

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

**目录**

[1 实验目标概述 1](#_Toc37611518)

[2 实验环境配置 1](#_Toc37611519)

[3 实验过程 1](#_Toc37611520)

[3.1 待开发的三个应用场景 1](#_Toc37611521)

[3.2 面向可复用性和可维护性的设计：PlanningEntry<R> 1](#_Toc37611522)

[3.2.1 PlanningEntry<R>的共性操作 1](#_Toc37611523)

[3.2.2 局部共性特征的设计方案 2](#_Toc37611524)

[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） 2](#_Toc37611525)

[3.3 面向复用的设计：R 2](#_Toc37611526)

[3.4 面向复用的设计：Location 2](#_Toc37611527)

[3.5 面向复用的设计：Timeslot 2](#_Toc37611528)

[3.6 面向复用的设计：EntryState及State设计模式 2](#_Toc37611529)

[3.7 面向应用的设计：Board 2](#_Toc37611530)

[3.8 Board的可视化：外部API的复用 2](#_Toc37611531)

[3.9 可复用API设计及Façade设计模式 2](#_Toc37611532)

[3.9.1 检测一组计划项之间是否存在位置独占冲突 2](#_Toc37611533)

[3.9.2 检测一组计划项之间是否存在资源独占冲突 2](#_Toc37611534)

[3.9.3 提取面向特定资源的前序计划项 2](#_Toc37611535)

[3.10 设计模式应用 2](#_Toc37611536)

[3.10.1 Factory Method 3](#_Toc37611537)

[3.10.2 Iterator 3](#_Toc37611538)

[3.10.3 Strategy 3](#_Toc37611539)

[3.11 应用设计与开发 3](#_Toc37611540)

[3.11.1 航班应用 3](#_Toc37611541)

[3.11.2 高铁应用 3](#_Toc37611542)

[3.11.3 进程应用 3](#_Toc37611543)

[3.11.4 课表应用 3](#_Toc37611544)

[3.11.5 学习活动应用 3](#_Toc37611545)

[3.12 基于语法的数据读入 3](#_Toc37611546)

[3.13 应对面临的新变化 3](#_Toc37611547)

[3.13.1 变化1 3](#_Toc37611548)

[3.13.2 变化2 4](#_Toc37611549)

[3.13.3 变化3 4](#_Toc37611550)

[3.14 Git仓库结构 4](#_Toc37611551)

[4 实验进度记录 4](#_Toc37611552)

[5 实验过程中遇到的困难与解决途径 4](#_Toc37611553)

[6 实验过程中收获的经验、教训、感想 5](#_Toc37611554)

[6.1 实验过程中收获的经验和教训 5](#_Toc37611555)

[6.2 针对以下方面的感受 5](#_Toc37611556)

# 实验目标概述

根据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

此次实验环境与实验2无异。

我的GitHub Lab3仓库的URL地址：

[https://github.com/ComputerScienceHIT/Lab3-xxxxxxxxxx](https://github.com/ComputerScienceHIT/Lab3-1180100217)

# 实验过程

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

## 待开发的三个应用场景

列出你所选定的三个应用。

我选定的三个应用场景为：

* 航班管理（Flight Schedule）
* 高铁车次管理（Train Schedule）
* 大学课表管理（Course Schedule）

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。

共性：它们的位置和时间都可以提前设定。

差异：航班管理包含两个位置，高铁车次管理有多个位置，大学课程管理则只有一个位置。航班管理和高铁车次管理的位置设定后就不可变，而大学课程管理则可以再变。航班管理和课程管理的资源为单个，而车次管理的资源为多个，且车次管理可以阻塞。

## 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。

### PlanningEntry<R>的共性操作

首先所有的PlanningEntry<R>都可以获取基本信息，包括名称、状态、当前对应的时间对以及当前的位置对应的方法为getName、getState、getCurrentTimeslot和getCurrentLocation。同时我们需要计划项支持转移状态，即包含allocated, run, cancel, end方法来转移计划项的状态，以及开放setState接口来便于状态模式的实现。另外为了检测冲突，还需要实现conflictResourceWith和conflictLocationWith来检测资源和位置的冲突。

为此，我实现了抽象类CommonPlanningEntry<R>，实现了共同的getName、getState、setState以及通过状态模式实现了allocated, run, cancel, end方法，将这些方法委托给EntryState类。

1 @Override

2 public void allocated() throws IllegalStateTransitionException {

3 state.allocated();

4 }

5

6 @Override

7 public void run() throws IllegalStateTransitionException {

8 state.run();

9 }

10

11 @Override

12 public void cancel() throws IllegalStateTransitionException {

13 state.cancel();

14 }

15

16 @Override

17 public void end() throws IllegalStateTransitionException {

18 state.end();

19 }

### 局部共性特征的设计方案

#### 位置特性

对于只有一个位置的计划项，我设计了SingleLocationEntry接口。它包含getLocation方法和setLocation方法，用于获取和设置计划项的位置。对于这个接口，我设计了SingleLocationEntryImpl类来实现其基本功能，具体方法是包装了一个Location变量。

对于包含两个位置的计划项（一个起点，一个终点），我设计了PairLocationsEntry接口。它包含：getStartLocation和getTargetLocation方法，用于获取起点和终点；getLocations方法返回一个包含两个Location的List，依次为起点和终点；setStartLocaion和setTargetLocation方法来设置起点和终点；setLocations来同时设置起点和终点，其中List的大小为二，依次为起点和终点。对于这个接口，我设计了PairLocationsEntryImpl类来实现其基本功能，具体方法为包装两个Location变量，分别表示起点和终点。

对于包含更多的位置有序的计划项，我设计了MultipleLocationsEntry接口。它包含了getLocations和setLocations方法，用于获取和修改位置。对于这个接口，我设计了MultipleLocationsEntry类来实现其基本功能，具体实现为使用一个List<Location>，同时在操作时要防御性复制，防止客户端通过其他方法间接修改List。

由于部分计划项设置位置后不可修改，所以添加了两个修饰器：修饰PairLocationsEntry的PairLocationsEntryUnmodifiableDecorator和 修饰MultipleLocationsEntry的MultipleLocationsEntryUnmodifiableDecorator。这两个修饰器将所有的setters重写，改为抛出UnsupportedOperationException异常。

1 // MultipleLocationsEntryUnmodifiableDecorator

2 @Override

3 public void setLocations(List<Location> locations) {

4 throw new UnsupportedOperationException();

5 }

1 // PairLocationsEntryUnmodifiableDecorator

2 @Override

3 public void setStartLocation(Location startLocation) {

4 throw new UnsupportedOperationException();

5 }

6

7 @Override

8 public void setTargetLocation(Location targetLocation) {

9 throw new UnsupportedOperationException();

10 }

11

12 @Override

13 public void setLocations(List<Location> locations) {

14 throw new UnsupportedOperationException();

15 }

#### 资源特性

对于具有单个可区分资源的计划项，我设计了SingleResourceEntry<R>接口，其中R代表资源的类型。它包含getResource和setResource方法，分别用于获取和更改资源。对于这个接口，我设计了SingleResourceEntryImpl<R>类来实现其基本功能，具体方法是包装了一个R变量。

对于具有多个带次序的可区分资源的计划项，我设计了MultipleSortedResourceEntry<R>接口，同样R代表资源的类型。它包含getResources和setResources方法，分别用于获取和更改资源。对于这个接口，我设计了MultipleSortedResourceEntryImpl<R>类来实现其基本功能，具体实现为包装一个List<R>变量。

#### 起止时间对特性

对于只有一个起止时间对的计划项，我设计了SingleTimeslotEntry接口。它包含了getTimeslot和setTimeslot方法，分别用于获取和更改起止时间对。对于这个接口，我设计了SingleTimeslotEntryImpl类来实现其基本功能，具体方法是包装了一个Timeslot变量。

对于包含一组起止时间对且可阻塞的计划项，我设计了MultipleTimeslotEntry和BlockableEntry接口。MultipleTimeslotEntry接口包含了getTimeslots和setTimeslots方法，分别用于获取和更改起止时间对，而BlockableEntry接口则包含了block接口，用于将计划项阻塞（无单独实现的类）。对于MultipleTimeslotEntry接口，我设计了MultipleTimeslotEntryImpl类来实现其基本功能，具体方法是包装了一个List<Timeslot>变量。

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

#### 航班管理（Flight Schedule）

对于航班管理，设计一个对应的包——flight来管理所有相关的类。

对于航班计划项，我设计了接口FlightPlanningEntry。由于航班计划项包含一个起点和一个终点，位置设定后不可更改，具有单个可区分资源（飞机），具有一个起止时间对且可以提前设置，所以FlightPlanningEntry继承了PlanningEntry<Plane>， PairLocationsEntry， SingleResourceEntry<Plane>和 SingleTimeslotEntry接口。

为了实现FlightPlanningEntry接口，我设计了一个FlightEntry类。它继承了CommonPlanningEntry<Plane>类来实现复用。FlightEntry的抽象函数为AF(e) = {timeslot, startLocation, targetLocation, plane}，表示不变量为变量不是null且所有的locations都是可以共享的。

在此次实验中我使用了方案5（CRP），通过接口组合实现局部共性特征的复用，利用委托的方式用PairLocationsEntryImpl实现PairLocationsEntry接口的功能，用SingleResourceEntryImpl实现SingleResourceEntry接口的功能，用SingleTimeslotEntryImpl实现SingleTimeslotEntry接口的功能。由于位置设定后不可修改，所以在设置好位置后需要用PairLocationsEntryUnmodifiableDecrator修饰。

1 public FlightEntry(String name, Timeslot timeslot, Location startLocation, Location targetLocation) {

2 this.name = name;

3 state = new WaitingState(this);

4

5 pairLocationsEntryImpl =

6 new PairLocationsEntryUnmodifiableDecorator(new PairLocationsEntryImpl(startLocation, targetLocation));

7 singleTimeslotEntryImpl = new SingleTimeslotEntryImpl(timeslot);

8 singleResourceEntryImpl = new SingleResourceEntryImpl<>();

9 checkRep();

10 }

而对于其自身的其他函数：

getCurrentTimeslot：由于只有一个时间对，所以只需要返回存储的时间对即可。

1 @Override

2 public Timeslot getCurrentTimeslot() {

3 return singleTimeslotEntryImpl.getTimeslot();

4 }

getCurrentLocation：判断当前计划项的状态，除非处于结束状态，那么飞机一定在起始位置。

1 @Override

2 public Location getCurrentLocation() {

3 return state.toString().contains("ENDED") ? getTargetLocation() : getStartLocation();

4 }

conflictLocationWith：所有机场可以共用，直接返回false。

conflictResourceWith：先判断特殊情况并处理，之后判断时间段是否交叉，资源是否相同。

1 @Override

2 public boolean conflictResourceWith(PlanningEntry<?> entry) {

3 if (!(entry instanceof FlightPlanningEntry))

4 return false;

5 if (singleResourceEntryImpl.getResource() == null)

6 return false;

7 FlightPlanningEntry anotherEntry = (FlightPlanningEntry)entry;

8 Timeslot anotherTimeslot = anotherEntry.getTimeslot();

9 if (!getTimeslot().intersect(anotherTimeslot))

10 return false;

11 return getResource().equals(anotherEntry.getResource());

12 }

#### 高铁车次管理（Train Schedule）

对于高铁车次管理，设计一个对应的包——train来管理所有相关的类。

对于航班计划项，我设计了接口TrainPlanningEntry。由于高铁车次计划项包含一个起点、一个终点以及一系列中间位置，位置设定后不可更改，具有多个带次序的可区分资源（一组前后连接的车厢，均有编号），具有一组起止时间（因为可以阻塞）且可以提前设置，所以TrainPlanningEntry继承了PlanningEntry<TrainCabin>， MultipleLocationsEntry， MultipleResourceEntry<TrainCabin>和 MultipleTimeslotEntry接口。由于其包含多个位置和时间对，所以需要添加四个新的方法：hasNextLocation、moveToNextLocation、hasNextTimeslot和moveToNextTimeslot来管理位置和时间对。

为了实现TraintPlanningEntry接口，我设计了一个TrainEntry类。它继承了CommonPlanningEntry<TrainCabin>类来实现复用。TrainEntry的抽象函数为AF(e) = {timeslots, locations, train carbins}，表示不变量为变量不是null,所有的locations都是可以共享的而且时间对的数目应当为位置数目减一。

对于TrainEntry的部分公共特性，我利用委托的方式用MultipleLocationsEntryImpl实现MultipleLocationsEntry接口的功能，用MultipleResourceEntryImpl实现MultipleResourceEntry接口的功能，用MultipleTimeslotEntryImpl实现MultipleTimeslotEntry接口的功能。由于TrainEntry的位置设置后同样不可修改，所以设置后需要用MultipleLocationsEntryUnmodifiableDecorator来修饰。

1 public TrainEntry(String name, List<Timeslot> timeslots, List<Location> locations) {

2 this.name = name;

3 state = new WaitingState(this);

4

5 timeslotIndex = 0;

6 locationIndex = 0;

7 segments = timeslots.size();

8 multipleLocationsEntryImpl = new MultipleLocationsEntryImpl();

9 multipleLocationsEntryImpl.setLocations(locations);

10 multipleLocationsEntryImpl = new MultipleLocationsEntryUnmodifiableDecorator(multipleLocationsEntryImpl);

11 multipleTimeslotEntryImpl = new MultipleTimeslotEntryImpl();

12 multipleTimeslotEntryImpl.setTimeslots(timeslots);

13 multipleSortedResourceEntryImpl = new MultipleSortedResourceEntryImpl<TrainCabin>();

14 checkRep();

15 }

而对于其自身的其他函数：

getCurrentTimeslot：添加一个变量timeslotIndex来管理当前到了第几个时间对，进而获得当前时间。

1 @Override

2 public Timeslot getCurrentTimeslot() {

3 return multipleTimeslotEntryImpl.getTimeslots().get(timeslotIndex);

4 }

hasNextTimeslot：判断timeslotIndex和时间对的数目之间的大小关系。为了提高效率，而位置不可变，位置数目和时间对数目存在关系，所以可以将时间对的数目存储在一个私有常量segments中。

1 @Override

2 public boolean hasNextTimeslot() {

3 return timeslotIndex + 1 < segments;

4 }

moveToNextTimeslot：自加timeslotIndex即可。

1 @Override

2 public void moveToNextTimeslot() {

3 ++timeslotIndex;

4 }

getCurrentLocation、hasNextLocation和moveToNextLocation：它们的实现与时间对的管理类似，设计一个locationIndex来管理：

1 @Override

2 public Location getCurrentLocation() {

3 return multipleLocationsEntryImpl.getLocations().get(locationIndex);

4 }

5

6 @Override

7 public boolean hasNextLocation() {

8 return locationIndex + 1 <= segments;

9 }

10

11 @Override

12 public void moveToNextLocation() {

13 ++locationIndex;

14 }

conflictLocationWith：由于火车站可以共享，所以直接返回false。

conflictResourceWith：首先处理所有特殊情况，之后判断资源之间是否有交叉，若有，则判断时间对是否有交叉。

1 @Override

2 public boolean conflictResourceWith(PlanningEntry<?> entry) {

3 if (!(entry instanceof TrainPlanningEntry))

4 return false;

5 if (multipleSortedResourceEntryImpl.getResources().isEmpty())

6 return false;

7

8 TrainPlanningEntry anotherEntry = (TrainPlanningEntry)entry;

9 List<Timeslot> timeslots = getTimeslots(), anotherTimeslots = anotherEntry.getTimeslots();

10 List<TrainCabin> trainCabins = getResources(), anotherTrainCabins = anotherEntry.getResources();

11 boolean found = false;

12 Set<TrainCabin> set = new HashSet<>();

13 for (TrainCabin trainCabin : trainCabins) {

14 set.add(trainCabin);

15 }

16 for (TrainCabin trainCabin : anotherTrainCabins) {

17 if (set.contains(trainCabin)) {

18 found = true;

19 break;

20 }

21 }

22 if (found) {

23 for (Timeslot timeslot : timeslots)

24 for (Timeslot timeslot2 : anotherTimeslots)

25 if (timeslot.intersect(timeslot2))

26 return true;

27 }

28 return false;

29 }

#### 大学课表管理（Course Schedule）

对于大学课程管理，设计一个对应的包——course来管理相关的类。

对于大学课程计划项，我设计了接口CoursePlanningEntry。大学课程计划项只有一个位置（教室）和一个起止时间对，具有单个可区分资源（教师），所以CoursePlanningEntry继承了PlanningEntry<Teacher> 、SingleLocationEntry和SingleResourceEntry<Teacher>接口。

为了实现CoursePlanningEntry，我实现了CourseEntry类。它继承了CommonPlaningEntry<Teacher>来实现复用。CourseEntry类的抽象函数为AF(e) = {timeslot, location, teacher},表示不变量为变量不为null且所有位置都不可共享。

对于CourseEntry的部分公共特性，我采用委托的方式实现。利用SingleLocationEntryImpl来实现SingleLocationEntry接口的功能，又利用SingleResourceEntryImpl来实现SingleResourceEntry接口的功能。

1 public CourseEntry(String name, Timeslot timeslot, Location location) {

2 this.name = name;

3 state = new WaitingState(this);

4

5 singleLocationEntryImpl = new SingleLocationEntryImpl(location);

6 singleTimeslotEntryImpl = new SingleTimeslotEntryImpl(timeslot);

7 singleResourceEntryImpl = new SingleResourceEntryImpl<>();

8 checkRep();

9 }

而对于其自身的其他函数：

getCurrentTimeslot和getCurrentLocation：本来就只有一个，所以直接返回就好了。

1 @Override

2 public Timeslot getCurrentTimeslot() {

3 return singleTimeslotEntryImpl.getTimeslot();

4 }

5

6 @Override

7 public Location getCurrentLocation() {

8 return singleLocationEntryImpl.getLocation();

9 }

conflictLocationWith和conflictResourceWith：思路类似，判断两个计划项的时间对是否交叉。若交叉，则判断位置（资源）是否相同。

1 @Override

2 public Location getCurrentLocation() {

3 return singleLocationEntryImpl.getLocation();

4 }

5

6 @Override

7 public boolean conflictLocationWith(PlanningEntry<?> entry) {

8 if (!(entry instanceof CoursePlanningEntry))

9 return false;

10 CoursePlanningEntry anotherEntry = (CoursePlanningEntry)entry;

11 Timeslot anotherTimeslot = anotherEntry.getTimeslot();

12 if (!getTimeslot().intersect(anotherTimeslot))

13 return false;

14 return getLocation().equals(anotherEntry.getLocation());

15 }

16

17 @Override

18 public boolean conflictResourceWith(PlanningEntry<?> entry) {

19 if (!(entry instanceof CoursePlanningEntry))

20 return false;

21 if (singleResourceEntryImpl.getResource() == null)

22 return false;

23 CoursePlanningEntry anotherEntry = (CoursePlanningEntry)entry;

24 Timeslot anotherTimeslot = anotherEntry.getTimeslot();

25 if (!getTimeslot().intersect(anotherTimeslot))

26 return false;

27 return getResource().equals(anotherEntry.getResource());

28 }

## 面向复用的设计：R

PlanningEntry<R>中的泛型参数 R，用于表示计划项对应的资源类型。

### 航班管理（Flight Schedule）

对于航班计划项，其对应的资源为“飞机”， 属性包括飞机编号、机型号(A350、B787、 C919 等)、座位数、机龄(例如 2.5 年)。我设计使用Plane类来实现，其中飞机编号、机型号为字符串，座位数为整数，机龄为浮点数。

1 public class Plane {

2 private final String id;

3 private final String type;

4 private final int seats;

5 private final double age;

6

7 public Plane(String id, String type, int seats, double age) {

8 this.id = id;

9 this.type = type;

10 this.seats = seats;

11 this.age = age;

12 checkRep();

13 }

14 }

此类本质为对于四个变量的包装，拥有对应的getters。其表示不变量为id和type不为null，seats为正数且age非负。判断两个Plane相等当且仅当四个变量相等。

### 高铁车次管理（Train Schedule）

高铁应用中的资源是“车厢”，属性包括车厢唯一编号、类型（商务、一等、二等、软卧、硬卧、硬座、行李车、餐车）、定员数（例如 100 人）、 出厂年份。我设计了 TrainCabin类来实现，其中编号、定员数和出厂年份使用整数，而类型使用一种enum类使用，包含上面的八种类型。

1 public class TrainCabin {

2

3 public enum TrainCabinType {

4 BUSINESS, FIRST\_CLASS, SECOND\_CLASS, SOFT\_SLEEPER, HARD\_SLEEPER, HARD\_SEAT, LUGGAGE\_CAR, DINING\_CAR

5 }

6

7 private final int id;

8 private final TrainCabinType type;

9 private final int seats;

10 private final int factoryYear;

11

12 public TrainCabin(int id, TrainCabinType type, int seats, int factoryYear) {

13 this.id = id;

14 this.type = type;

15 this.seats = seats;

16 this.factoryYear = factoryYear;

17 checkRep();

18 }

19 }

此类本质为对于四个变量的包装，拥有对应的getters。其表示不变量为id为正，type不为null，seats为正数且age非负。判断两个TrainCabin相等当且仅当四个变量相等。

### 大学课程管理（Course Schedule）

课表应用中的资源为“教师”，属性包括身份证号、姓名、性别、职称。我设计了Teacher类来实现，其中四个变量均为字符串。

1 public class Teacher {

2 private final String idNumber, name, sex, title;

3

4 public Teacher(String idNumber, String name, String sex, String title) {

5 this.idNumber = idNumber;

6 this.name = name;

7 this.sex = sex;

8 this.title = title;

9 checkRep();

10 }

11 }

此类本质为对于四个变量的包装，拥有对应的getters。其表示不变量为四个变量均不为null。判断两个Teacher相等当且仅当四个变量相等。

## 面向复用的设计：Location

一个“位置”对象的属性包括:经度、纬度、名称、是否可共享使用。所谓 的“是否可共享”是指:该位置是否可同时被多个计划项所使用。但是其实不是所有的计划项的位置都包含经纬度，为此我设计了一个immutable的接口——Location，包含getName和isShareable两个方法。

为了实现实现Location的基本功能，我实现了CommonLocation类。它包含两个变量：一个字符串表示名称，一个布尔变量表示可否共享。它只包含getters。两个位置相等等等价于它们的名称相等。而为了拓展包含经纬度的位置类，我又设计了LocationWithLatitudeAndLongitude类，它继承了CommonLocation类来实现复用，同时它又包含了两个变量：latitude和longitude，用于表示经纬度，也有对应的getters

## 面向复用的设计：Timeslot

Timeslot类用于表示一个单独的起止时间对，它包含两个时间，是一个immutable类。由于它是一个immutable类，所以它的内部时间变量也使用一个immutable类——LocalDateTime——来表示。Timeslot类的表示不变量为开始时间必须小于开始时间。Timeslot类包含对于起止时间对的getters，同时包含两个静态函数来构造一个Tiemslot类的实例。

Timeslot的of函数利用两个LocalDateTime来构造一个Timeslot。

1 public static Timeslot of(LocalDateTime startTime, LocalDateTime endTime) {

2 return new Timeslot(startTime, endTime);

3 }

Timeslot的parse函数则使用两个格式为yyyy-MM-dd HH:mm的字符串，将它们转换为起始时间对。转换的过程使用了LocalDateTime的parse方法。但是由于ISO\_LOCAL\_DATE\_TIME的格式为yyyy-MM-ddTHH:mm:ss，所以需要对格式进行转换。

1 public static Timeslot parse(String startText, String endText) throws DateTimeParseException {

2 return new Timeslot(LocalDateTime.parse(startText.replace(' ', 'T'), TIMESLOT\_FORMATTER),

3 LocalDateTime.parse(endText.replace(' ', 'T'), TIMESLOT\_FORMATTER));

4 }

为了方便判断两个计划项是否会发生冲突，所以又设计了一个方法intersect来判断两个时间对的时间是否有交集。通过分情况讨论来实现。

1 public boolean intersect(Timeslot anotherTimeslot) {

2 if (equals(anotherTimeslot))

3 return true;

4

5 LocalDateTime anotherStartTime = anotherTimeslot.getStartTime();

6 LocalDateTime anotherEndTime = anotherTimeslot.getEndTime();

7 if (startTime.compareTo(anotherStartTime) >= 0 && startTime.compareTo(anotherEndTime) < 0)

8 return true;

9 if (endTime.compareTo(anotherStartTime) > 0 && endTime.compareTo(anotherEndTime) <= 0)

10 return true;

11 if (startTime.compareTo(anotherEndTime) >= 0)

12 return false;

13 if (endTime.compareTo(anotherStartTime) <= 0)

14 return false;

15 return true;

16 }

## 面向复用的设计：EntryState及State设计模式

出于简化代码结构，将代码解耦，提高代码的复用度的目的，我在实验中使用了状态模式。设计EntryState接口来提取各个状态的方法。它包含allocated、run、cancel和end方法，用于转移各个机计划项的状态。由于部分计划项是可阻塞的，所以又设计了一个BlockableEntry来抽象出block方法。

由于部分状态转移不合法，所以我又设计了一个checked Exception类 IllegalStateTransitionException类来表示一次非法的状态转移。

为了提高代码的复用率，所以我设计了一个抽象类CommonEntryState。鉴于大部分的状态可以到达的状态有限，所以我让CommonEntryState的所有方法都抛出IllegalStateTransitionException，之后根据各个计划项的具体情况再修改。

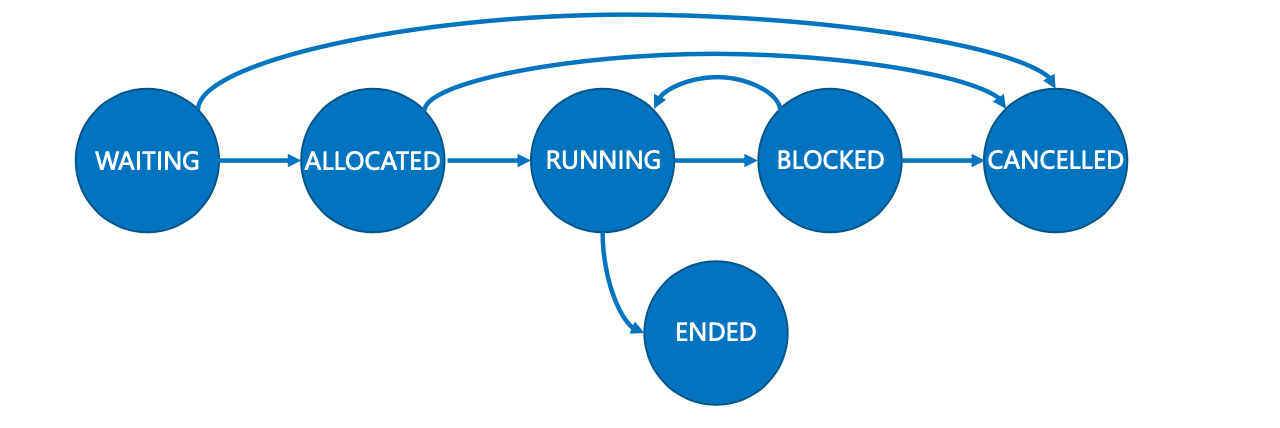


图 3‑1可阻塞的计划项的状态转移图

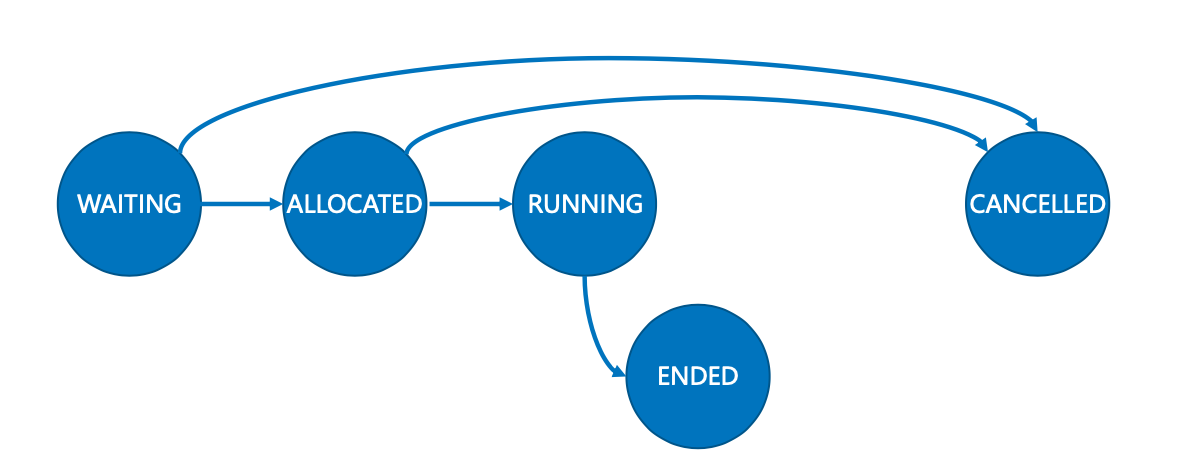


图 3‑2不可阻塞的计划项的状态转移图

## 面向应用的设计：Board

对于每个应用场景，每个特定位置都有一个信息板，用于实时公布该位置上过去发生过的和后续即将发生的计划项，以便于广大用户及时了解计划项的状态变化。针对本实验我实现的三个应用，相应的信息板如下所述:

### 航班管理（Flight Schedule）

某机场里的航班状态显示屏：分为抵达屏和出发屏，前者展示抵达该机场的航班信息及其状态，后者展示从该机场出发的航班信息及其状态。注意:第一列里的所有时间都是计划降落时间(对抵达航班来说)、计划起飞时间(对出发航班来说)。所以我使用一个List来存储它包含的计划项，每次添加计划项时都会判断新的计划项是否与之前的计划项是否产生冲突。

### 高铁车次管理（Train Schedule）

高铁站里的车次状态显示屏：与航班管理的状态显示屏类似，只是需要额外考虑经停列车的情况。

### 大学课程管理（Course Schedule）

教室外悬挂的教室占用情况表：列出当日在本教室内上的所有课程及其状态。与航班管理的状态显示屏相同，第一列的时间都是计划时间。

## Board的可视化：外部API的复用

为了实现GUI来展示航班信息，我使用了JTable来存储GUI信息，使用generateJTable方法来生成一个JTable。为此我为每个Board都设计了一个私有类BoardTable来生成一个JTable。它利用渲染器来实现子表格的功能。例如在flight类中，BoardTable类重写了getCellRenderer方法。

1 @Override

2 public TableCellRenderer getCellRenderer(int row, int column) {

3 switch (row) {

4 case 1:

5 return new TableCellRenderer() {

6 private String[] columnNames = {"Time", "Plane", "Location", "State"};

7 private JTable subTable = new JTable(new DefaultTableModel(data1, columnNames));

8

9 @Override

10 public Component getTableCellRendererComponent(JTable table, Object value, boolean isSelected,

11 boolean hasFocus, int row, int column) {

12 return subTable;

13 }

14 };

15 case 3:

16 return new TableCellRenderer() {

17 private String[] columnNames = {"Time", "Plane", "Location", "State"};

18 private JTable subTable = new JTable(new DefaultTableModel(data2, columnNames));

19

20 @Override

21 public Component getTableCellRendererComponent(JTable table, Object value, boolean isSelected,

22 boolean hasFocus, int row, int column) {

23 return subTable;

24 }

25 };

26 default:

27 return tcr;

28 }

29 }

这样就实现了子表格的功能。之后使用visualize方法显示表格，其实现的效果如下：



图 3‑3航班管理的可视化Board

其他的两个场景实现的方法类似。

## 可复用API设计及Façade设计模式

### 检测一组计划项之间是否存在位置独占冲突

由于各个PlanningEntry都实现了conflictLocationWith方法，所以我只用依次枚举计划项对，利用conflictLocationWith方法来判断两个计划项是否存在位置独占冲突。

1 public boolean checkLocationConflict(List<? extends PlanningEntry<R>> entries) {

2 for (ListIterator<? extends PlanningEntry<R>> iterator1 = entries.listIterator(); iterator1.hasNext();) {

3 PlanningEntry<?> entry1 = iterator1.next();

4 for (ListIterator<? extends PlanningEntry<R>> iterator2 = entries.listIterator(iterator1.nextIndex());

5 iterator2.hasNext();) {

6 PlanningEntry<R> entry2 = iterator2.next();

7 if (entry1.conflictLocationWith(entry2))

8 return true;

9 }

10 }

11 return false;

12 }

### 检测一组计划项之间是否存在资源独占冲突

与检测一组计划项之间是否存在位置独占冲突的方法类似，利用conflictResourceWith方法类判断。

1 public boolean checkResourceExclusiveConflict(List<? extends PlanningEntry<R>> entries) {

2 for (ListIterator<? extends PlanningEntry<R>> iterator1 = entries.listIterator(); iterator1.hasNext();) {

3 PlanningEntry<R> entry1 = iterator1.next();

4 for (ListIterator<? extends PlanningEntry<R>> iterator2 = entries.listIterator(iterator1.nextIndex());

5 iterator2.hasNext();) {

6 PlanningEntry<R> entry2 = iterator2.next();

7 if (entry1.conflictResourceWith(entry2))

8 return true;

9 }

10 }

11 return false;

12 }

### 提取面向特定资源的前序计划项

首先找到所有包含特定资源的计划项，之后根据它们的时间关系来更新答案。当新的计划项的在给定的计划项之前且在前一个答案的之后时，更新答案。

1 if (anotherSingleTimeslotEntry.getTimeslot().getEndTime()

2 .compareTo(singleTimeslotEntry.getTimeslot().getStartTime()) <= 0) {

3 if (resultEntry != null) {

4 if (anotherSingleTimeslotEntry.getTimeslot().getStartTime()

5 .compareTo(anotherSingleTimeslotEntry.getTimeslot().getStartTime()) <= 0) {

6 resultEntry = entry;

7 }

8 } else {

9 resultEntry = entry;

10 }

11 }

## 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

利用factory method，客户端可以避免通过 new 的方式实例化具体的 PlanningEntry 子类型，同时将参数统一。为此我抽象了一个Factory接口，用来实现工厂方法。

对于每个计划项，我实现了一个对应的Factory类，利于对于航班计划项，我实现的Factory类将参数转化为对应构造函数的参数。

1 public class Factory implements base.Factory<Plane> {

2

3 public static final Factory FACTORY = new Factory();

4

5 @Override

6 public PlanningEntry<Plane> creatPlanningEntry(String name, List<Timeslot> timeslots, List<Location> locations) {

7 if (timeslots.size() != 1 || locations.size() != 2)

8 throw new IllegalArgumentException();

9 return FlightPlanningEntry.factoryMethod(name, timeslots.get(0), locations.get(0), locations.get(1));

10 }

11

12 }

### Iterator

我为自己的Board类添加了迭代器的功能，使它实现Iterable接口，可通过迭代器按照时间从早到晚的次序遍历其中包含的所有计划项。为此我为Board添加了setSortKeyLocation方法来实现设置迭代根据地点迭代。同时由于航班、高铁的迭代还又根据离开、到达等情况，所以我又添加了setIteratorType方法来设置迭代的参数。当需要迭代时，我将所有的符合条件的计划项添加到列表sortedEntries，之后根据参数排序，最后返回sortedEntries的迭代器即可。

### Strategy

将PlanningEntryAPIs的方法抽象到一个接口，之后在实现暴力策略，即PlanningEntryAPIsBruteForceStrategy类。它继承了PlanningEntryAPIs类。随后在PlanningEntryAPIsWithMapStrategy中重写checkResourceExclusiveConflict方法，使用Map来优化。

1 public boolean checkResourceExclusiveConflict(List<? extends PlanningEntry<R>> entries) {

2 Map<R, List<PlanningEntry<R>>> resourceMap = new HashMap<>();

3 for (PlanningEntry<R> entry : entries) {

4 if (entry instanceof SingleResourceEntry) {

5 SingleResourceEntry<R> singleResourceEntry = (SingleResourceEntry<R>)entry;

6 R resource = singleResourceEntry.getResource();

7 List<PlanningEntry<R>> list;

8 if (resourceMap.containsKey(resource)) {

9 list = resourceMap.get(resource);

10 } else {

11 list = new ArrayList<>();

12 resourceMap.put(resource, list);

13 }

14 list.add(entry);

15 } else if (entry instanceof MultipleSortedResourceEntry) {

16 MultipleSortedResourceEntry<R> multipleSortedResourceEntry = (MultipleSortedResourceEntry<R>)entry;

17 for (R resource : multipleSortedResourceEntry.getResources()) {

18 List<PlanningEntry<R>> list;

19 if (resourceMap.containsKey(resource)) {

20 list = resourceMap.get(resource);

21 } else {

22 list = new ArrayList<>();

23 resourceMap.put(resource, list);

24 }

25 list.add(entry);

26 }

27 }

28 }

29 for (List<PlanningEntry<R>> list : resourceMap.values()) {

30 int n = list.size();

31 for (int i = 0; i < n; ++i)

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

33 if (list.get(i).conflictResourceWith(list.get(j)))

34 return true;

35 }

36 return false;

37 }

## 应用设计与开发

利用上述设计和实现的ADT，实现手册里要求的各项功能。

### 航班应用

我使用JSwing类实现航班应用。我将对于文件、资源、位置和计划项的管理置于菜单栏中，在主要的展示页面中展示状态，具体页面如下：

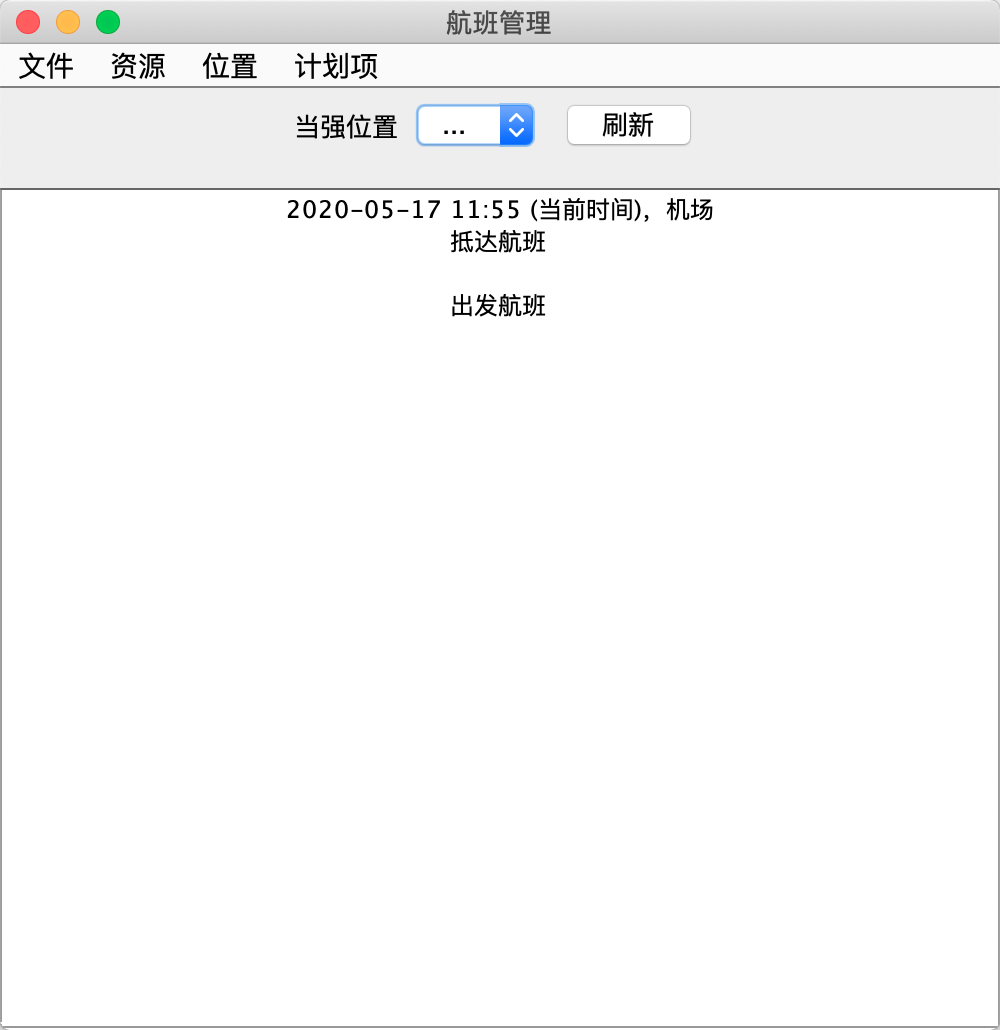


图 3‑4航班管理应用

中间的当前位置可以选择，点击刷新按键后会更新状态显示屏。

其中文件菜单栏可以加载合法的解析文件，从文件中解析出对应的计划项，同时所有前面的计划项都会别清除。

资源菜单栏中包含添加和删除资源，同时也可以查询与该资源相关的所有计划项。在添加资源时我利用正则表达式一定程度上限制了输入的合法性。期间利用了<https://github.com/xfhy/JournalManageSystem/blob/27c5964746f6f67c6e98b5b375ca8f92dc92223d/src/liang/guo/diary/util/MyRegExp.java>的代码。如飞机名称输入时的内容（包括未完成的输入）必须符合表达式"^[NB]\d{0,4}$"。无法使用正则限定的部分在之后判断。删除的则使用选择的方式在拥有的资源中选择删除。当资源为空时则会报错。



图 3‑5添加资源对话框



图 3‑6删除资源对话框



图 3‑7资源为空报错

对于位置的管理和计划项的管理类似。而当需要改变计划项的状态时这可以在计划项的管理中打开更改计划项对话框，实现状态转移。检测计划项冲突可以在计划项菜单中打开检测冲突对话框。之后便会报出是否存在冲突。

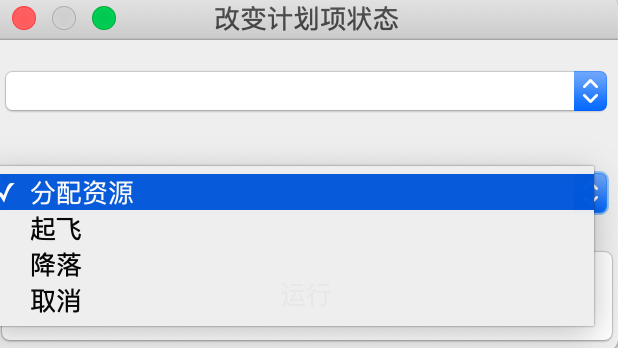


图 3‑8改变计划项状态对话框



图 3‑9检测冲突对话框



图 3‑10无冲突提示

### 高铁应用

高铁应用与航班管理相似，不过由于其具有多个位置，所以其添加计划项的界面与航班管理的界面有所不同。其中“添加一对起止位置/时间”按键会将当前界面的起止时间对和起止地点添加到计划项中。要求新的起点必须是上一个终点（第一项除外），所有的时间递增。



图 3‑11添加计划项界面

### 课表应用

应用与航班管理的界面类似，不过地址变为一个地点。

## 基于语法的数据读入

此功能由parse方法实现。

由于需要忽略缩进的影响，所以读入的每一行时需要删除前后的空字符（如空格、制表符）。之后按照顺序对输入解析。对于各个不同的键值对，采用正则表达式来判断是否合法。之后来解析出对应的值。

首先使用getNonmptyLineFunction来获得消除缩进后的字符串。

1 final Function<Void, String> getNonmptyLineFunction = x -> {

2 while (scanner.hasNext()) {

3 ++line[0];

4 String lineString = scanner.nextLine().trim();

5 if (lineString.isEmpty())

6 continue;

7 return lineString;

8 }

9 return null;

10 };

然后按照顺序利用正则表达式对键值对检测，如飞机的的正则表达式为"^Flight:\d{4}-\d{2}-\d{2},[A-Z]{2}\d{2,4}$"，机龄的正则表达式为"^Age:(0|0\.0|[1-9]?\d(.\d)?)$"。

如果发现有文件解析不合法，则抛出一个checked Exception类FileParseException，它的信息为第几行发生错误。

1 public class FileParseException extends Exception {

2 private static final long serialVersionUID = 7900029340035359753L;

3

4 public FileParseException(File file, int line) {

5 super(String.format("Parse file %s failed on line %d.", file.getName(), line));

6 }

7 }

## 应对面临的新变化

只考虑你选定的三个应用的变化即可。

### 变化1

变化：对航班管理的变化：航班支持经停，即包含最多 1 个中间经停机场，例如哈尔滨-威海-深圳。

评估：将之前使用的PairLocationsEntry和SingleTimeslotEntry更改为之前的设计的MultipleLocationsEntry和MultipleTimeslotsEntry就可以实现更改，之后更改其他调用的接口，修改约为40~50行。

实现：将FlightPlannginEntry实现的接口PairLocationsEntry和SingleTimeslotEntry改为MultipleLocationsEntry和MultipleTimeslotsEntry，FlighEntry中则改为使用MultipleLocationsEntryImpl和MultipleTimeslotsEntryImpl。同时在应用的添加界面增加中间点的按钮。

### 变化2

变化：如果高铁车次已经分配了车厢资源，则不能被取消。

评估：修改train中的AllocatedState中的cancel函数，修改应在十行内。

实现：让AllocatedState的cancel函数抛出异常。

### 变化3

变化：课程可以有多个教师一起上课，且需要区分次序（即多名教师的优先级）。

评估：修改Teacher类，将CoursePlanningEntry的SingleResourceEntry接口改为MultipleResourceEntry，修改约为20~30行。

变化：在Teacher类中添加一个变量priority来表示优先级。将CourseEntry的SingleResourceEntry更改为MultipleResourceEntryImpl，修改对应的状态转移。

## 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。

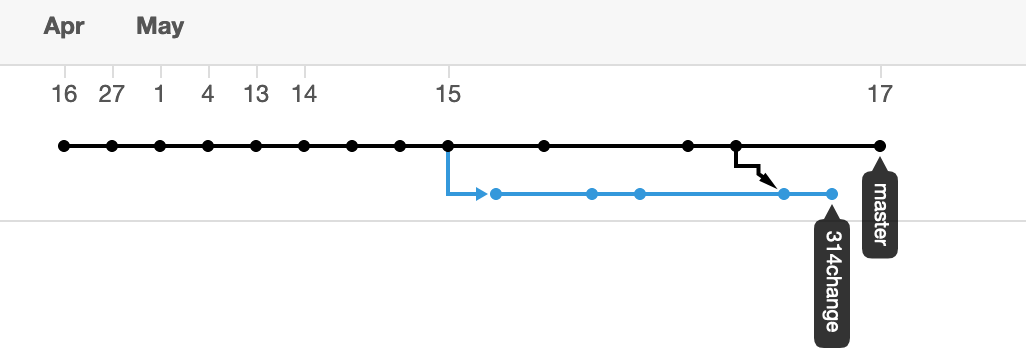


图 3‑12Object Graph

# 实验进度记录

请使用表格方式记录你的进度情况，以超过半小时的连续编程时间为一行。

|  |  |  |  |
| --- | --- | --- | --- |
| 日期 | 时间段 | 计划任务 | 实际完成情况 |
| 2020-05-13 | 20:00-23:00 | 完成base包 | 完成 |
| 2020-05-14 | 20:00-23:00 | 完成flight包，包括GUI | 完成 |
| 2020-05-15 | 20:00-23:00 | 完成train和course包 | 完成 |
| 2020-05-16 | 00:00-02:00 | 完成APIs和314change | 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 |
| 使用正则表达式限制输入 | 在GitHub上搜索具有类似功能的项目，最后参考了GitHub上的代码 |
| 子表格实现 | 通过网上搜索，查到使用JTable中的渲染器实现。 |

# 实验过程中收获的经验、教训、感想

## 实验过程中收获的经验和教训

在此次实验中我体会到了OOP思想对于实现复用，减少工作量的作用。通过抽象，我可以更好的把握整个实验中各个部分的关系。而通过将各个项目中的公共部分抽象实现，可以有效得提高复用率，减少耦合度。

不过实现所有的代码后发现还是有很多部分可以抽象，如Board、App，它们有许多重复的代码，我在实现时为了简单实现复制了大量代码，但其实它们的功能重合度很高。不过好的抽象的难度较高，在实验的过程中我发现自己有过过度抽象，部分抽象毫无意义，反而徒增工作量。

## 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？

面向ADT的编程让代码的结构更为清晰，大大削减了代码量，减少了BUG产生的概率。同时面向ADT的编程使得单元测试的意义更加明确。

复用可以减少代码量，减少BUG，同时更改功能时也可以大大减少变化量。

1. 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？

为ADT撰写复杂的specification, invariants, RI, AF，时刻注意ADT是否有rep exposure，我可以更加容易的实现测试优先的编程，也可以让自己更加容易地恢复到工作状态。你愿意在以后的编程中坚持这么做。

1. 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？

难度较大，因为要减少各个部分的耦合度，同时对于specification的要求也提高了，使得类和方法的作用更加清晰。但是完成后发现调用自己的API可以大大减少工作量，不用重写代码了还是很开心。

1. 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？

设计模式可以大大提高实现时的效率。由于设计模式，我虽然在提前规划项目的结构时花费较长的时间，但是base包完成后好面的实现变很容易。

1. 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？

有点复杂和麻烦，不过正则表达式使得功能可以更容易实现。

1. 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？

首先是在对于不同功能的抽象，需要对于功能的抽象清晰且恰到好处，不能过度，也不能不够。要实现思考出将哪些方法抽象到统一个接口中。

其次需要有清晰的项目规划，因为ADT在之后的项目要多次使用，如果ADT的接口或者层级关系有问题，到后来项目复杂的时候改起来非常麻烦。

ADT要有清晰的说明，方法名要清晰。不仅仅是别人可能读不懂，类多了之后自己可能也读不懂。

1. “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？

继承可以减少代码量，但是如果单一地使用继承会使得项目的类过度，所以各个设计模式要综合使用，在合适的场景使用合适的设计模式。

同时不要可以追求设计模式，搞得为了OOP而OOP，过度抽象反而得不偿失。

1. 关于本实验的工作量、难度、deadline。

工作量较大，难度除了GUI部分适当，GUI部分难度较大。但是实验的部分要求不明确，导致设计时常需要修改已经实现的部分。

1. 到目前为止你对《软件构造》课程的评价。

软件构造非常倾向于构造，意味着要设计软件，不能凭感觉写。有了ADT、设计模式的知识后，可以让程序的结构更加清晰，大大提过了自己对于较大的程序设计的知识。