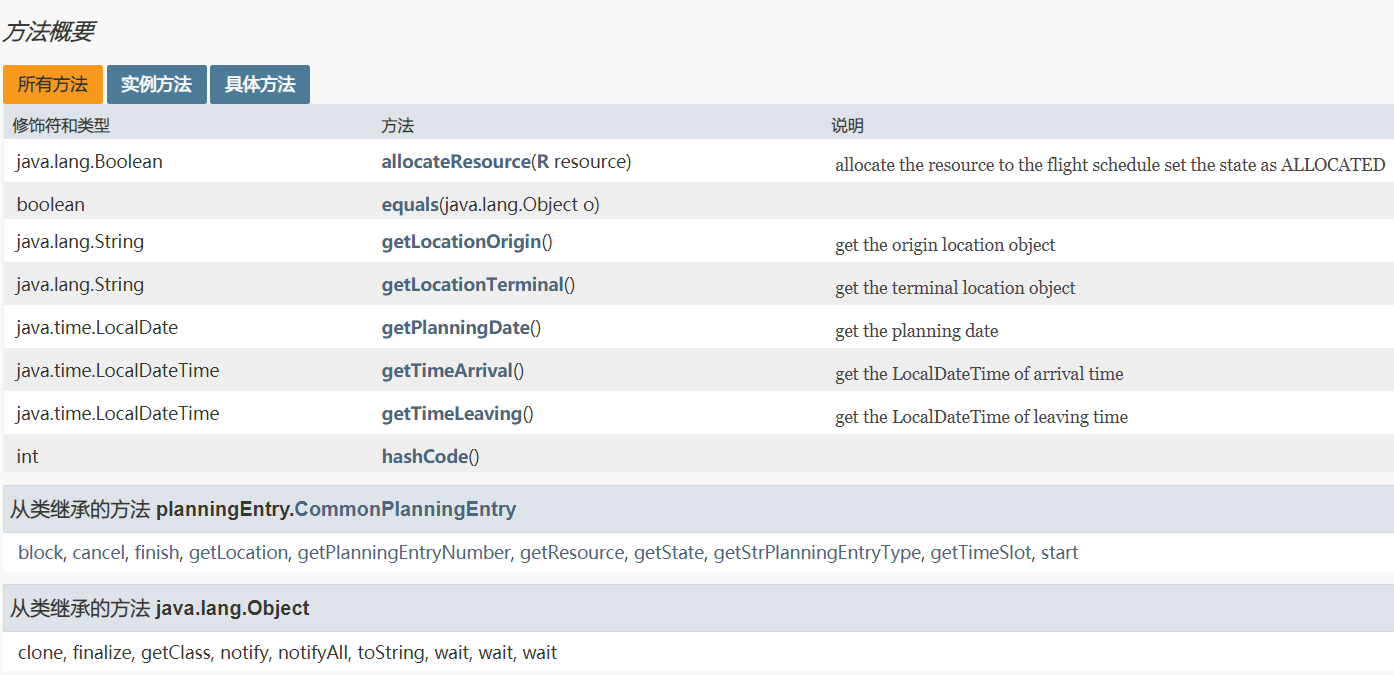
    @Override

    public int hashCode() {

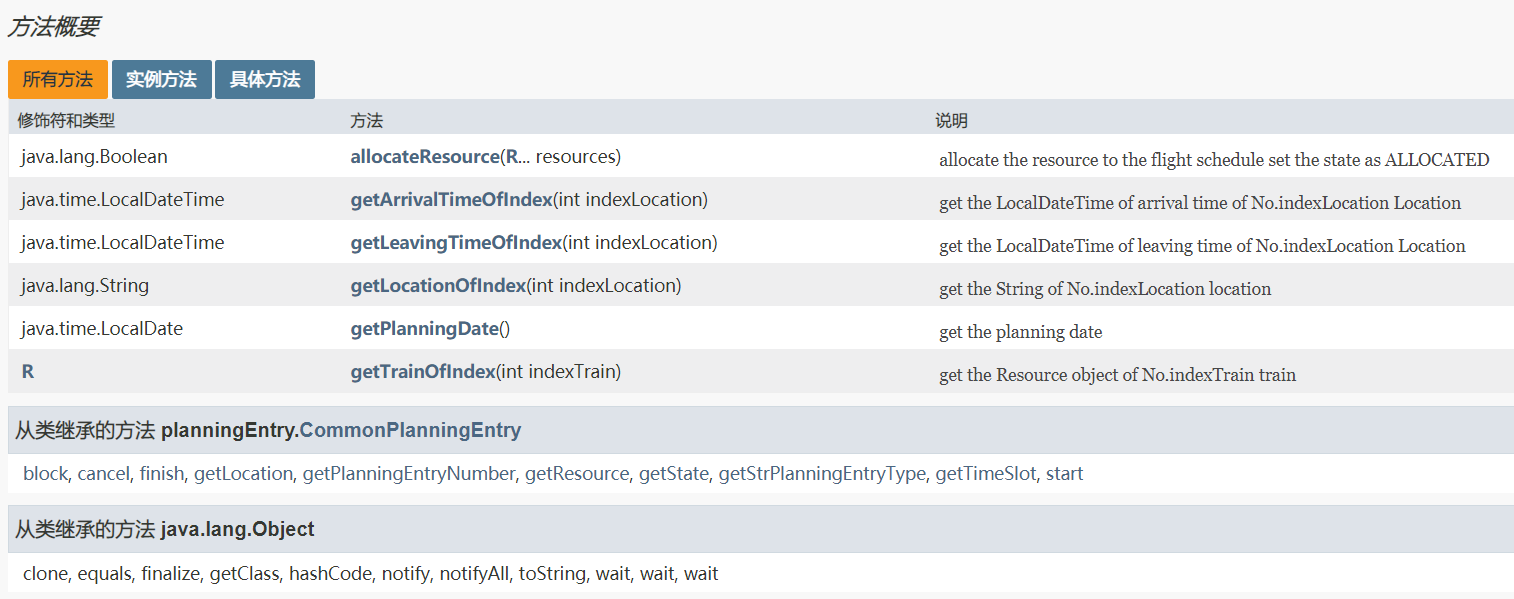
        return Objects.hashCode(intResourceNumber);

}

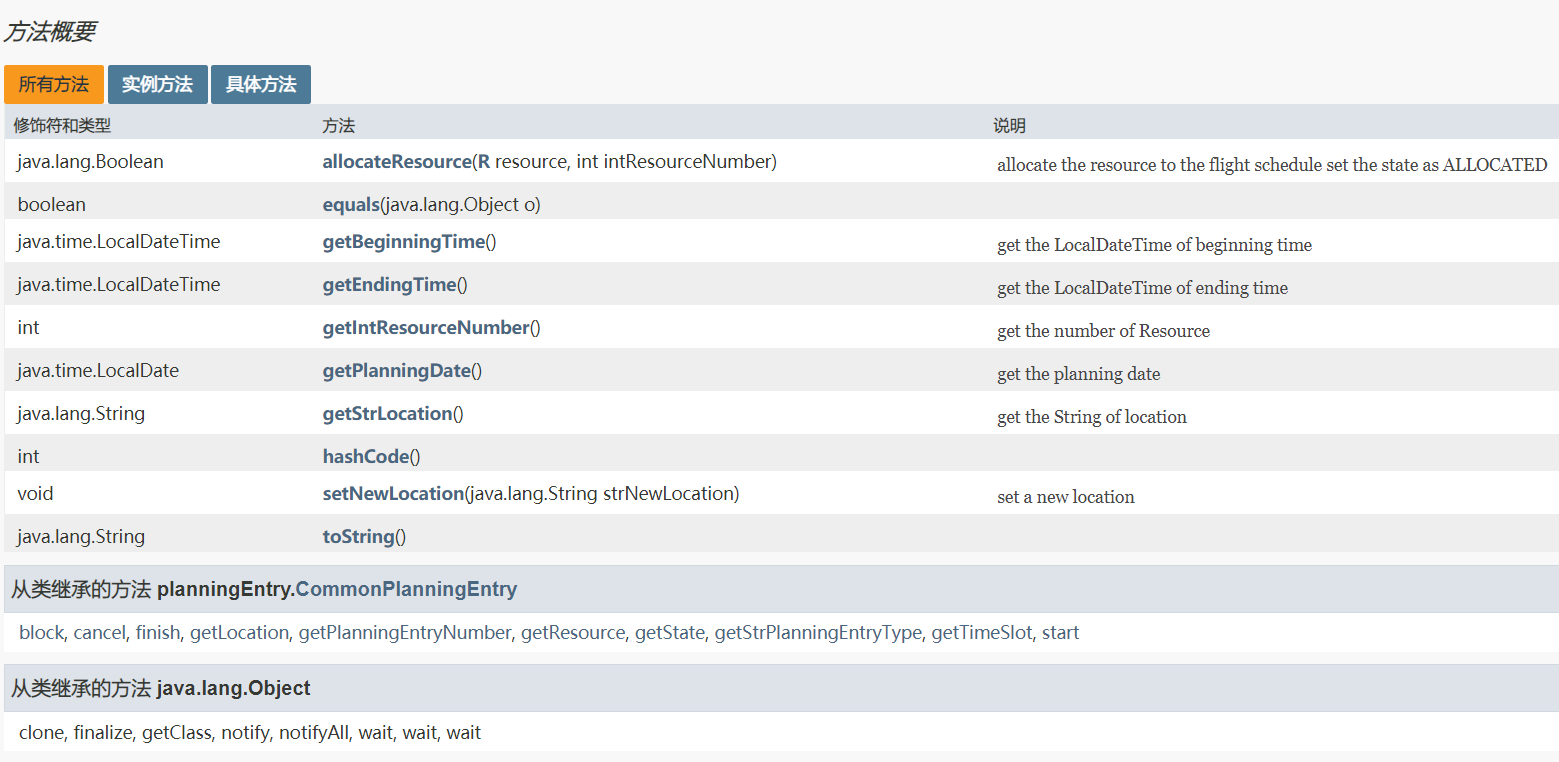
航班管理javadoc



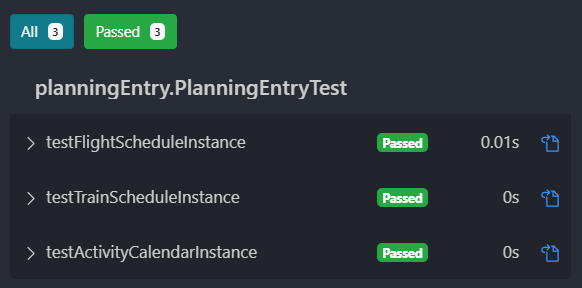
高铁管理javadoc



活动日程javadoc

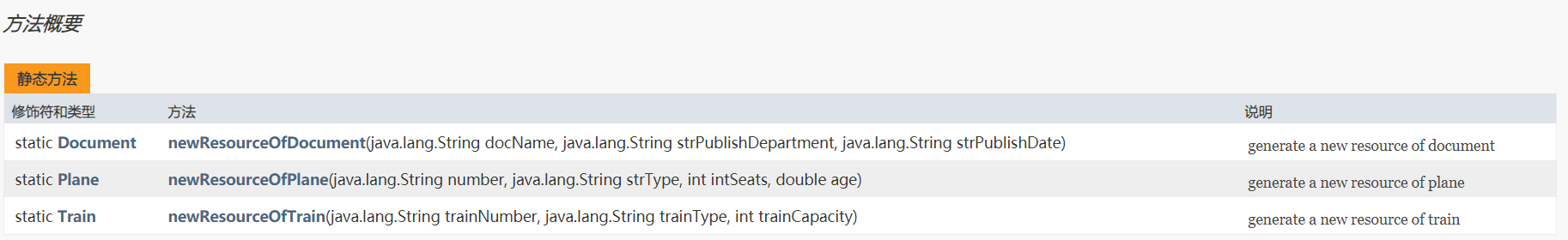


JUnit测试

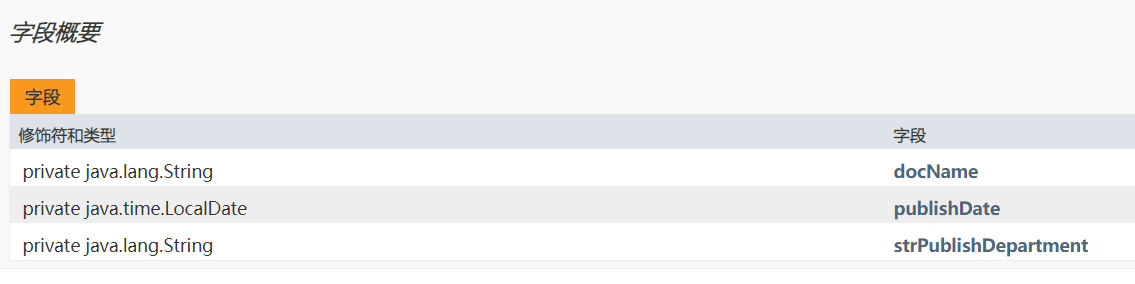


## 面向复用的设计：R

资源有多种，因为Resource被设计为一个接口，有3个实现类：Plane、Train和Document。Resource接口中有3种子类的工厂方法。



3种子类存储有各自的独特信息，以Document为例：



其中publishDate在构造时用String输入，降低前置条件，并在构造方法中转换。

this.publishDate = LocalDate.parse(strPublishDate, DateTimeFormatter.ofPattern("yyyy-MM-dd"));

3种子类均为immutable，设计有各个成员变量的Getter，并且根据要求重写equals等。以Plane为例：

    @Override

    public boolean equals(Object *o*) {

        if (o == this)

            return true;

        if (!(o instanceof Plane)) {

            return false;

        }

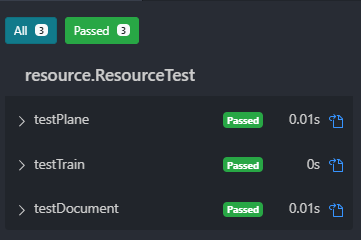
        Plane plane = (Plane) o;

        return Objects.equals(number, plane.number) && Objects.equals(strType, plane.strType)

                && intSeats == plane.intSeats && age == plane.age;

    }

Junit测试



## 面向复用的设计：Location

由于我选择的3种计划项位置数量各不相同，因此我采用一个List来存储若干的位置，通过PlanningEntry的不同Getter来获取。

Field、AF、RI、Safety：

    private final List<String> locations = new ArrayList<String>();

    /\*

     \* 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

     \*/

在构造器中，参数为若干个String类型地址，将这些String均加入List。

    /\*\*

     \* constructor

     \* @param *locations*

     \*/

    public Location(String... *locations*) {

        for (String str : locations)

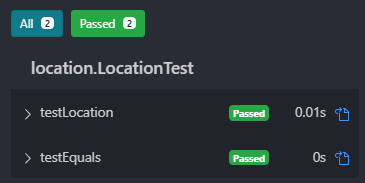
            this.locations.add(str);

        checkRep();

    }

因此，Location本质上是一个存储有多个String的List。

Junit测试



## 面向复用的设计：Timeslot

Time Slot和Location是共同设计的，存储有两个List，分别代表对应的位置的到达和离开时间。由此设计，可以精确到每个地点的到达和离开时间，若为第一个地点，则到达和离开时间相同；若为最后一个地点也如此；若只有一个地点，则也如此。

由此，3种不同的计划项，通过不同的Getter实现不同的特征。

    private final List<LocalDateTime> arrival = new ArrayList<>();

    private final List<LocalDateTime> leaving = new ArrayList<>();

    /\*

     \* 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

     \*/

## 面向复用的设计：EntryState及State设计模式

EntryState是一个可变对象，成员变量有类型为enum的state。

    private EntryStateEnum state;

AF、RI：

    /\*

     \* 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

     \*/

在构造方法中，通过字符串参数toUpperCase，再对应到EntryStateEnum中的某一个枚举，进行初始化。

    /\*\*

     \* constructor

     \* @param *stateName*

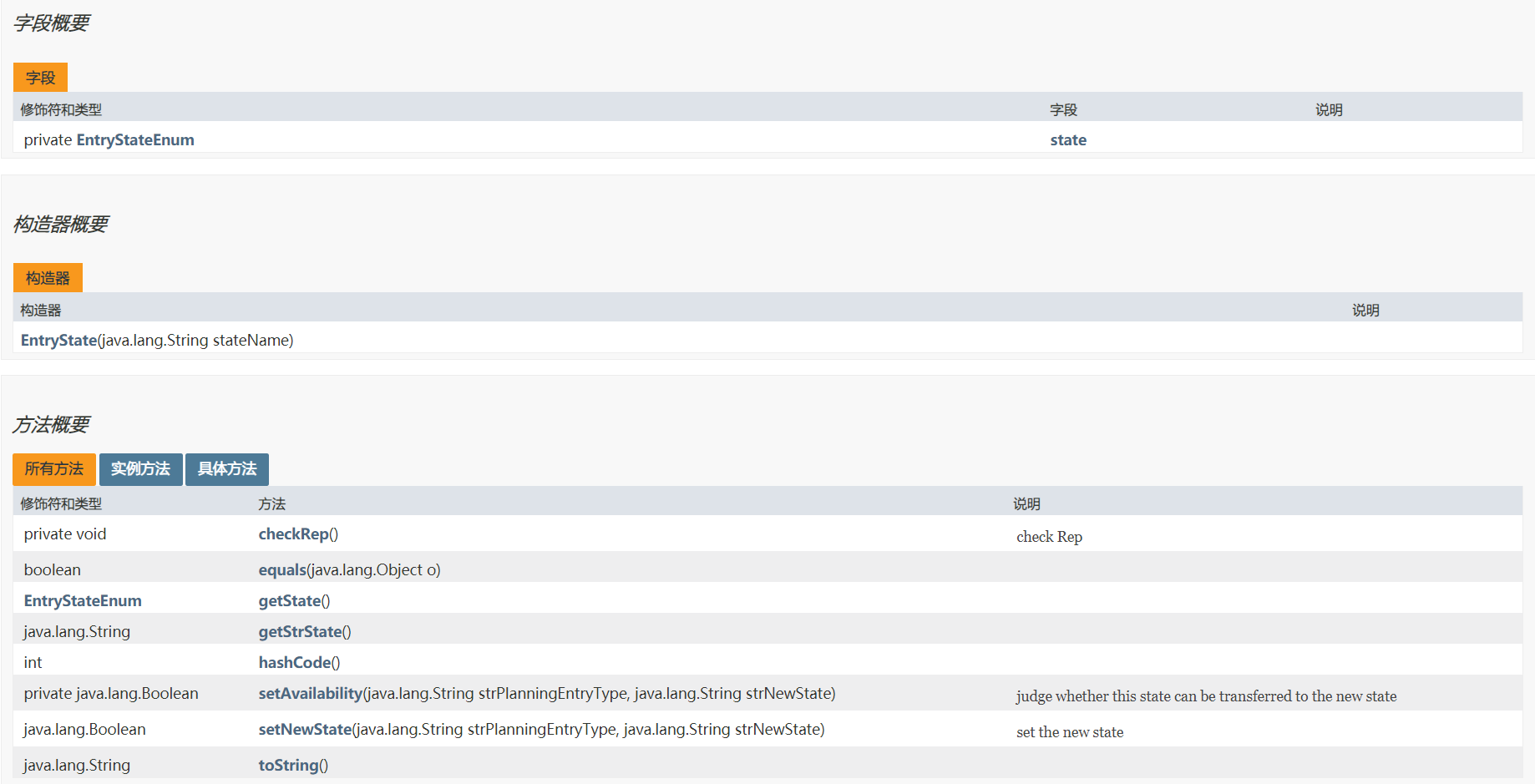
     \*/

    public EntryState(String *stateName*) {

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

        checkRep();

    }



状态是可变的，因此它需要设置一个Setter，即setNewState()。由于不同的计划项类型，可以设置的state不同，因此参数需要有计划项类型和新状态的字符串。这样的一个方法可以满足各种状态的转换。

    /\*\*

     \* 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());

            return true;

        }

        return false;

    }

判断合法性的工作交给另一个范围值为Boolean的方法setAvailability()，而该方法又将这项工作委派给EntryStateEnum中的静态Map变量。该Map分为两种，一种是可能被Block的，一种则不行。判断是否可以Block的工作在EntryStateEnum中进行，用一个List<String>来保存可以Block的类型的关键字（增强鲁棒性）。

    /\*\*

     \* judge whether this state can be transferred to the new state

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

     \* @param *strNewState*

     \* @return true if the current state can be transferred to the new state, false if not

     \*/

    private Boolean setAvailability(String *strPlanningEntryType*, String *strNewState*) {

        List<EntryStateEnum> availableStatesList = new ArrayList<EntryStateEnum>(

                Arrays.asList(this.getState().newStateAchievable(strPlanningEntryType)));

        return availableStatesList.contains(EntryStateEnum.valueOf(strNewState.toUpperCase()));

    }

该方法基于委派EntryState查询“可以到达的新状态”的一个List，确认新状态在该List中，的方法来确定Availability。EntryStateEnum有这些枚举变量：

    /\*\*

     \* they represent 6 states of the planning entry

     \*/

    WAITING, ALLOCATED, RUNNING, BLOCKED, ENDED, CANCELLED;

静态存储能/不能Block的“可以到达的新状态”的Map，Key为该枚举，Value为List<EntryStateEnum>。通过匿名对象初始化方法来初始化。

    /\*\*

     \* achievable states map for entries able to be blocked

     \*/

    public static final Map<EntryStateEnum, EntryStateEnum[]> newStateAchievableBlockedAble = new HashMap<EntryStateEnum, EntryStateEnum[]>() {

        private static final long serialVersionUID = 1L;

        {

            put(WAITING, new EntryStateEnum[] { ALLOCATED, CANCELLED });

            put(ALLOCATED, new EntryStateEnum[] { RUNNING, CANCELLED });

            put(RUNNING, new EntryStateEnum[] { BLOCKED, ENDED });

            put(BLOCKED, new EntryStateEnum[] { RUNNING, CANCELLED });

            put(CANCELLED, new EntryStateEnum[] {});

            put(ENDED, new EntryStateEnum[] {});

        }

    };

    /\*\*

     \* achievable states map for entries not able to be blocked

     \*/

    public static final Map<EntryStateEnum, EntryStateEnum[]> newStateAchievableBlockedDisable = new HashMap<EntryStateEnum, EntryStateEnum[]>() {

        private static final long serialVersionUID = 1L;

        {

            put(WAITING, new EntryStateEnum[] { ALLOCATED, CANCELLED });

            put(ALLOCATED, new EntryStateEnum[] { RUNNING, CANCELLED });

            put(RUNNING, new EntryStateEnum[] { ENDED });

            put(CANCELLED, new EntryStateEnum[] {});

            put(ENDED, new EntryStateEnum[] {});

        }

    };

保存可以Block的计划项名称：

    /\*\*

     \* define which is able to be blocked

     \*/

    public static final List<String> keyWords = new ArrayList<String>() {

        private static final long serialVersionUID = 1L;

        {

            add("Train");

        }

    };

建立一个属于枚举的成员方法，返回“可以到达的新状态”。调用原状态的枚举对象查询该List（即this代表当前状态）：

    /\*\*

     \* get all states achievable

     \* @param *strPlanningEntryType*

     \* @return array of the states

     \*/

    public EntryStateEnum[] newStateAchievable(String *strPlanningEntryType*) {

        for (String str : keyWords)

            if (strPlanningEntryType.contains(str))

                return EntryStateEnum.newStateAchievableBlockedAble.get(this);

        return EntryStateEnum.newStateAchievableBlockedDisable.get(this);

    }

因此，在状态模式的设计种，一次设置新状态的操作，经过：

PlanningEntryCollection

-> PlanningEntry

-> EntryState.setNewState() {

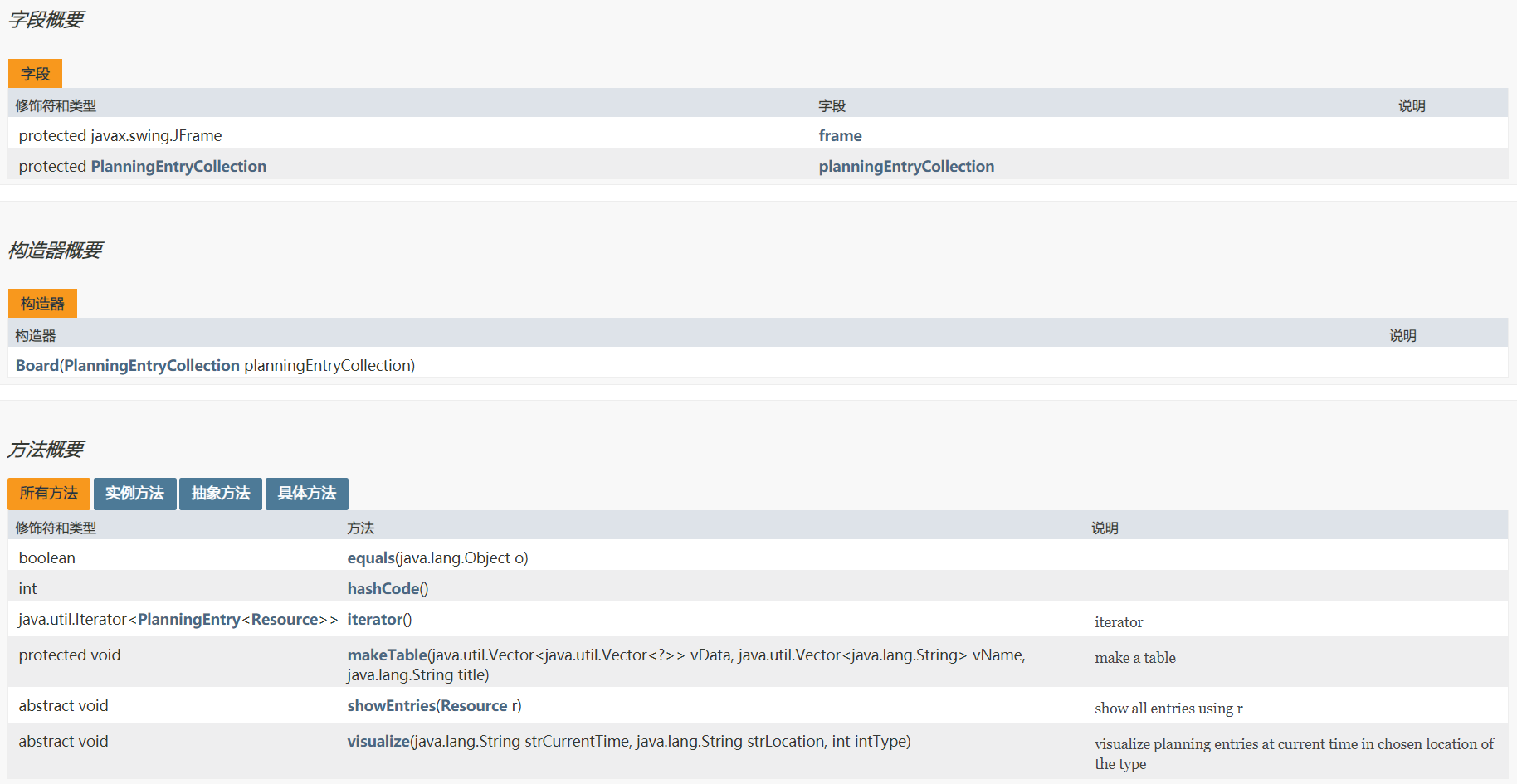
EntryState.setAvailability() -> EntryStateEnum.newStateAchievable()

}

完成一次指定操作。其中，在PlanningEntryCollection和PlanningEntry中采用外观模式包装成5个方法，分别到达5种状态。

## 面向应用的设计：Board

Board是每个地方的信息板，以机场为例，每个机场有1小时内到达航班和起飞航班的显示。Board是一个抽象类，有3个不同的实现类，分别完成3个应用场景的Board。在初始化时，保存PlanningEntryCollection作为成员变量，以便遍历PlanningEntry。并构造一个JFrame用于可视化。



下面里Flight Board为例：

设定参数：

    /\*\*

     \* choose flights within HOURS\_RANGE before or later

     \*/

    private static final int HOURS\_RANGE = 1;

    /\*\*

     \* visualization label of arrival

     \*/

    public static final int ARRIVAL = 1;

    /\*\*

     \* visualization label of leaving

     \*/

    public static final int LEAVING = -1;

构造方法，继承父类Board的构造函数：

    public FlightBoard(PlanningEntryCollection *planningEntryCollection*) {

        super(planningEntryCollection);

    }

在可视化时，输入为当前时间（也可以即时获得）、位置字符串和类型（起飞/到达），若位置为空，则认为查询所有机场。

    /\*\*

     \* visualize planning entries at current time in chosen location of the type

     \* @param *strCurrentTime*

     \* @param *strLocation*

     \* @param *intType*

     \*/

    public abstract void visualize(String *strCurrentTime*, String *strLocation*, int *intType*);

    }

## Board的可视化：外部API的复用

首先获得PlanningEntryCollection的PlanningEntry进行遍历，获得时间该航班的（到达/起飞）时间，与当前时间进行比对，若差距在预设的范围内（HOURS\_RANGE=1）便将该PlanningEntry的信息记录到Vector上，再将该Vector加入二维Vector上，该二维Vector用于生成JTable。

Board.makeTable()中新建JTable，将信息输入表格，再将表格加入JFrame，委派JFrame进行可视化。

    @Override

    public void visualize(String *strCurrentTime*, String *strLocation*, int *intType*) {

        // iterator

        Iterator<PlanningEntry<Resource>> iterator = super.iterator();

        // new 2D-vector

        Vector<Vector<?>> vData = new Vector<>();

        // new titles

        Vector<String> vName = new Vector<>();

        String[] columnsNames = new String[] { "Time", "Entry Number", "Origin", "", "Terminal", "State" };

        for (String name : columnsNames)

            vName.add(name);

        while (iterator.hasNext()) {

            FlightSchedule<Resource> planningEntry = (FlightSchedule<Resource>) iterator.next();

            // if the location isn't chosen, then the board be as all airports'

            if (!strLocation.isEmpty()) {

                if (intType == FlightBoard.ARRIVAL) {

                    if (!planningEntry.getLocationTerminal().toLowerCase().equals(strLocation.toLowerCase()))

                        continue;

                } else {

                    if (!planningEntry.getLocationOrigin().toLowerCase().equals(strLocation.toLowerCase()))

                        continue;

                }

            }

            // time

            LocalDateTime currentTime = LocalDateTime.parse(strCurrentTime,

                    DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm"));

            LocalDateTime scheduleTime = intType == FlightBoard.ARRIVAL ? planningEntry.getTimeArrival()

                    : planningEntry.getTimeLeaving();

            // check time in range

            if (scheduleTime.isBefore(currentTime.plusHours(HOURS\_RANGE))

                    && scheduleTime.isAfter(currentTime.minusHours(HOURS\_RANGE))) {

                // get information

                String strScheduleTime = scheduleTime.toString().substring(11);

                String planningEntryNumber = planningEntry.getPlanningEntryNumber();

                String locationOrigin = planningEntry.getLocationOrigin();

                String locationTerminal = planningEntry.getLocationTerminal();

                String state = planningEntry.getState().getStrState();

                // load in 1D vector

                Vector<String> vRow = new Vector<>();

                vRow.add(strScheduleTime);

                vRow.add(planningEntryNumber);

                vRow.add(locationOrigin);

                vRow.add("-->");

                vRow.add(locationTerminal);

                vRow.add(state);

                // add in 2D-vector

                vData.add((Vector<?>) vRow.clone());

            }

        }

        // visualization (extends from Board.maketable)

        makeTable(vData, vName, intType == ARRIVAL ? "Arrival" : "Leaving");

    }

此外，我还添加了可视化所有entry的功能：

    @Override

    public void showEntries(Resource *r*) {

        Iterator<PlanningEntry<Resource>> iterator = super.iterator();

        Vector<Vector<?>> vData = new Vector<>();

        Vector<String> vName = new Vector<>();

        String[] columnsNames = new String[] { "Time", "Entry Number", "Origin", "", "Terminal", "State" };

        for (String name : columnsNames)

            vName.add(name);

        while (iterator.hasNext()) {

            FlightSchedule<Resource> planningEntry = (FlightSchedule<Resource>) iterator.next();

            if (planningEntry.getResource() != null && !planningEntry.getResource().equals(r))

                continue;

            String strScheduleTime = planningEntry.getTimeLeaving() + " - " + planningEntry.getTimeArrival();

            String planningEntryNumber = planningEntry.getPlanningEntryNumber();

            String locationOrigin = planningEntry.getLocationOrigin();

            String locationTerminal = planningEntry.getLocationTerminal();

            String state = planningEntry.getState().getStrState();

            Vector<String> vRow = new Vector<>();

            vRow.add(strScheduleTime);

            vRow.add(planningEntryNumber);

            vRow.add(locationOrigin);

            vRow.add("-->");

            vRow.add(locationTerminal);

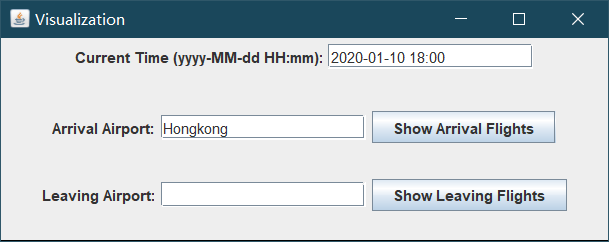
            vRow.add(state);

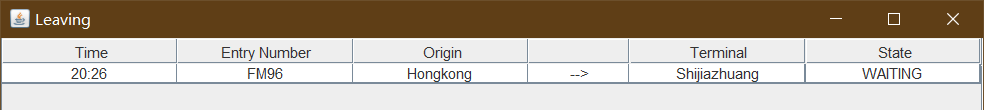
            vData.add((Vector<?>) vRow.clone());

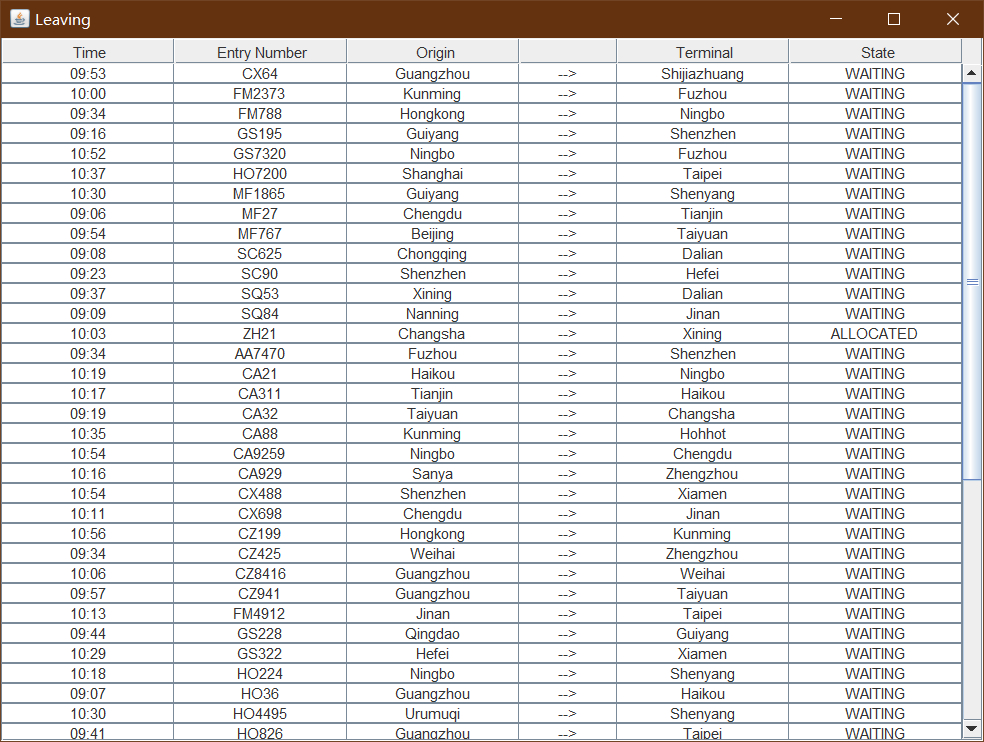
        }

        super.makeTable(vData, vName, "Entries");

可视化效果：







## PlanningEntryCollection的设计

该ADT是PlanningEntry的集合类。该集合类应该能够存储所有计划项、所有位置和可用资源，以及作为一个“管理者”的身份来操作这些成员变量。将PlanningEntryCollection作为抽象类，定义实现类的功能：

/\*\*

 \* planning entry collection is used to:

 \* manage resource, locations;

 \* generate / cancel / allocate / start / block / finish a planning entry;

 \* ask the current state

 \* search the conflict in the set of planning entry ( location / resource )

 \* present all the plan that one chosen resource has been used (Waiting, Running, Ended)

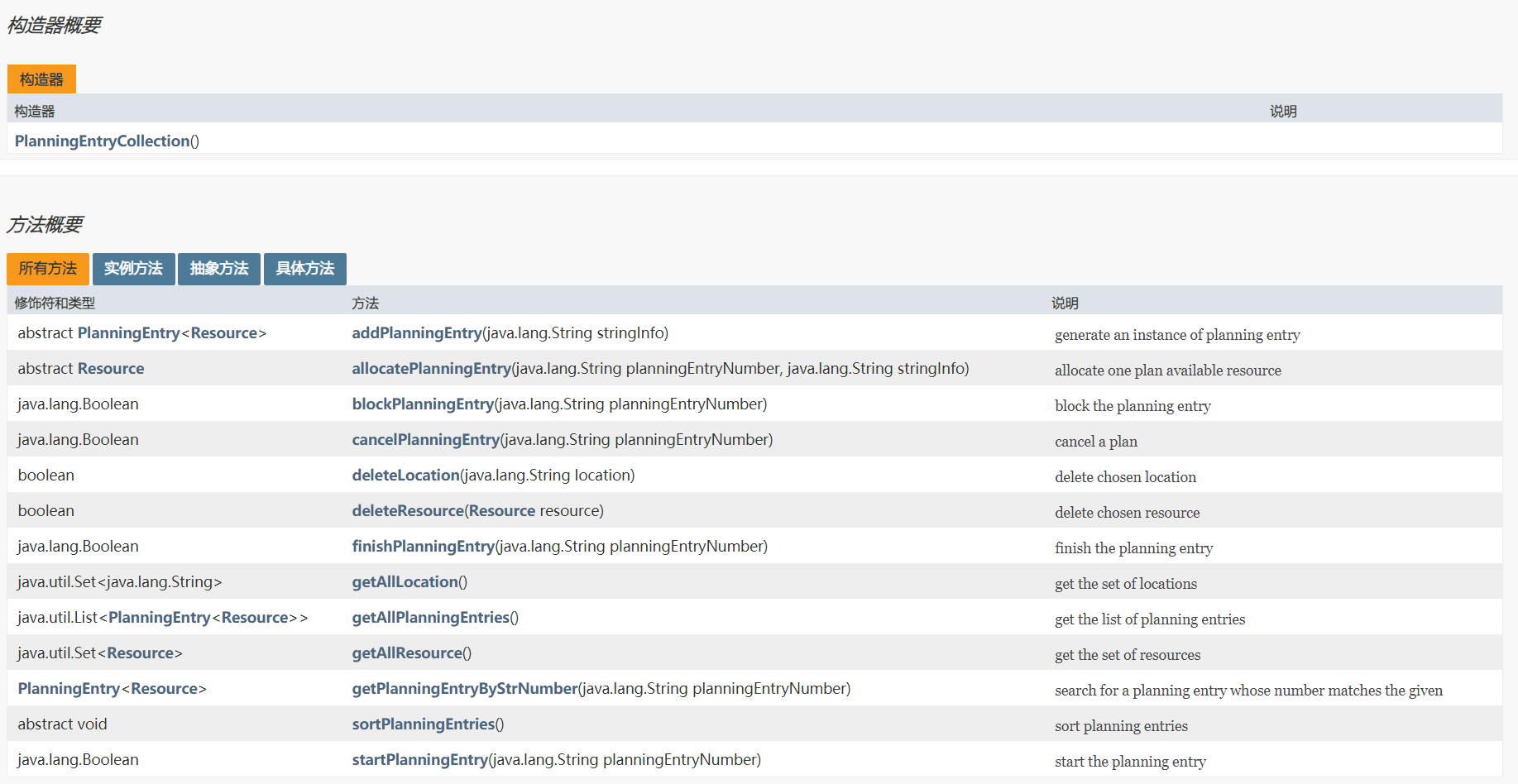
 \* show the board

 \*/

主要有6个功能：

1. 管理资源和位置；
2. 操作一个计划项：新建、取消、分配资源、开启、暂停、结束；
3. 查询计划项当前状态；
4. 查找冲突
5. 获得所有计划项
6. 打印信息板





在PlanningEntryCollection中实现了一些共性方法，并定义了一些抽象方法。不同子类实现有差异的方法有：添加计划项、分配资源、排序计划项；其余为相同的实现。

开始、取消、暂停、结束一个计划项这4种操作利用外观模式，将单个计划项中的成员方法进行封装。首先在所有计划项中找到该计划项，若找到则再进行对应的操作。以cancelPlanningEntry()为例：

    /\*\*

     \* cancel a plan

     \* @param *planningEntryNumber*

     \* @return true if cancelled successfully

     \*/

    public Boolean cancelPlanningEntry(String *planningEntryNumber*) {

        PlanningEntry<Resource> planningEntry = this.getPlanningEntryByStrNumber(planningEntryNumber);

        return planningEntry == null ? false : planningEntry.cancel();

    }

此外，共性方法还有计划项、资源、位置的Getter，以及删除单个资源和位置的方法。

接下来是有差异的方法。首先是新增一个计划项，以Flight Schedule为例。首先addPlanningEntry()有一个重载方法，可以将已经提取好的参数直接输入并新建：

    /\*\*

     \* generate a planning entry by given params

     \* @param *planningEntryNumber*

     \* @param *departureAirport*

     \* @param *arrivalAirport*

     \* @param *departureTime*

     \* @param *arrivalTime*

     \* @return the flight schedule

     \*/

    public FlightSchedule<Resource> addPlanningEntry(String *planningEntryNumber*, String *departureAirport*,

            String *arrivalAirport*, String *departureTime*, String *arrivalTime*) {

        Location location = new Location(departureAirport, arrivalAirport);

        TimeSlot timeSlot = new TimeSlot(Arrays.asList(departureTime, arrivalTime),

                Arrays.asList(departureTime, arrivalTime));

        this.collectionLocation.addAll(location.getLocations());

        PlanningEntry<Resource> flightSchedule = PlanningEntry.newPlanningEntryOfFlightSchedule(location, timeSlot,

                planningEntryNumber);

        this.planningEntries.add(flightSchedule);

        return (FlightSchedule<Resource>) flightSchedule;

    }

通过重载的方法，提供两种不同的新建方式，方便在之后的GUI客户端新建计划项的操作。接下来实现在抽象类中定义的方法，主要是要通过正则表达式提取所需的要素，并调用上述方法。用Pattern对象定义模式，用Matcher对象进行匹配；若匹配成功，则用group方法提取参数输入上述方法。

    @Override

    public FlightSchedule<Resource> addPlanningEntry(String *stringInfo*) {

        Pattern pattern = Pattern.compile(

                "Flight:(.\*?),(.\*?)\n\\{\nDepartureAirport:(.\*?)\nArrivalAirport:(.\*?)\nDepatureTime:(.\*?)\nArrivalTime:(.\*?)\nPlane:(.\*?)\n\\{\nType:(.\*?)\nSeats:(.\*?)\nAge:(.\*?)\n\\}\n\\}\n");

        Matcher matcher = pattern.matcher(stringInfo);

        if (!matcher.find())

            return null;

        String planningEntryNumber = matcher.group(2);

        String departureAirport = matcher.group(3);

        String arrivalAirport = matcher.group(4);

        String departureTime = matcher.group(5);

        String arrivalTime = matcher.group(6);

        return this.addPlanningEntry(planningEntryNumber, departureAirport, arrivalAirport, departureTime, arrivalTime);

    }

分配资源也是有差异的方法，与新建计划项的方法类似。我也新增了几个重载方法，模块化、也方便调用。有两种Pattern，对应两种输入模式。

    @Override

    public Resource allocatePlanningEntry(String *planningEntryNumber*, String *stringInfo*) {

        if (this.getPlanningEntryByStrNumber(planningEntryNumber) == null)

            return null;

        Pattern pattern1 = Pattern.compile(

                "Flight:(.\*?),(.\*?)\n\\{\nDepartureAirport:(.\*?)\nArrivalAirport:(.\*?)\nDepatureTime:(.\*?)\nArrivalTime:(.\*?)\nPlane:(.\*?)\n\\{\nType:(.\*?)\nSeats:(.\*?)\nAge:(.\*?)\n\\}\n\\}\n");

        Pattern pattern2 = Pattern.compile("Plane:(.\*?)\n\\{\nType:(.\*?)\nSeats:(.\*?)\nAge:(.\*?)\n\\}\n");

        Matcher matcher = pattern1.matcher(stringInfo);

        if (!matcher.find()) {

            matcher = pattern2.matcher(stringInfo);

            if (!matcher.find())

                return null;

        }

        String number = matcher.group(7);

        String strType = matcher.group(8);

        int intSeats = Integer.valueOf(matcher.group(9));

        double age = Double.valueOf(matcher.group(10));

        return this.allocateResource(planningEntryNumber, number, strType, intSeats, age);

    }

最后是按时间顺序排序计划项sortPlanningEntries()。首先定义一个Comparator，并重写其compare方法，然后调用Collections.sort()方法进行排序。在Flight Schedule中的Comparator对象，compare方法通过获取时间进行比较，具体如下：

Comparator<PlanningEntry<Resource>> comparator = new Comparator<PlanningEntry<Resource>>() {

            @Override

            public int compare(PlanningEntry<Resource> *o1*, PlanningEntry<Resource> *o2*) {

                return ((FlightSchedule<Resource>) o1).getTimeLeaving()

                        .isBefore(((FlightSchedule<Resource>) o2).getTimeArrival()) ? -1 : 1;

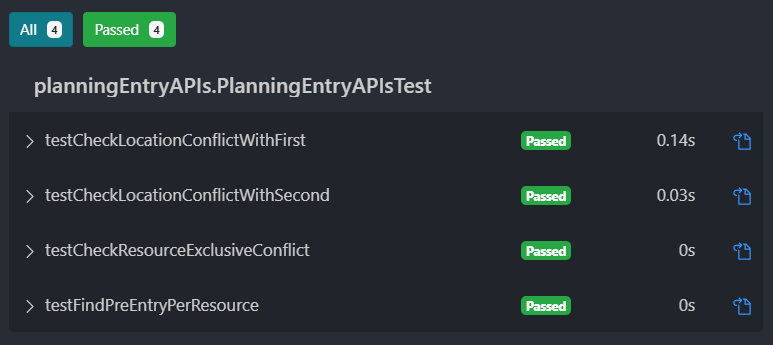
            }

        };

在各个子类型中，还有依据计划项编号获取计划项等封装方法，减轻客户端操作集合类的难度。

## 可复用API设计及Façade设计模式

Junit测试



### 检测一组计划项之间是否存在位置独占冲突

要检测一组计划项之间是否存在位置冲突，主要应该检测每一个位置的若干计划项是否有时间冲突。首先要保存下每个位置的所有计划项，我使用的是一个Map、键为位置String、值为使用该位置的所有计划项的List。

Map<String, List<ActivityCalendar<Resource>>> locationMap = new HashMap<>();

接下来，遍历所有计划项。对于每个计划项：若该计划项的位置未被加入Map的键集合，则加入并将值赋值为仅有该计划项的List；否则，将该计划项加入原有的值的List中，并考察List中是否有冲突，最后更新该值。

if (locationMap.keySet().contains(strLocation)) {

                List<ActivityCalendar<Resource>> calendars = new ArrayList<>();

                calendars.addAll(locationMap.get(strLocation));

                calendars.add(activityCalendar);

                ……

                locationMap.remove(strLocation);

                locationMap.put(strLocation, calendars);

            } else {

                locationMap.put(strLocation, new ArrayList<ActivityCalendar<Resource>>() {

                    private static final long serialVersionUID = 1L;

                    {

                        add(activityCalendar);

                    }

                });

            }

考察List是否有冲突，要遍历任意两个计划项，是否存在这时间重叠。对于任意两个不同计划项c1、c2，分别获取它们的起始时间和结束时间，进行比较：若一方的起始时间早于另一方的结束时间且结束时间晚于起始时间，则认为冲突。

LocalDateTime t1b = c1.getBeginningTime(), t1e = c1.getEndingTime();

LocalDateTime t2b = c2.getBeginningTime(), t2e = c2.getEndingTime();

if ((t1b.isBefore(t2e) || t1b.isEqual(t2e)) && (t1e.isAfter(t2b) || t2e.isEqual(t2b)))

return true;

返回true表示冲突，返回false表示无冲突。

流程图：



### 检测一组计划项之间是否存在资源独占冲突

该操作与上一方法类似，代码级别复用即可。

要检测一组计划项之间是否存在资源冲突，主要应该检测使用每一个资源的若干计划项是否有时间冲突。首先要保存下每个位置的所有计划项，我使用的是一个Map、键为资源、值为使用该位置的所有计划项的List。

接下来，遍历所有计划项。对于每个计划项：若该计划项的资源未被加入Map的键集合，则加入并将值赋值为仅有该计划项的List；否则，将该计划项加入原有的值的List中，并考察List中是否有冲突，最后更新该值。

考察List是否有冲突，要遍历任意两个计划项，是否存在这时间重叠。对于任意两个不同计划项p1、p2，分别获取它们的起始时间和结束时间，进行比较：若一方的起始时间早于另一方的结束时间且结束时间晚于起始时间，则认为冲突。

返回true表示冲突，返回false表示无冲突。

流程图：



### 提取面向特定资源的前序计划项

提取特定资源的前序计划项，是要搜索使用同一资源、计划时间在选定计划项之前且最晚（与选定计划项时间最近）的计划项。该方法类似与在一组数据中选取符合条件的最大值。整体思路就是遍历、筛选、比较

首先，初始化“最晚时间”和“前序计划项”；

    LocalDateTime latestDateTime = LocalDateTime.MIN;

    PlanningEntry<Resource> prePlanningEntry = null;

然后，遍历所有计划项，选出使用相同资源的计划项（在迭代时比较资源是否相同来进行筛选）：

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

            if (entries.get(i).getResource().equals(e.getResource())) {

                ……

        }

在迭代中比较，若符合筛选条件，且比原最晚时间更晚，则更新：

LocalDateTime endingTime = planningEntry.getTimeArrival();

if (endingTime.isAfter(latestDateTime)

&& endingTime.isBefore(e.getTimeLeaving())) {

     latestDateTime = endingTime;

     prePlanningEntry = planningEntry;

}

最后返回prePlanningEntry即可。

流程图：



## 设计模式应用

请分小节介绍每种设计模式在你的ADT和应用设计中的具体应用。

### Factory Method

设置3个PlanningEntry接口的工厂方法，分别新建1种计划项子类型。以Flight Schedule为例，需要输入Location、TimeSlot和计划项编号3个参数，返回一个FlightSchedule计划项类型：

    /\*\*

     \* a factory method for generating an instance of Flight Schedule

     \* @param *<R>*

     \* @param *location*

     \* @param *timeSlot*

     \* @param *planningEntryNumber*

     \* @return a empty instance of planning entry of  flight schedule

     \*/

    public static <R> FlightSchedule<R> newPlanningEntryOfFlightSchedule(Location *location*, TimeSlot *timeSlot*, String *planningEntryNumber*) {

        return new FlightSchedule<R>(location, timeSlot, planningEntryNumber);

    }

### Iterator

在Collection中，用一个List存储所有的计划项；因此在Board中，迭代器的方法该存储计划项的list.iterator()。

public Iterator<PlanningEntry<Resource>> iterator() {

    return planningEntryCollection.getAllPlanningEntries().iterator();

}

此外，在需要比较PlanningEntry时，新建comparator对象，重写compare方法。

Comparator<PlanningEntry<Resource>> comparator = new Comparator<PlanningEntry<Resource>>() {

      @Override

      public int compare(PlanningEntry<Resource> *o1*, PlanningEntry<Resource> *o2*) {

           return ((FlightSchedule<Resource>) o1).getTimeLeaving()

                  .isBefore(((FlightSchedule<Resource>) o2).getTimeArrival()) ? -1 : 1;

      }

};

在FlightBoard.visualize()方法中，使用该迭代器生成方法：

Iterator<PlanningEntry<Resource>> iterator = super.iterator();

### Strategy

在抽象类PlanningEntryAPIs中设置抽象方法：

    /\*\*

     \* For Activity Calendar

     \* check locations of planning entry in entries if they are conflicted

     \* @param *entries*

     \* @return true if there are locations conflict

     \*/

    public abstract boolean checkLocationConflict(List<PlanningEntry<Resource>> *entries*);

分别用若干子类来实现该方法。我使用了PlanningEntryAPIsFirst和PlanningEntryAPIsSecond两个类分别实现了检查位置冲突的方法。

public class PlanningEntryAPIsFirst extends PlanningEntryAPIs {

    @Override

    public boolean checkLocationConflict(List<PlanningEntry<Resource>> *entries*) {

        ……

}

public class PlanningEntryAPIsSecond extends PlanningEntryAPIs {

    @Override

    public boolean checkLocationConflict(List<PlanningEntry<Resource>> *entries*) {

        ……

}

在调用该方法时，首先新建两种对象之一，再调用其方法。注意：静态方法不能被重写。

boolean flag = (new PlanningEntryAPIsFirst()).checkLocationConflict(flightScheduleCollection.getAllPlanningEntries());

## 应用设计与开发

### 航班应用

#### 初始化数据

在界面显示之前，App会先将指定文件的数据读入，并且存储到App静态变量flightScheduleCollection中，以便之后功能的使用。

根据语法读入的既定规则，将每13行（航班数据是13行为单位，用静态常量存储）作为一个数据单元，每读入13行字符串则新建一个计划项。

    /\*\*

     \* read file and add planning entries in txt

     \* @param *strFile*

     \* @throws Exception

     \*/

    public static void readFile(String *strFile*) throws Exception {

        BufferedReader bReader = new BufferedReader(new FileReader(new File(strFile)));

        String line = "";

        int cntLine = 0;

        StringBuilder stringInfo = new StringBuilder("");

        while ((line = bReader.readLine()) != null) {

            if (line.equals(""))

                continue;

            stringInfo.append(line + "\n");

            cntLine++;

            if (cntLine % INPUT\_ROWS\_PER\_UNIT == 0) {

                FlightSchedule<Resource> flightSchedule = flightScheduleCollection

                        .addPlanningEntry(stringInfo.toString());

                if (flightSchedule != null)

                    flightScheduleCollection.allocatePlanningEntry(flightSchedule.getPlanningEntryNumber(),

                            stringInfo.toString());

                stringInfo = new StringBuilder("");

            }

        }

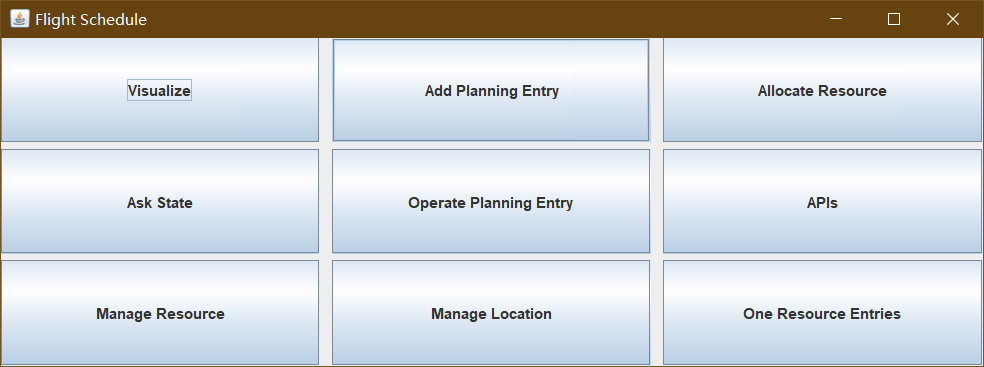
        bReader.close();

        // flightScheduleCollection.sortPlanningEntries();

    }

#### 起始界面

该界面的框架基于用网格布局，包含了9类要求实现功能，每类功能在点击之后会新建窗口，进行交互；若有多项功能合并为一个按钮，则还会在点开后进行选择。



此后的JFrame也大多于此设计类似。

JFrame mainFrame = new JFrame("Flight Schedule");

mainFrame.setDefaultCloseOperation(JFrame.DISPOSE\_ON\_CLOSE);

mainFrame.setLayout(new GridLayout(3, 3, 10, 5));

mainFrame.setVisible(true);

mainFrame.setSize(800, 300);

#### 可视化

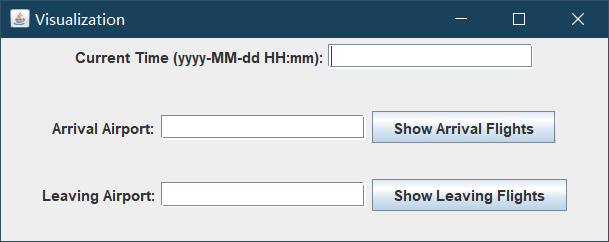
首先在主框架中新建一个按钮并命名，再添加动作Listener，若按下该按键，则启动专属于可视化的窗口。

JButton visualizeButton = new JButton("Visualize");

mainFrame.add(visualizeButton);

visualizeButton.addActionListener((e) -> visualization());

在可视化的窗口中，添加输入时间的流面板，以及输入到达/起飞机场的文本框。



该窗口的两个按键会调用Board中的功能，具体在3.8节中阐述。

#### 新建计划项

新建该面板，提供输入各项参数的文本框，并在按下按键后，将信息整理成与文本数据相同格式的数据字符串，添加计划项。

航班有5个信息，则需要5个输入信息的流面板。批量化处理

String[] panelsName = new String[] { "Planning Entry Number:", "Departure Airport:", "Arrival Airport:", "Departure Time (yyyy-MM-dd HH:mm):", "Arrival Time (yyyy-MM-dd HH:mm):" };

        List<JPanel> panelsList = new ArrayList<>();

        List<JTextField> textList = new ArrayList<>();

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

            JPanel newPanel = new JPanel();

            panelsList.add(newPanel);

            newPanel.setLayout(new FlowLayout());

            newPanel.add(new JLabel(panelsName[i]));

            JTextField newText = new JTextField(LINE\_WIDTH);

            textList.add(newText);

            newPanel.add(newText);

            addPlanningEntryFrame.add(newPanel);

        }

在Action Listener中，若按钮被按下，则读入字符串，返回新建的结果，弹出提示信息（成功/失败）。

        enterButton.addActionListener((e) -> {

            List<String> gotString = new ArrayList<>();

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

                gotString.add(textList.get(i).getText());

            }

            flightScheduleCollection.addPlanningEntry(gotString.get(0), gotString.get(1), gotString.get(2),

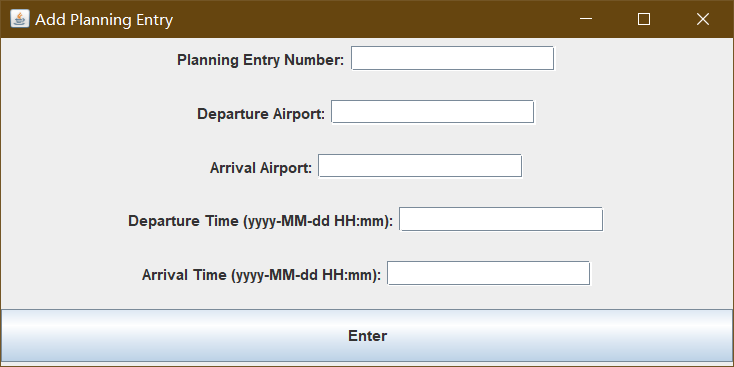
                    gotString.get(3), gotString.get(4));

            addPlanningEntryFrame.dispose();

            JOptionPane.showMessageDialog(addPlanningEntryFrame, "Successfully", "Add Planning Entry", JOptionPane.PLAIN\_MESSAGE);

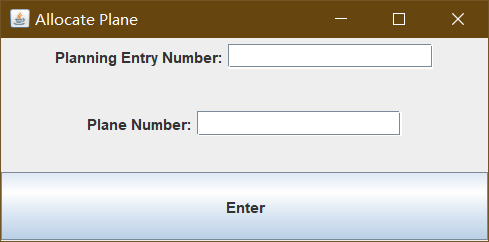
            addPlanningEntryFrame.dispose();

        });



#### 分配资源

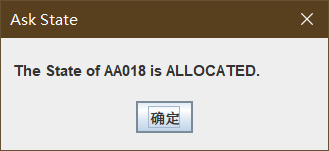
读入方法与上述类似：读入资源信息，通过按钮的动作进行操作的启动。



#### 询问状态

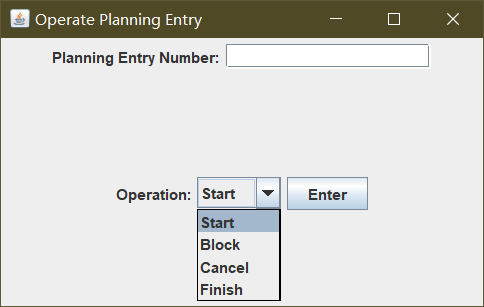
读入方法与上述类似：读入计划项编号信息，通过按钮启动，再通过提示框来显示状态。





#### 计划项操作（启动、取消、暂停、结束）

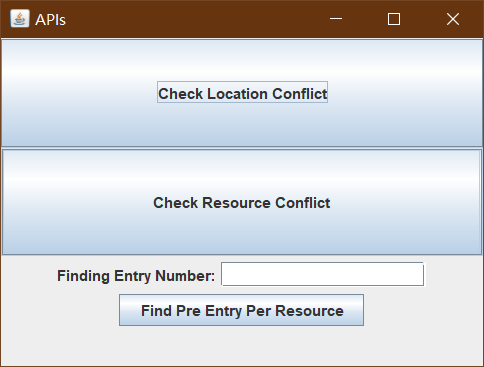
通过一个CombBox给出操作的选项，对指定计划项进行操作。





#### API：检查冲突、查找前置计划项

给出3个流面板，前两个是检查位置/资源按键，后一个是输入计划项编号查找前置项。



#### 管理（增加/删除）资源

在读入用户输入的资源，并进行增加/删除前，要给出所有资源的信息，再通过必要信息进行操作。

        String resourcesStrings = "";

        Set<Resource> allResource = flightScheduleCollection.getAllResource();

        List<Resource> allResourceList = new ArrayList<>();

        int i = 0;

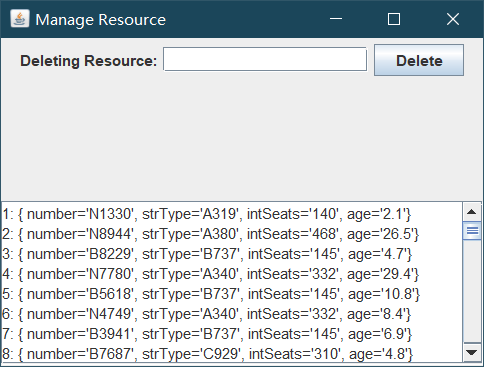
        for (Resource plane : allResource) {

            i++;

            resourcesStrings += String.valueOf(i) + ": " + ((Plane) plane).toString() + "\n";

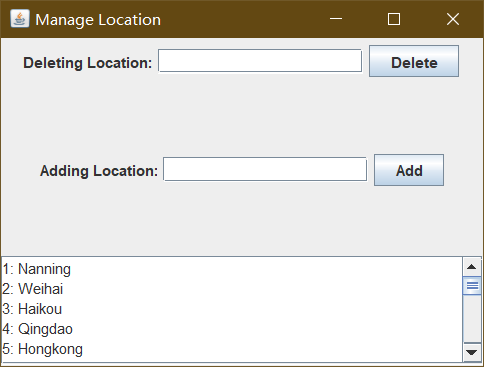
            allResourceList.add(plane);

        }



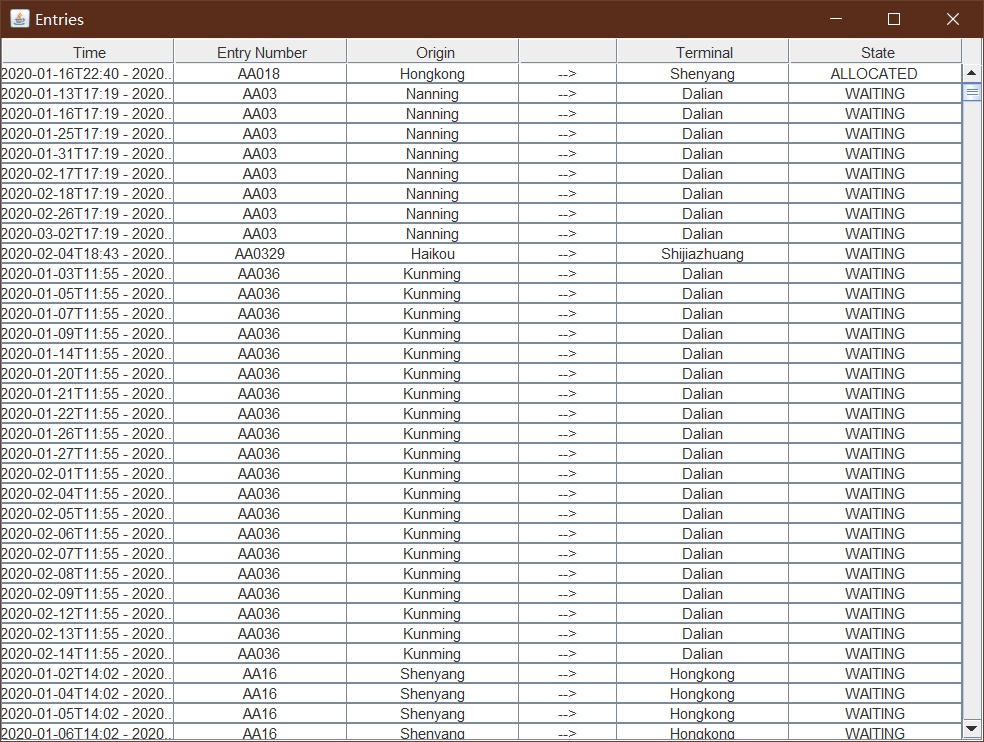
#### 管理（增加/删除）位置

与上述类似。



#### 同一资源的计划项

搜索同一编号的资源的计划项汇集成表格，委派Board进行显示。



### 高铁应用

与上述类似。差别主要在于表格的显示需要多个站点。

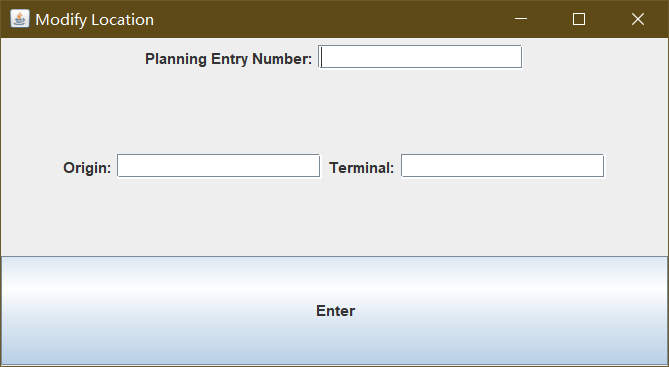


### 学习活动应用

与上述类似。



新增修改地点的操作。



## 基于语法的数据读入

#### 航班

依据给定数据的格式，可以设计正则表达式，并保存在Pattern对象中。

Pattern pattern = Pattern.compile("Flight:(.\*?),(.\*?)\n\\{\nDepartureAirport:(.\*?)\nArrivalAirport:(.\*?)\nDepatureTime:(.\*?)\nArrivalTime:(.\*?)\nPlane:(.\*?)\n\\{\nType:(.\*?)\nSeats:(.\*?)\nAge:(.\*?)\n\\}\n\\}\n");

每读入一个单位的数据就进行匹配，若Matcher对象有找到，则匹配成功。

Matcher matcher = pattern.matcher(stringInfo);

if (!matcher.find()) return null;

根据数据格式，进行提取。

        String planningEntryNumber = matcher.group(2);

        String departureAirport = matcher.group(3);

        String arrivalAirport = matcher.group(4);

        String departureTime = matcher.group(5);

        String arrivalTime = matcher.group(6);

最后基于这些参数新建计划项。

#### 高铁

高铁的数据格式和样例均没有给定，于是我需要自行建立。该设计参考航班数据的格式。

首先预设一些高铁的站点。

String[] cities = new String[] { "Harbin", "Beijing", "Shanghai", "Shenzhen", "Guangzhou" };

对于每一个新建的计划项，随机站点数，和每个站点的信息；还有随机高铁信息（或固定信息选调）。例如，一趟火车的整体信息：

String planningDate = "2020-01-01";

String planningNumber = String.valueOf(i);

String trainNumber = String.valueOf(Math.random());

String trainType = "Business";

String trainCapacity = String.valueOf(100);

站点的信息可以随机为：

String stations = "";

for (int j = 1; j <= M; j++) {

stations += cities[j - 1] + " " + String.format("2020-01-01 10:%d", j \* 10) + " " + String.format("2020-01-01 10:%d", j \* 10) + "\n";

}

最后，参考给定数据格式，合并这些信息，保存到文件中：

content += String.format("Train:%s,%s\n{\n%sTrain:%s\n{\nTrainType:%s\nTrainCapacity:%s\n}\n}\n", planningDate, planningNumber, stations, trainNumber, trainType, trainCapacity);

打开文件保存：

try {

            File file = new File("data/TrainSchedule/TrainSchedule\_1.txt");

            if (!file.exists()) {

                file.createNewFile();

            }

            FileWriter fileWriter = new FileWriter(file.getName(), true);

            fileWriter.write(content);

            fileWriter.flush();

            fileWriter.close();

        } catch (IOException *e*) {

            e.printStackTrace();

        }

数据样例如下：

|  |
| --- |
| Train:2020-01-01,1 |
| { |
| Harbin 2020-01-01 10:10 2020-01-01 10:10 |
| Beijing 2020-01-01 10:20 2020-01-01 10:20 |
| Shanghai 2020-01-01 10:30 2020-01-01 10:30 |
| Shenzhen 2020-01-01 10:40 2020-01-01 10:40 |
| Guangzhou 2020-01-01 10:50 2020-01-01 10:50 |
| Train:A0 |
| { |
| TrainType:Business |
| TrainCapacity:100 |
| } |
| } |

#### 学习活动

与高铁的数据设计类似，主要差别在单个地点。

在设计计划项和资源信息时（以4月课程为例）：

String planningDate = String.format("2020-04-0%d", i % 9 + 1);

String activityNumber = String.valueOf(i);

String room = "Zhengxin44";

String beginningTime = String.format("2020-04-0%d 10:00", i % 9 + 1);

String endingTime = String.format("2020-04-0%d 12:00", i % 9 + 1);

String docName = "Software Construction";

String publishDepartment = "HIT";

String publishDate = "2000-01-01";

按照指定格式加入数据：

content += String.format(

                    "Activity:%s,%s\n{\nRoom:%s\nBeginningTime:%s\nEndingTime:%s\nDocument:A4\n{\nDocName:%s\nPublishDepartment:%s\nPublishDate:%s\n}\n}\n",

                    planningDate, activityNumber, room, beginningTime, endingTime, docName, publishDepartment, publishDate);

数据样例：

|  |
| --- |
| Activity:2020-04-02,1 |
| { |
| Room:Zhengxin44 |
| BeginningTime:2020-04-02 10:00 |
| EndingTime:2020-04-02 12:00 |
| Document:A4 |
| { |
| DocName:Software Construction |
| PublishDepartment:HIT |
| PublishDate:2000-01-01 |
| } |
| } |

## 应对面临的新变化

评估之前的设计是否可应对变化、代价如何

如何修改设计以应对变化

### 变化1：航班

航班管理中变化在于从2个站变成3个站，根据之前的思路，在取3个站位置/资源时用不同的静态常量作为下标即可。

public static final int ORIGIN = 0, MID = 1, TERMINAL = 2;

因此，获取位置的方法可以更改为（type为上述常量之一）：

    public String getLocation(int *type*) {

        return super.getLocation().getLocations().get(type);

    }

获取资源同理。

所以，该变化可以通过继承父类，进行方法的增加/重载以达到变化的目的。总体代价不大，仅有3个Getter。

### 变化2：高铁

高铁的变化在于，若被分配资源后则不能被取消。这个变化可以通过重写cancel()方法，增加判断语句限制前置状态即可。

    @Override

    public Boolean cancel() {

        if (this.state.getState().equals(EntryStateEnum.ALLOCATED))

            return false;

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

    }

因此，其代价为重写一个方法，代价很小。

### 变化3：学习活动

学习活动的变化在于，学习活动也可以被临时暂停。这个变化可以通过增加可阻塞的活动名单。

EntryStateEnum.BlockAbleKeyWords.add("Activity");

该操作在构造方法中进行，代价极小。

## Git仓库结构

请在完成全部实验要求之后，利用Git log指令或Git图形化客户端或GitHub上项目仓库的Insight页面，给出你的仓库到目前为止的Object Graph，尤其是区分清楚314change分支和master分支所指向的位置。

# 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 |
| 2020-04-09 | 晚上 | 构建项目 | 完成 |
| 2020-04-10 | 下午 | 设计PlanningEntry框架 | 初步完成 |
| 2020-04-11 | 下午 | 进一步晚上PlanningEntry框架 | 基本完成 |
| 2020-04-11 | 晚上 | 取消Decorator模式 | 完成 |
| 2020-04-11 | 晚上 | 设计Location、Resource、TimeSlot，并精简计划项为单接口单抽象类 | 完成 |
| 2020-04-12 | 下午 | 完成整体框架 | 基本完成 |
| 2020-04-13 | 下午 | 补充方法Spec和实现部分简单方法 | 基本完成 |
| 2020-04-14 | 晚上 | 设计Location、Resource、TimeSlot、EntryState测试用例 | 完成 |
| 2020-04-14 | 下午 | 设计Collection | 初步完成 |
| 2020-04-15 | 下午 | 完成新增计划项、分配资源的操作 | 完成 |
| 2020-04-16 | 下午 | 设计Board类 | 完成 |
| 2020-04-17 | 下午 | 完成Collection并测试 | 完成 |
| 2020-04-18 | 晚上 | 设计App及其GUI界面 | 初步完成 |
| 2020-04-19 | 下午 | 迁移Flight设计到Train和Activity | 基本完成 |
| 2020-04-20 | 下午 | 完善App和修改ADT | 完成 |
| 2020-04-21 | 晚上 | 增加测试用例 | 完成 |
| 2020-04-22 | 下午 | 增加314change | 完成 |
| 2020-04-22 | 晚上 | 验收 | 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 |
| PlanningEntryAPIs静态方法无法在子类中重写 | 更改为成员方法，并且更改客户端使用方式 |
|  |  |
|  |  |

# 实验过程中收获的经验、教训、感想

## 实验过程中收获的经验和教训

## 针对以下方面的感受

1. 重新思考Lab2中的问题：面向ADT的编程和直接面向应用场景编程，你体会到二者有何差异？本实验设计的ADT在五个不同的应用场景下使用，你是否体会到复用的好处？
2. 重新思考Lab2中的问题：为ADT撰写复杂的specification, invariants, RI, AF，时刻注意ADT是否有rep exposure，这些工作的意义是什么？你是否愿意在以后的编程中坚持这么做？
3. 之前你将别人提供的API用于自己的程序开发中，本次实验你尝试着开发给别人使用的API，是否能够体会到其中的难处和乐趣？
4. 在编程中使用设计模式，增加了很多类，但在复用和可维护性方面带来了收益。你如何看待设计模式？
5. 你之前在使用其他软件时，应该体会过输入各种命令向系统发出指令。本次实验你开发了一个解析器，使用语法和正则表达式去解析输入文件并据此构造对象。你对语法驱动编程有何感受？
6. Lab1和Lab2的大部分工作都不是从0开始，而是基于他人给出的设计方案和初始代码。本次实验是你完全从0开始进行ADT的设计并用OOP实现，经过五周之后，你感觉“设计ADT”的难度主要体现在哪些地方？你是如何克服的？
7. “抽象”是计算机科学的核心概念之一，也是ADT和OOP的精髓所在。本实验的五个应用既不能完全抽象为同一个ADT，也不是完全个性化，如何利用“接口、抽象类、类”三层体系以及接口的组合、类的继承、设计模式等技术完成最大程度的抽象和复用，你有什么经验教训？
8. 关于本实验的工作量、难度、deadline。
9. 到目前为止你对《软件构造》课程的评价。