[BUAA-OO] 第二单元总结

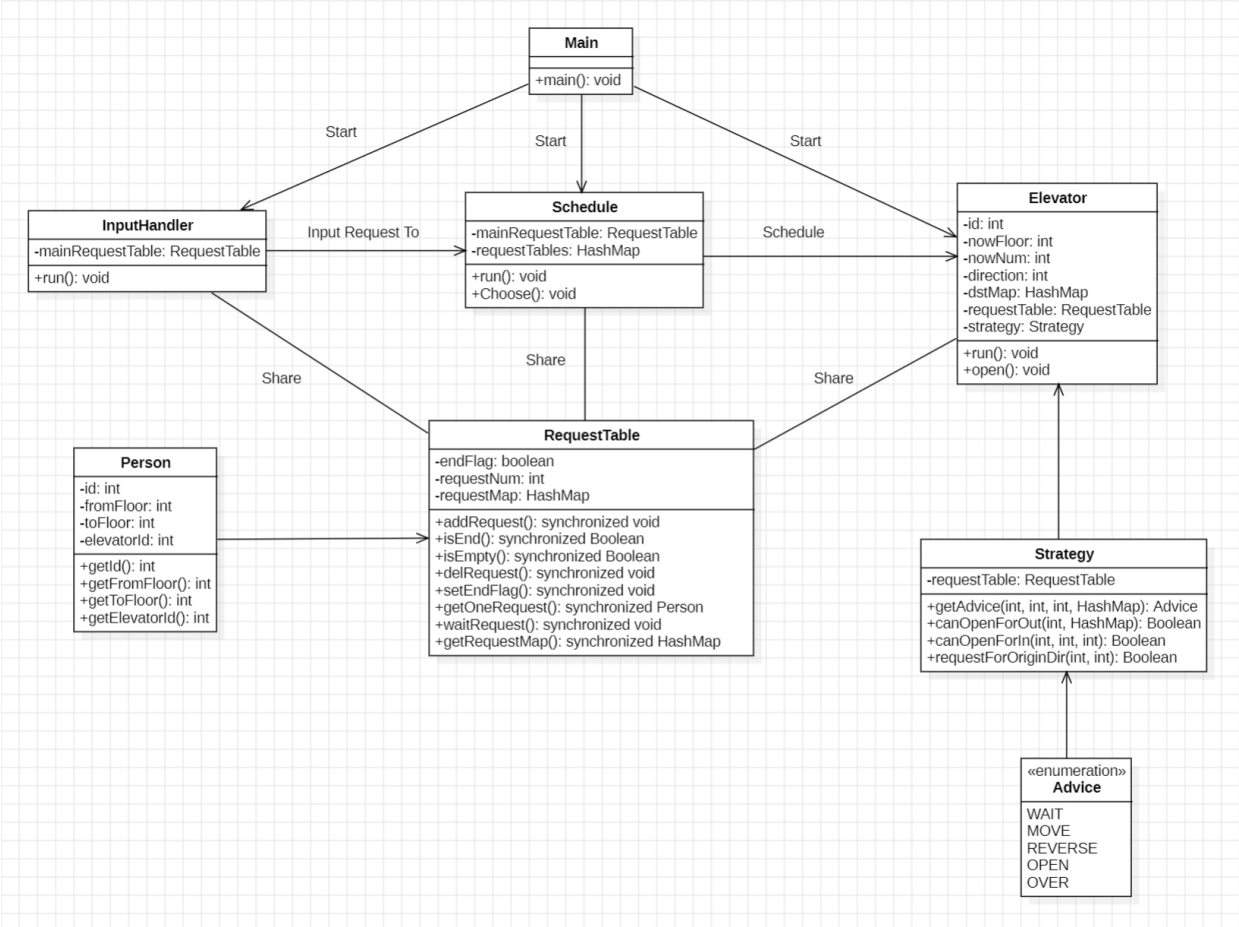
前言

第二单元的主要任务是完成一个“**电梯调度问题**”，问题的关键在于“**多线程**”，我们需要接触并学习多线程的思想，处理线程交互和维护线程安全问题。

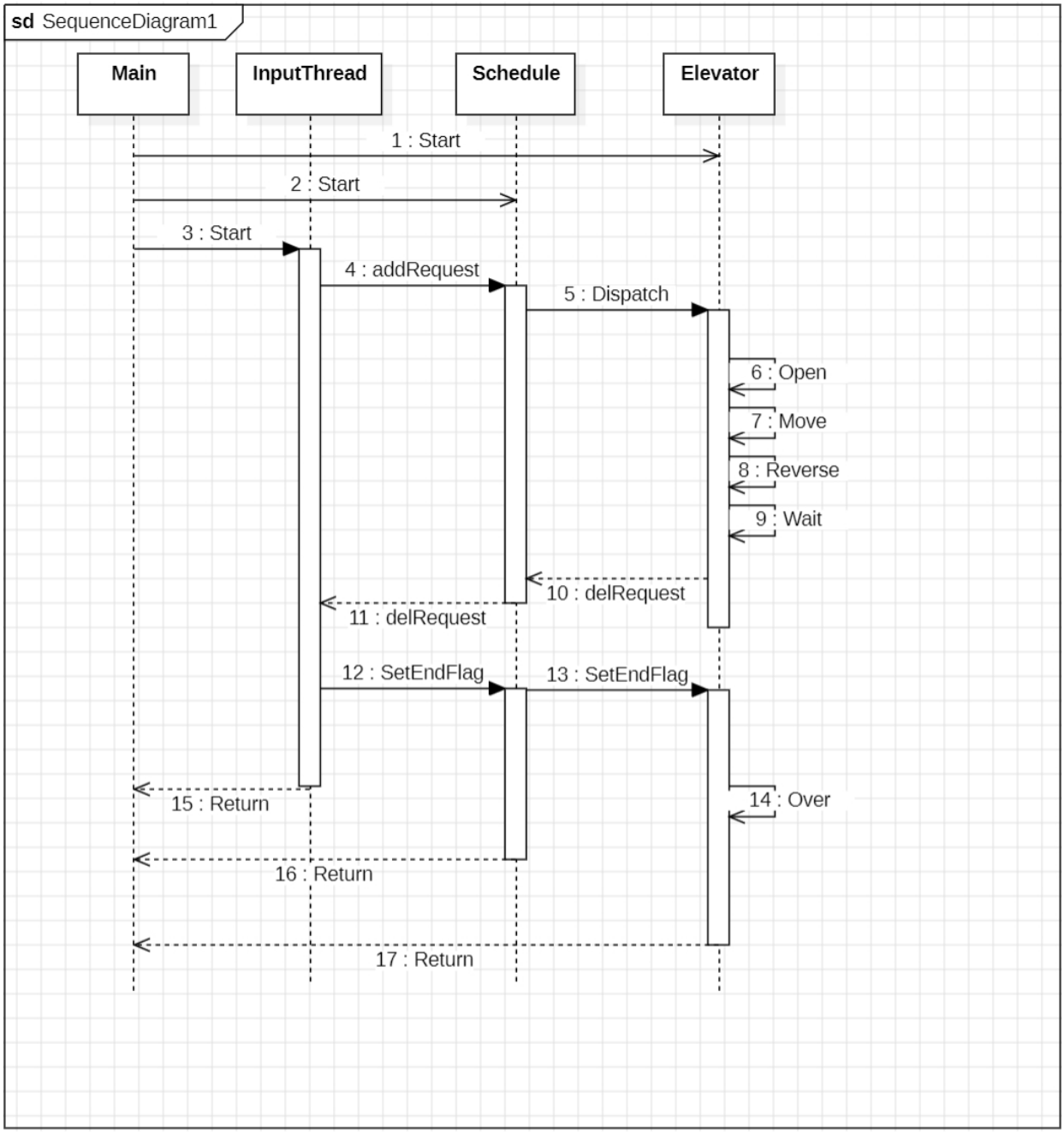
第五次作业

本次作业的目标是完成一个简单的**电梯调度**任务，相较于往年来说难度略有下降，取消了“横向电梯”的设置，并且在本次作业中每个人只能乘坐确定的电梯，所以也不存在电梯最优分配问题，所以本次作业的主要目标是初步了解并学习使用**多线程**编程，虽然难度不大，但是对于第一次接触多线程的我来说也是折磨了我很久，建议提前预习oolens公众号上关于**“多线程”的文章**，另外学习参考**课上实验的架构**，对完成本次作业有很大的帮助。

UML类图：



协作图



代码架构分析：

本次作业我采用了**“生产者-消费者”**模式，采用**“电梯和策略相分离”**的架构，对于电梯调度策略选取**“LOOK算法”**，接下来我将一一介绍：

“生产者-消费者”模型

查阅资料可知：

> 生产者消费者模式是通过一个容器来解决生产者和消费者的强耦合问题。生产者和消费者彼此之间不直接通讯，而通过阻塞队列来进行通讯，所以生产者生产完数据之后不用等待消费者处理，直接扔给阻塞队列，消费者不找生产者要数据，而是直接从阻塞队列里取，阻塞队列就相当于一个缓冲区，平衡了生产者和消费者的处理能力

> 这个阻塞队列就是用来给生产者和消费者解耦的。
>
> 如果缓冲区已经满了，则生产者线程阻塞；
>
> 如果缓冲区为空，那么消费者线程阻塞；

因此我们设置一个类`RequestTable`作为**“阻塞队列”**，**输入线程**(`InputHandler`)、调度线程(`Schedule`)、**电梯线程**(`Elevator`)**共享**这一对象——由输入线程作为**生产者**处理输入，给共享对象`RequestTable`加入请求；`Schedule`对共享对象`RequestTable`进行调度分配给对应的电梯的`RequestTable`（每个电梯都用有属于自己的一个`RequestTable`），然后电梯线程作为**消费者**对请求进行处理。

RequestTable

要实现上述过程，`RequestTable`类的编写至关重要，其包含三个成员变量：

```
private boolean endFlag;//标记请求是否结束
private int requestNum;//请求数量
private HashMap<Integer, ArrayList<Person>> requestMap;//出发地在key层的所有请求（人）的集合
```

注意到由于该类是**共享**类，在多线程运行的过程中，在同一时间可能要多个线程同时对其进行访问，这时就会存在数据**冒险**，为了让同一时间只有一个线程能对其进行访问，我们要给该类的所有**方法加锁**(`synchronized`)，并在方法里用`notifyall()`方法**唤醒**所有等待的线程，以继续运行：

```
public synchronized void addRequest(Person person) {
        int fromFloor = person.getFromFloor();
        this.requestMap.get(fromFloor).add(person);
        requestNum++;
        notifyAll();
}
//其他方法...
```

InputHandler

该类在课程组提供的输入输出接口文档中有详细的demo可参考

Schedule

作为调度策略，其**首要任务**就是将生产者的**“产品”**分配给消费者**“使用”**，因此其既要跟`InputHandler`**共享主请求**`mainRequestTable`，又要跟电梯类**共享**每个电梯维护的`requestTable`，因此其成员变量为：

```
private RequestTable mainRequestTable;//主请求
private HashMap<Integer, RequestTable> requestTables;//各个电梯的请求队列
```

而调度的实现过程大致为（可**参考实验的架构**）：

```
@Override
    public void run() {
        while (true) {
            if (mainRequestTable.isEmpty() && mainRequestTable.isEnd()) {
                //...给每个电梯的请求队列标记结束
```

```
                return;
            }
            Person person = mainRequestTable.getOneRequest();//...从
mainRequestTable中得到一个request
            if (person == null) {
                continue;
            }
            //...将person需求加入合适的电梯表
        }
    }
```

至此我们完成了**生产者和消费者模式**的应用。

电梯和策略相分离

面对请求，电梯可以有不同的处理策略，为了**灵活变换策略**，我们将电梯与策略进行**分离**，单独设置一个`Strategy`策略类，根据电梯的`RequestTable`调用**特定**策略进行处理，返回给电梯一个**建议**，电梯根据建议进行相应的操作，如此我们便可以实现灵活、高效地运行。

为了增强建议的可读性，我们建立一个**枚举类**`Advice`,存储电梯的各种指令:

```
public enum Advice {
    OPEN, WAIT, MOVE, REVERSE, OVER
}
```

电梯在得到策略类给出的建议后可以分别做不同的**操作**:

```
    @Override
    public void run() {
        try {
            while (true) {
                Advice advice = strategy.getAdvice(//...)//...从Strategy类得到建议
                if (advice == Advice.OVER) {
                    //...
                } else if (advice == Advice.MOVE) {
                    //...
                } else if (advice == Advice.REVERSE) {
                    //...
                } else if (advice == Advice.OPEN) {
                    //...
                } else {//Advice.WAIT
                    //...
                }
            }
        } catch (InterruptedException e) {
            throw new RuntimeException();
        }
    }
```

至此我们完成了**电梯与策略的分离。**

LOOK算法

事实上在指导书中课程组给定的是**ALS**策略，但是在参考学长学姐的架构之后，我认为**LOOK**算法更加简单易懂并且它也是大多数人乃至实际大多数电梯运行所采用的算法，因此我采用了LOOK算法作为策略，其内容是这样的（参考Hyggge学长博客）：

> **LOOK**
>
> 下面来简单介绍我在本次作业中使用的调度算法（LOOK）——
>
> - 首先为电梯规定一个初始方向，然后电梯开始沿着该方向运动。
> - 到达某楼层时，
>
>   首先判断是否需要开门
>
>   - 如果发现电梯里有人可以出电梯（到达目的地），则开门让乘客出去；
>   - 如果发现该楼层中有人想上电梯，并且目的地方向和电梯方向相同，则开门让这个乘客进入。
> - 接下来，进一步判断电梯里是否有人
>
>   。如果电梯里还有人，则沿着当前方向移动到下一层。否则，检查请求队列中是否还有请求（目前其他楼层是否有乘客想要进电梯）——
>
>   - 如果请求队列不为空，且某请求的发出地是电梯"前方"的某楼层，则电梯继续沿着原来的方向运动。
>   - 如果请求队列不为空，且所有请求的发出地都在电梯"后方"的楼层上，或者是在该楼层有请求但是这个请求的目的地在电梯后方（因为电梯不会开门接反方向的请求），则电梯掉头并进入"判断是否需要开门"的步骤（循环实现）。
>   - 如果请求队列为空，且输入线程没有结束（即没有输入文件结束符），则电梯停在该楼层等待请求输入（wait）。
>
>   > 注意：电梯等待时运行方向不变。在我的设计中，运行方向是电梯的一个状态量而不是过程量，用来表示**下一次move时的方向**。当有新请求进入请求队列时，电梯被唤醒，此时电梯的运行方向仍然是电梯wait前的方向。
>
>   - 如果请求队列为空，且输入线程已经结束，则电梯线程结束。

`Strategy`类的编写事实上就是将**LOOK**算法的叙述转化为代码即可，这里我就不展开说明了，不过请留意**是否可以开门让人进入电梯**(`canOpenForIn`)和**是否可以开门让人出电梯**(`canOpenForOut`)以及**是否某请求的发出地是电梯"前方"的某楼层**(`requestForOriginDir`)的实现。

wait与notify

当很长时间没有数据到达，而输入又没结束时，为了**避免长时间轮询导致线程空转**，浪费CPU资源，我们要用`wait()`方法**释放锁**，当被`notify`之后再与其他线程**争夺锁**，运行之后的代码。这里给出详细解释：

> 关于对象锁，notify只是唤醒一个线程B，B这个线程要等到当前对象释放synchronized的对象锁才能执行，也就是A flag.wait()才能执行。再用高级点的说法，就是notify是等待池进入锁池。
>
> wait()方法 表示持有对象锁的线程A准备释放对象锁权限，释放CPU资源并进入等待。

> Wait() 和notify() 方法只能从synchronized方法或块中调用，需要在其他线程正在等待的对象上调用notify方法。
>
> notifyAll 通知JVM唤醒所有竞争该对象锁的线程，线程A synchronized 代码作用域结束后，JVM通过算法将对象锁权限指派给某个线程X，所有被唤醒的线程不再等待。线程X synchronized 代码作用域结束后，之前所有被唤醒的线程都有可能获得该对象锁权限，这个由JVM算法决定。

在本程序中，**所有**的加锁方法都加入了`notifyall`方法（除了`waitRequest`），在**两处**调用了`wait`方法，分别是：

```
//RequestTable.getOneRequest()
if (this.isEmpty() && !this.isEnd()) {//当请求为空但并没有结束（可参考实验代码）
            try {
                wait();//等待
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
```

```
//Elevator.run()
else {//Advice.WAIT
      requestTable.waitRequest();
}
```

至此我们完成了本次作业的代码架构。

优化

本次作业由于一开始的良好架构，并没有进行更深的优化，在强测的性能上也取得了不错的成绩。

但是后续发现其实可以在一处小细节上做优化。在我的设计中，电梯执行开门操作后是先让外面的人进来再让电梯内到达目的地的人出去，但是由于电梯载客量的问题，可能会明明电梯并没有满载，电梯外的人却进不来的问题。要想解决这个问题进行优化也十分容易，只需要**先让人出去再放人进来**即可，相信这样的优化可以使得电梯性能在某些情况下有小幅度的提升。

bug分析

本次作业在强测和互测上都取得了不错的成绩，没有出现明显的bug，也没有对同房间的其他小伙伴造成伤害（很和谐

代码复杂度

Complexity metrics	周日	14 4月 2024 13:53:27 CST		
Method	CogC	ev(G)	iv(G)	v(G)
Elevator.Elevator(int, RequestTable)	1	1	2	2
Elevator.open()	13	4	10	10
Elevator.run()	16	3	6	9
InputHandler.InputHandler(RequestTable)	0	1	1	1
InputHandler.run()	5	3	3	4
Main.main(String[])	3	1	4	4
Person.Person(int, int, int, int)	0	1	1	1
Person.getElevatorId()	0	1	1	1
Person.getFromFloor()	0	1	1	1
Person.getId()	0	1	1	1
Person.getToFloor()	0	1	1	1
RequestTable.RequestTable()	1	1	2	2
RequestTable.addRequest(Person)	0	1	1	1
RequestTable.delRequest(int, int)	0	1	1	1
RequestTable.getOneRequest()	8	3	7	7
RequestTable.getRequestMap()	0	1	1	1
RequestTable.isEmpty()	0	1	1	1
RequestTable.isEnd()	0	1	1	1
RequestTable.setEndFlag()	0	1	1	1
RequestTable.waitRequest()	0	1	1	1
Schedule.Choose(Person)	0	1	1	1
Schedule.Schedule(RequestTable, HashMap<Intege	0	1	1	1
Schedule.run()	9	4	5	6
Strategy.Strategy(RequestTable)	0	1	1	1
Strategy.canOpenForIn(int, int, int)	11	4	7	8
Strategy.canOpenForOut(int, HashMap<Integer, A	2	2	1	2
Strategy.getAdvice(int, int, int, HashMap<Inte	15	6	4	7
Strategy.requestForOriginDir(int, int)	7	3	3	7
Class	OCavg	OCmax	WMC	
Advice	n/a	n/a	0	
Elevator	5.33	8	16	
InputHandler	2	3	4	
Main	4	4	4	
Person	1	1	5	
RequestTable	1.56	5	14	
Schedule	2.33	5	7	
Strategy	3.2	6	16	
Package	v(G)avg	v(G)tot		
	3	84		
Module	v(G)avg	v(G)tot		
hw5	3	84		
Project	v(G)avg	v(G)tot		
project	3	84		

发现`Elevator`的`run()`方法和`open()`方法的复杂度爆红了，主要原因是电梯运行过程中有多个运行操作，因此造成了较多的分支判断；同时电梯在进行开门的操作时，需要让电梯内的人出去，让电梯外的人进来，涉及多个共享类的访存，因此复杂度较高。此外一些策略类的方法也爆了红，我想也是因为分支判断较多吧。

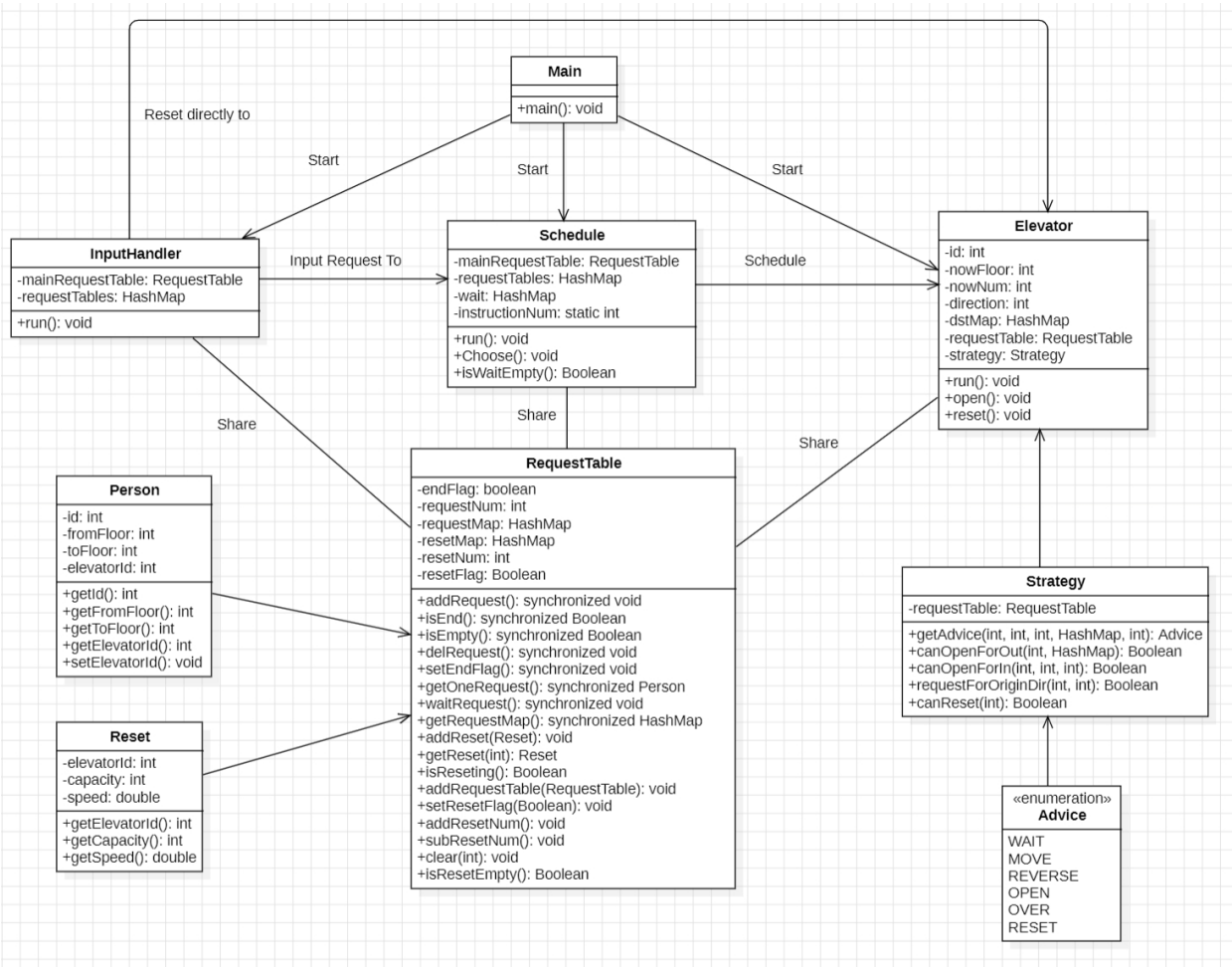
体会与感想

本次作业总体上取得了较为不错和理想的成绩，对多线程的理解和应用也有了初步的认识，但是由于本次题目较为简单，并没有出现死锁、轮询等线程不安全问题，这在后面的作业难度提升之后会陆续暴露出来。总而言之，在进行多线程设计时，一定要确保线程的安全，否则将会有难以预料的后果。并且由于多线程本身具有的不确定性，一定要进行多次测试，尽可能暴露问题，无论是自己构造数据还是借用大佬的评测机。

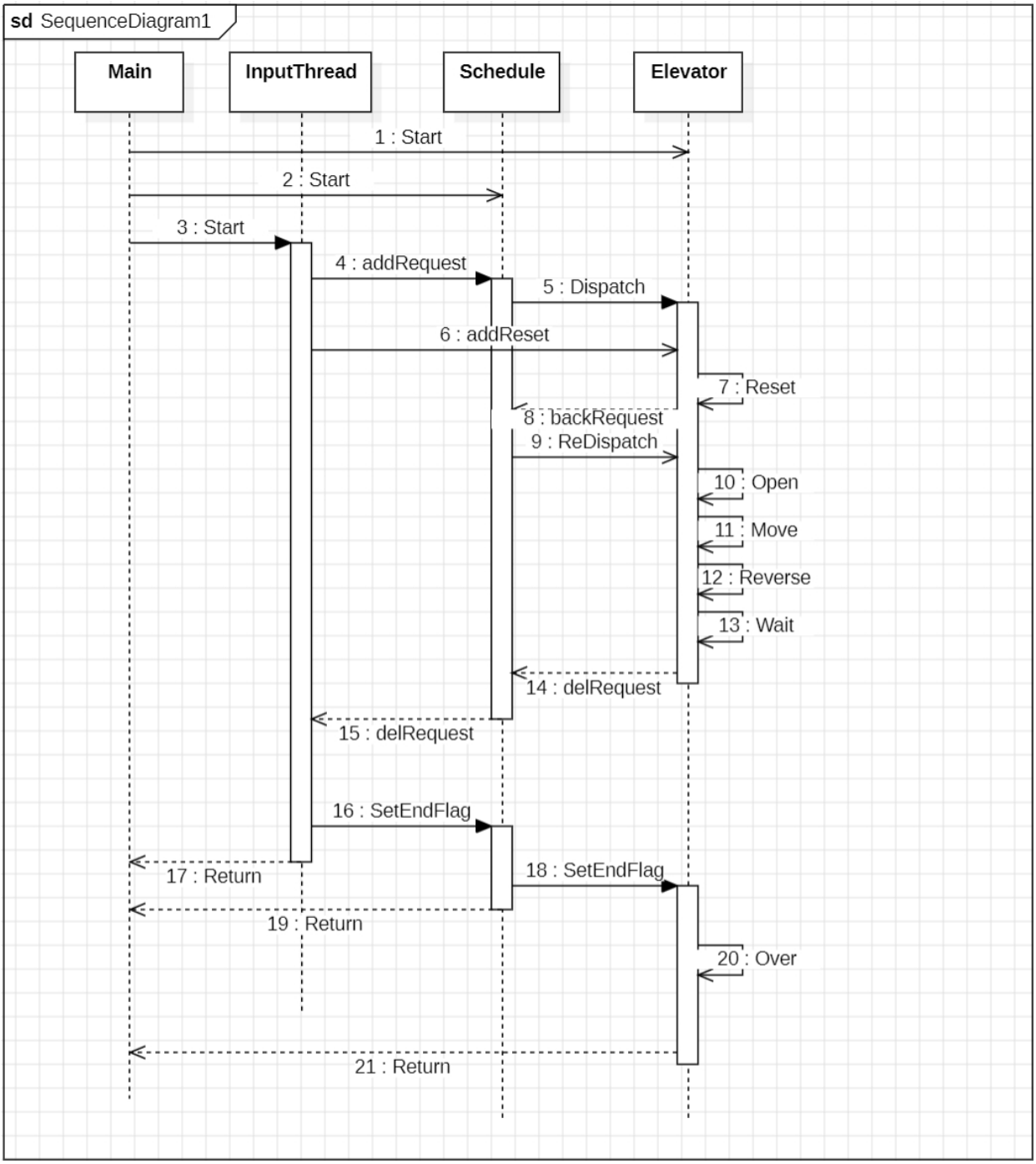
第六次作业

本次作业在第五次作业的基础上新增了`Reset`指令，要求电梯在收到`Reset`指令之后，将电梯内所有人员放出，进行载客量和运行速度的调整之后，重新投入运行。由于加入的`Reset`指令涉及人员的流动，也就是请求列表的删除和新增，线程不安全的风险**大大增加**；同时与hw5不同的是，本次作业不再指定电梯，需要我们自行设计**电梯调度器**（好在我在hw5中就已经设计了），根据调度策略的不同，每个人的性能分将会有较大的差别，因此本次作业我完成的十分痛苦（为什么每次都是单元的第二次作业），虽然没有进行重构，但是在追求性能分和确保正确性上来回徘徊，最后我还是选择了相对稳妥的方法，确保正确性，虽然最后性能分的确很差，但是至少保证了正确分。

UML类图



协作图



代码架构分析

Reset指令分析

本次架构沿袭了hw5的生产者消费者模型，不同的是增加了一个共享变量类`Reset`，用来**读入**、**传递**和**执行**重置指令。具体来说：

当输入处理线程读到了重置指令，则立马向对应电梯线程的请求队列传递`Reset`变量，注意这里我们**并没有经过调度器的调度**：

```
requestTables.get(reset.getElevatorId()).addReset(reset);
mainRequestTable.addResetNum();//总表的请求数量作为调度线程是否结束的标志
```

这样做的原因是：一方面`Reset`请求已经**指定了电梯**，不需要经过调度线程进行分配；另一方面，题目要求在接收到`Reset`指令到电梯开始`Reset`的过程中电梯不能输出超过两个`Arrive`，换句话说，当我们输入线程接收到`Reset`指令后要**尽可能迅速**地向电梯进行指令传递，使得电梯尽快中断当前进程，转而执行`Reset`指令，跳过调度线程能很好地实现这一点。

修改电梯运行的策略。在`strategy`给出策略的最开始，需要先检查是否有`Reset`指令，如果有则给出`Reset`的`advice`，否则继续判断其他的`advice`。换句话说，`Reset`**指令的优先级最高。**

```
public Advice getAdvice(int id, int nowNum, int nowFloor, int direction,
                            HashMap<Integer, ArrayList<Person>> dstMap,int
capacity) {
    if (canReset(id)) {
        return Advice.RESET;
    } else {
        //...others
    }
}
//canReset(int id)
public Boolean canReset(int id) {
    Reset reset = requestTable.getReset(id);
    if (reset == null) {
        return false;
    }
    return true;
}
```

而定位到`Reset`指令的具体执行过程。我是设计为：当电梯执行`Reset`时，将电梯内所有到达目的地的人out，将未到达目的地的人out并修改此人的`fromFloor`为`nowFloor`，再**加入总请求表**重新分配，另外将本电梯的请求表**清空**，之后结束`Reset`。

```
public void reset() throws InterruptedException {
    if (this.nowNum != 0) {
        //out person
    }
    this.requestTable.setResetFlag(true);
    //...Reset Begin
    this.mainRequestTable.addRequestTable(this.requestTable);//加入总表
    this.requestTable.clear(id);//清除本表
    this.mainRequestTable.subResetNum();//Reset数量-1
    //...Reset End
    this.requestTable.setResetFlag(false);
}
```

此时不要以为问题就轻松解决了，刚开始我也是这么想的，结果到周六中午才发现超时的问题，在截止的最后十分钟改出bug并提交，真可谓惊心动魄

结束条件

由于我们这样的设计，调度线程在输入结束之前并不会结束，我们要给定一个结束条件让调度线程结束，才能避免出现RTLE的情况。因此我用`Reset`的**总数量**标记调度线程是否应该结束，因为与hw5不同的便是，`Reset`指令的执行也许会向总表新增请求，这就需要调度线程**重新进行调度**，因此只有当`Reset`指令全部执行完以后，调度线程才可以真正的休息，因此我在共享类`RequestTable`中维护了一个共享变量`resetNum`来指示调度线程是否应该结束：

```
if (mainRequestTable.isEmpty() && mainRequestTable.isEnd()
                    && mainRequestTable.isResetEmpty() && isWaitEmpty()) {
    for (int i = 1; i <= 6; i++) {
        requestTables.get(i).setEndFlag();
    }
    return;
}
```

这样做就能正确结束程序，避免发生tle的情况。

眼尖的你可能发现了，在结束条件中还多了一个`isWaitEmpty()`条件，这个又是因为什么呢，这就不得不提到接下来的**调度策略**了。

调度策略

由于新增调度要求，我们必须设计一个调度策略。在正式写之前，有如下几种常见策略：

- 均分策略（i%6+1）
- 随机策略(Random)
- 影子电梯（深克隆电梯进行模拟运行，选择时间最短的那个）

在权衡利弊和性价比后，我果断选择了**均分策略**（事实证明性价比是最高的

因为他的实现非常简单，只需要在调度线程中新增一个全局静态变量`instructionNum`，在每次要进行调度的时候`+1`，再`%6+1`即可得到目标电梯的`id`，再进行分配即可：

```
private static int instructionNum = -1;
//choose
public int Choose(Person person) { //调度策略:?
    instructionNum++;
    //set person elevatorId instruction%6+1;
    //........
}
```

但是这时候我们又遇到了一个问题，万一此时电梯在`Reset`怎么办，因为题目要求电梯在`Reset Begin`后到`Reset End`的这段时间里**不能接受任何请求**。也许你会想，维护一个`resetFlag`变量进行判断即可，遇到`Reset`的电梯就跳过。但是这样做就可以了吗？试想这样一种情况：1-5号电梯都在`Reset`，只留下6号电梯空闲，而此时同时来了70个请求，那么是否这70个请求都会传递给6号电梯，当1-5号电梯结束`Reset`之后，一直保持**空闲**状态，而6号电梯忙的要死，这就会大大增加**RTLE风险**。因此为了解决这个问题，我们需要维护一个等待队列`wait`，也就是说，当请求分配时遇到相应电梯在`Reset`，**不进行跳过**，而是继续将请

求**分配给他但是不告诉他**（？，将请求加入`wait`队列中，一旦电梯完成`Reset`，将`wait`队列中的请求一股脑扔给电梯，这样就保证各个电梯都有活干（别想用请假逃避任务。

```
//choose 加入wait队列
if (requestTables.get(instructionNum % 6 + 1).isReseting()) {
    wait.get(instructionNum % 6 + 1).add(person);
    return 0;
}
```

```
//run 若不在Reset，分配wait中的请求给相应电梯
for (int i = 1; i <= 6; i++) {
    if (!requestTables.get(i).isReseting() && !wait.get(i).isEmpty()) {
        for (int j = 0; j < wait.get(i).size(); j++) {
            TimableOutput.println("RECEIVE-" + wait.get(i).get(j).getId() + "-"
+ i);
            requestTables.get(i).addRequest(wait.get(i).get(j));
        }
        wait.replace(i, new ArrayList<>());
    }
}
```

相应的，在调度线程的结束条件上也需要加上一条`isWaitEmpty()`

至此hw6的功能新增大体上完成了

优化

本次作业同样没有进行较好的优化，主要是时间上不支持。其实有很大的优化空间，主要就是在电梯线程的调度上，如果能实现影子电梯，或者像我舍友一样实现一种局部最优的形式，相比较均分策略都能实现很大的性能提分。本次作业最终在强测阶段不出意外的性能分很低。

bug分析

本次作业在强测中未发现bug，但是在互测中挨了一刀。后来观察数据发现，在本地跑似乎并没有问题，在原封不动再次提交代码进行bug修复后通过了。。。我想应该是**线程的不安全**问题，虽然成功进行了修复，但是代码应该还是存在问题的，但是如果功利来说.....我摆了

hack策略

本次作业由于是**多线程任务**，本身就可能存在很多**不确定因素**，因此hack起来格外容易。我的主要思路一是**卡时间**，二是想办法**触发死锁**。最终也证明这两个思路也是大家hack的主要思路，不过是否hack成功也得看运气，很多人本地几乎次次爆，但是交上去却一点问题没有，我想可能是上帝在掷骰子吧.....

代码复杂度分析

Complexity metrics	周日	14 4月 2024 15:28:31 CST		
Method	CogC	ev(G)	iv(G)	v(G)
Elevator.Elevator(int, RequestTable, RequestTabl	1	1	2	2
Elevator.open()	13	4	10	10
Elevator.reset()	10	5	5	5
Elevator.run()	17	3	7	10
InputHandler.InputHandler(RequestTable, HashMap<	0	1	1	1
InputHandler.run()	9	3	5	6
Main.main(String[])	3	1	4	4
Person.Person(int, int, int)	0	1	1	1
Person.getElevatorId()	0	1	1	1
Person.getFromFloor()	0	1	1	1
Person.getId()	0	1	1	1
Person.getToFloor()	0	1	1	1
Person.setElevatorId(int)	0	1	1	1
RequestTable.RequestTable()	1	1	2	2
RequestTable.addRequest(Person)	0	1	1	1
RequestTable.addRequestTable(RequestTable)	3	1	3	3
RequestTable.addReset(Reset)	0	1	1	1
RequestTable.addResetNum()	0	1	1	1
RequestTable.clear(int)	1	1	2	2
RequestTable.delRequest(int, int)	0	1	1	1
RequestTable.getOneRequest()	8	3	7	7
RequestTable.getRequestMap()	0	1	1	1
RequestTable.getReset(int)	0	1	1	1
RequestTable.isEmpty()	0	1	1	1
RequestTable.isEnd()	0	1	1	1
RequestTable.isResetEmpty()	0	1	1	1
RequestTable.isReseting()	0	1	1	1
RequestTable.setEndFlag()	0	1	1	1
RequestTable.setResetFlag(Boolean)	0	1	1	1
RequestTable.subResetNum()	0	1	1	1
RequestTable.waitRequest()	0	1	1	1
Reset.Reset(int, int, double)	0	1	1	1
Reset.getCapacity()	0	1	1	1
Reset.getElevatorId()	0	1	1	1
Reset.getSpeed()	0	1	1	1
Schedule.Choose(Person)	2	2	2	2
Schedule.Schedule(RequestTable, HashMap<Integer,	1	1	2	2
Schedule.isWaitEmpty()	3	3	2	3
Schedule.run()	22	5	12	13
Strategy.Strategy(RequestTable)	0	1	1	1
Strategy.canOpenForIn(int, int, int, int)	11	4	7	8
Strategy.canOpenForOut(int, HashMap<Integer, Arra	2	2	1	2
Strategy.canReset(int)	1	2	1	2
Strategy.getAdvice(int, int, int, int, HashMap<In	22	7	5	8
Strategy.requestForOriginDir(int, int)	7	3	3	7
TestMain.Pair.Pair(K, V)	0	1	1	1
TestMain.Pair.equals(Object)	4	3	3	5
TestMain.Pair.getFirst()	0	1	1	1
TestMain.Pair.getSecond()	0	1	1	1
TestMain.Pair.hashCode()	0	1	1	1
TestMain.Pair.toString()	0	1	1	1
TestMain.TimeInputStream.TimeInputStream(ArrayLis	4	3	3	3
TestMain.TimeInputStream.read()	7	3	3	5

TestMain.TimeInputStream.read(byte[], int, int)	10	8	2	10
TestMain.main(String[])	1	1	2	2
Class	OCavg	OCmax	WMC	
Advice	n/a	n/a	0	
Elevator	5.5	9	22	
InputHandler	3	5	6	
Main	4	4	4	
Person	1	1	6	
RequestTable	1.44	5	26	
Reset	1	1	4	
Schedule	4	9	16	
Strategy	3.17	7	19	
TestMain	2	2	2	
TestMain.Pair	1.33	3	8	
TestMain.TimeInputStream	5	8	15	
Package	v(G)avg	v(G)tot		
	2.8	154		
Module	v(G)avg	v(G)tot		
hw6	2.8	154		
Project	v(G)avg	v(G)tot		
project	2.8	154		

本次复杂度相较上次有更多的爆红，在`Schedule`的`run`方法中甚至全爆，我想还是因为在新增入功能的时候没有考虑架构应该具有高内聚低耦合的特点，这也是我需要改进的不足之处

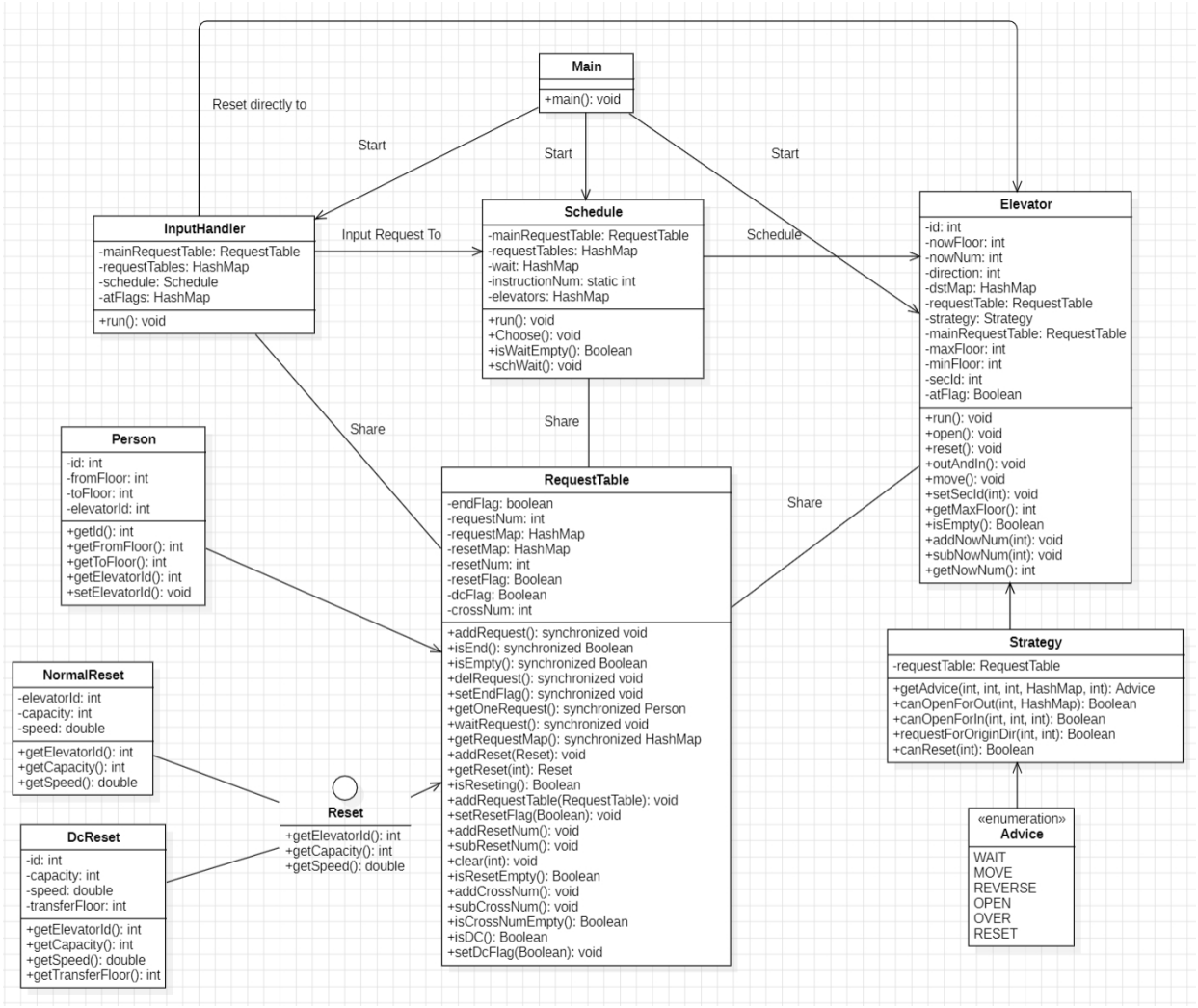
体会与感想

本次作业完成的可谓**惊心动魄**，在最后时刻涉险过关。本次作业的新增功能也让我头疼了很久，总体来说感觉是目前的**难度巅峰**，最后的性能分也没有拿多少还是很遗憾的，不过我已经尽了我的全力了。多线程的bug较为隐蔽，debug的方法也很匮乏，包括但不限于用print，调试的方法一般很难有用，所以在debug上花费了我大量的时间，但是最后好在结果还是好的，不仅学到了很多多线程的知识，还掌握了一定的多线程debug技巧，可谓收获颇丰，希望在以后的作业中再接再厉。

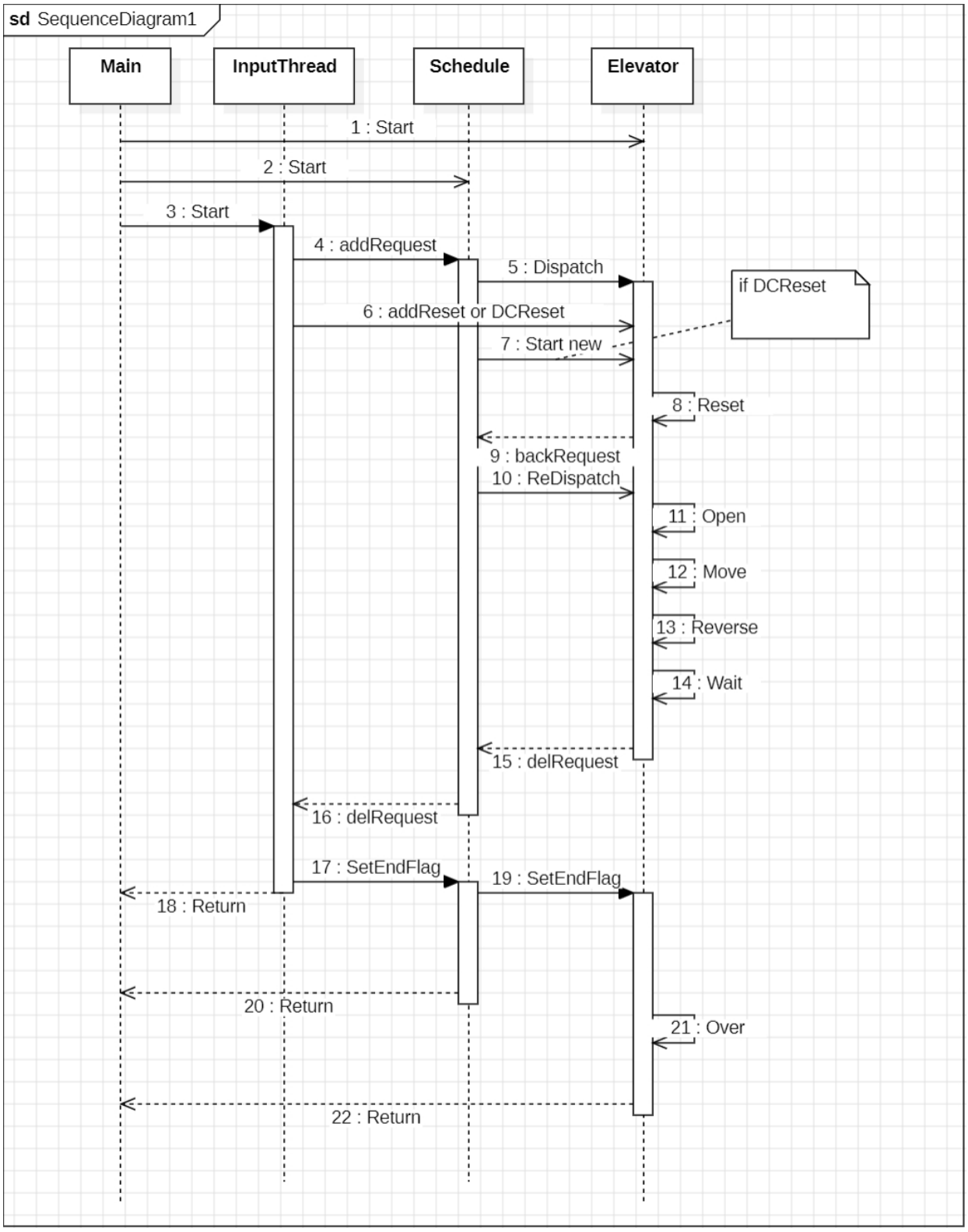
第七次作业

本次作业在前两次作业的基础上新增的**双轿厢**的设计。如果你仅仅是为了完成任务，不考虑性能分的话，完成此次作业还是较为简单的，无非只需要考虑以下几个问题：`ResetDC`类指令与之前的`Reset`指令的关系，可否用**接口**将他们联系在一起？`ResetDC`之后电梯该进行**怎样的操作**？双轿厢电梯到达**换乘层**该进行怎样的操作？如何确保AB两个轿厢**不能同时**到达换乘层？只需要解决以上问题，我们便能完成基本的任务。当然你还可以考虑，怎样设计请求的调度可以最大化发挥双轿厢电梯的优势？当然由于本周有蓝桥杯，我并没有时间考虑这个。此外我还犯了个让我追悔莫及的错误，那便是修改了原先的均分策略，改用人数+请求最少的分配策略，这让我的程序出现了RTLE的错误，在互测中被狠狠hack了，后悔啊！

UML类图


Main
+main(): void
Reset directly to
Start
Start
Start
InputHandler
-mainRequestTable: RequestTable
-requestTables: HashMap
-schedule: Schedule
-atFlags: HashMap
+run(): void
Input Request To
Schedule
-mainRequestTable: RequestTable
-requestTables: HashMap
-wait: HashMap
-instructionNum: static int
-elevators: HashMap
+run(): void
+Choose(): void
+isWaitEmpty(): Boolean
+schWait(): void
Schedule
Elevator
-id: int
-nowFloor: int
-nowNum: int
-direction: int
-dstMap: HashMap
-requestTable: RequestTable
-strategy: Strategy
-mainRequestTable: RequestTable
-maxFloor: int
-minFloor: int
-secId: int
-atFlag: Boolean
+run(): void
+open(): void
+reset(): void
+outAndIn(): void
+move(): void
+setSecId(int): void
+getMaxFloor(): int
+isEmpty(): Boolean
+addNowNum(int): void
+subNowNum(int): void
+getNowNum(): int
Share
Share
Share
Person
-id: int
-fromFloor: int
-toFloor: int
-elevatorId: int
+getId(): int
+getFromFloor(): int
+getToFloor(): int
+getElevatorId(): int
+setElevatorId(): void
RequestTable
-endFlag: boolean
-requestNum: int
-requestMap: HashMap
-resetMap: HashMap
-resetNum: int
-resetFlag: Boolean
-dcFlag: Boolean
-crossNum: int
+addRequest(): synchronized void
+isEnd(): synchronized Boolean
+isEmpty(): synchronized Boolean
+delRequest(): synchronized void
+setEndFlag(): synchronized void
+getOneRequest(): synchronized Person
+waitRequest(): synchronized void
+getRequestMap(): synchronized HashMap
+addReset(Reset): void
+getReset(int): Reset
+isReseting(): Boolean
+addRequestTable(RequestTable): void
+setResetFlag(Boolean): void
+addResetNum(): void
+subResetNum(): void
+clear(int): void
+isResetEmpty(): Boolean
+addCrossNum(): void
+subCrossNum(): void
+isCrossNumEmpty(): Boolean
+isDC(): Boolean
+setDcFlag(Boolean): void
Strategy
-requestTable: RequestTable
+getAdvice(int, int, int, HashMap, int): Advice
+canOpenForOut(int, HashMap): Boolean
+canOpenForIn(int, int, int): Boolean
+requestForOriginDir(int, int): Boolean
+canReset(int): Boolean
NormalReset
-elevatorId: int
-capacity: int
-speed: double
+getElevatorId(): int
+getCapacity(): int
+getSpeed(): double
Reset
+getElevatorId(): int
+getCapacity(): int
+getSpeed(): double
DcReset
-id: int
-capacity: int
-speed: double
-transferFloor: int
+getElevatorId(): int
+getCapacity(): int
+getSpeed(): double
+getTransferFloor(): int
«enumeration»
Advice
WAIT
MOVE
REVERSE
OPEN
OVER
RESET

协作图



代码架构分析

`DcReset`指令分析

本次作业新增了一个双轿厢电梯的重置要求，看起来是新增实则不然，因为我们已经实现了`Reset`指令的功能，`DcReset`再怎么样，他也是`Reset`嘛，于是我们就可以利用第一单元学到的知识，用一个`Reset`接口联系两种`Reset`指令，这样我们就**不需要再更改**原先的`Reset`部分的代码了，做到了增量式开发的要求。

```
public interface Reset {
    int getElevatorId();

    int getCapacity();

    double getSpeed();
}
```

同样的我们从**输入处理线程**讲起。当遇到`DcReset`指令时，同样直接传递给对应的电梯执行`Reset`操作（双轿厢的重置同样需要把电梯内的人out出来，这部分可以认为我们已经实现了），之后我们需要新增一个B电梯，将原先的电梯认为是A电梯，可以用一个`secId`变量进行区分。

```
//InputThread
this.schedule.create(elevatorId, transferFloor,
                                capacity, speed, atFlags.get(elevatorId));
```

```
//Schedule
public void create(int elevatorId, int transferFloor, int capacity,
                        double speed, Boolean atFlag) {

    //...创建新电梯
    elevator.setSecId(2);//设置secId以区分轿厢
    elevators.put(elevatorId + 6, elevator);
    elevator.start();//启动新增电梯线程
}
```

再来看电梯遇到`DcReset`指令后该如何操作，这里我们在`reset()`方法中新增一部分即可：

```
//Elevator/reset()
if (reset instanceof DcReset) {
    this.maxFloor = ((DcReset) reset).getTransferFloor();
    requestTable.setDcFlag(true);
    this.nowFloor = ((DcReset) reset).getTransferFloor() - 1;
    this.setSecId(1);
}
```

这部分是对原先存在的电梯进行操作，关键思想就是使得他成为双轿厢的**A轿厢**。

而在`outAndIn()`方法中，电梯里未到达目的地的人出去后还是把请求**扔回总表**。但这样的话原先针对六部电梯的均分策略将**不再有效**，因此为了减少改动量，节约时间，我牺牲了性能，将其扔回总表但已经设定其待分配的电梯为对应电梯的B轿厢。

```
//outAndIn()
if (secId == 1) {
    newPerson.setElevatorId(id + 6);
} else if (secId == 2) {
    newPerson.setElevatorId(id - 6);
}
mainRequestTable.addRequest(newPerson);
```

结束条件

此时，跟上一次作业同样的问题显现了，并且似乎更难解决。那就是，只要双轿厢电梯运行没结束，总存在乘客换乘的可能，那么调度线程就**始终不能结束**，我们当然可以让其在没有工作时`wait`，但是何时唤醒又成了一个问题，最开始我并没有考虑到这个问题，但是第一次提交所有的结果都是ctle，于是我发现存在轮询，也就是我的`schedule`线程始终没有结束也没有`wait`一直在**轮询**。

那么为了解决这个问题，我们就需要从根源出发，借鉴hw6的实现方式。这里是由于双轿厢电梯中存在从A到B的换乘请求，那么我们就为这类请求的数量维护一个共享变量`crossNum`即可，只有当`crossNum`也为0的时候，调度线程才能真正结束,否则进行等待：

```
if (mainRequestTable.isEnd()
                    && mainRequestTable.isResetEmpty() && isWaitEmpty()
                    && mainRequestTable.isCrossNumEmpty() &&
mainRequestTable.isEmpty()) {
    for (Integer key : requestTables.keySet()) {
        requestTables.get(key).setEndFlag();
    }
    //System.out.println("return");
    return;

} else if (mainRequestTable.isEnd()
        && mainRequestTable.isResetEmpty() && isWaitEmpty()
        && !mainRequestTable.isCrossNumEmpty() && mainRequestTable.isEmpty()) {
    //wait
}
```

这样就不会出现轮询的问题啦。提示：如果出现轮询，可以参照助教给出的debug方法，用`TimableOutput.println("************");`判断轮询出现的线程和分支。

抢锁

而为了让两个轿厢不能同时进入换乘层，我们自然而然可以想到锁，他的作用就是在同一时刻只有一个线程可以抢到锁并开始执行，直到执行完毕释放锁，另一个线程抢到锁继续执行。于是我们为电梯的移动增加一个**对象锁**，其监视对象为atFlag，又双轿厢电梯的AB轿厢共享：

```
//Elevator/run
else if (advice == Advice.MOVE) {
    if (secId == 1 && nowFloor == maxFloor - 1 && direction == 0
            || secId == 2 && nowFloor == minFloor + 1 && direction == 1) {//将要
到达换乘层
        synchronized (atFlag) {//抢锁
```

```
            move();
            this.atFlag = true;
            outAndIn();

        }//释放锁
    } else {//否则正常移动
        move();
    }
}
```

调度策略

调度策略没有大改，只需要在分配的时候把出发楼层大于换乘层的给B轿厢，小于的给A轿厢即可，这里就不赘述了。（但是我手贱在提交前改了我的分配策略.....好后悔）

至此我们基本完成hw7的任务了

优化

本次作业没精力做任何优化了，唯一自以为的优化还成了bug，希望以后的优化一定要谨慎谨慎再谨慎，千万以正确性为第一优先级。

bug分析

本次作业的bug发生在`Schedule`类中，具体来说是因为调度策略的修改引发了bug，我采用了**人数+请求最少**的分配策略，但是当电梯同时进行调度分配的时候，最少的那一个是**同一部电梯**，这也就造成了只有一部电梯在跑的情况.....还是%6好。

hack策略

还是利用并发性大数据卡时间，永远的神！

代码复杂度分析

Complexity metrics	周日	14 4月 2024 17:06:54 CST		
Method	CogC	ev(G)	iv(G)	v(G)
DcReset.DcReset(int, int, double, int)	0	1	1	1
DcReset.getCapacity()	0	1	1	1
DcReset.getElevatorId()	0	1	1	1
DcReset.getSpeed()	0	1	1	1
DcReset.getTransferFloor()	0	1	1	1
Elevator.Elevator(int, RequestTable, RequestTa	1	1	2	2
Elevator.Elevator(int, RequestTable, RequestTa	1	1	2	2
Elevator.addNowNum(int)	0	1	1	1
Elevator.getAtFlag()	0	1	1	1
Elevator.getMaxFloor()	0	1	1	1
Elevator.getMinFloor()	0	1	1	1
Elevator.getNowNum()	0	1	1	1
Elevator.getOut()	11	1	7	7
Elevator.isEmpty()	0	1	1	1
Elevator.move()	5	1	3	4
Elevator.open()	0	1	1	1
Elevator.openForIn()	19	4	12	12
Elevator.outAndIn()	20	1	10	12
Elevator.reset()	11	5	6	6
Elevator.run()	20	3	9	16
Elevator.setAtFlag(Boolean)	0	1	1	1
Elevator.setSecId(int)	0	1	1	1
Elevator.subNowNum(int)	0	1	1	1
InputHandler.InputHandler(RequestTable, HashMa	0	1	1	1
InputHandler.run()	10	3	6	7
Main.main(String[])	4	1	5	5
NormalReset.NormalReset(int, int, double)	0	1	1	1
NormalReset.getCapacity()	0	1	1	1
NormalReset.getElevatorId()	0	1	1	1
NormalReset.getSpeed()	0	1	1	1
Person.Person(int, int, int)	0	1	1	1
Person.getElevatorId()	0	1	1	1
Person.getFromFloor()	0	1	1	1
Person.getId()	0	1	1	1
Person.getToFloor()	0	1	1	1
Person.setElevatorId(int)	0	1	1	1
RequestTable.RequestTable()	1	1	2	2
RequestTable.addCrossNum()	0	1	1	1
RequestTable.addRequest(Person)	0	1	1	1
RequestTable.addRequestTable(RequestTable)	3	1	3	3
RequestTable.addReset(Reset)	0	1	1	1
RequestTable.addResetNum()	0	1	1	1
RequestTable.clear(int)	1	1	2	2
RequestTable.delRequest(int, int)	0	1	1	1
RequestTable.getOneRequest()	8	3	7	7
RequestTable.getRequestMap()	0	1	1	1
RequestTable.getRequestNum()	0	1	1	1
RequestTable.getReset(int)	0	1	1	1
RequestTable.isCrossNumEmpty()	0	1	1	1
RequestTable.isDC()	0	1	1	1
RequestTable.isEmpty()	0	1	1	1
RequestTable.isEnd()	0	1	1	1

RequestTable.isResetEmpty()	0	1	1	1
RequestTable.isReseting()	0	1	1	1
RequestTable.setDcFlag(Boolean)	0	1	1	1
RequestTable.setEndFlag()	0	1	1	1
RequestTable.setResetFlag(Boolean)	0	1	1	1
RequestTable.subCrossNum()	0	1	1	1
RequestTable.subResetNum()	0	1	1	1
RequestTable.waitRequest()	0	1	1	1
Schedule.Choose(Person)	23	2	8	8
Schedule.Schedule(RequestTable, HashMap<Intege	1	1	2	2
Schedule.create(int, int, int, double, Boolean	0	1	1	1
Schedule.getNum()	5	1	2	4
Schedule.isWaitEmpty()	3	3	2	3
Schedule.run()	27	7	16	18
Schedule.schWait()	38	1	11	11
Strategy.Strategy(RequestTable)	0	1	1	1
Strategy.canOpenForIn(int, int, int, int)	11	4	7	8
Strategy.canOpenForOut(int, HashMap<Integer, A	2	2	1	2
Strategy.canReset(int)	1	2	1	2
Strategy.getAdvice(int, int, int, int, HashMap	22	7	5	8
Strategy.requestForOriginDir(int, int)	7	3	3	7
TestMain.Pair.Pair(K, V)	0	1	1	1
TestMain.Pair.equals(Object)	4	3	3	5
TestMain.Pair.getFirst()	0	1	1	1
TestMain.Pair.getSecond()	0	1	1	1
TestMain.Pair.hashCode()	0	1	1	1
TestMain.Pair.toString()	0	1	1	1
TestMain.TimeInputStream.TimeInputStream(Array	4	3	3	3
TestMain.TimeInputStream.read()	7	3	3	5
TestMain.TimeInputStream.read(byte[], int, int	10	8	2	10
TestMain.main(String[])	1	1	2	2

Class	OCavg	OCmax	WMC
Advice	n/a	n/a	0
DcReset	1	1	5
Elevator	3.39	12	61
InputHandler	3.5	6	7
Main	5	5	5
NormalReset	1	1	4
Person	1	1	6
RequestTable	1.33	5	32
Schedule	5.29	10	37
Strategy	3.17	7	19
TestMain	2	2	2
TestMain.Pair	1.33	3	8
TestMain.TimeInputStream	5	8	15

Package	v(G)avg	v(G)tot
	2.87	238

Module	v(G)avg	v(G)tot
hw7	2.87	238

Project	v(G)avg	v(G)tot

project		2.87	238	

可以看到本次作业包括`Elevator`和`Schedule`在内的多个方法爆红了，我想也是正常的，因为本次作业我的架构十分混乱，方法也只是胡乱的叠加，复杂度理所应当会很高，并且本次作业可扩展性已经很低了，如果再来一次迭代我将无从下手。这也给我一个警示，在程序架构的过程中还是应该尽可能保证程序架构的清晰明了，留出可扩展的空间。

体会与感想

本次作业让我感到十分**遗憾**，明明可以正确完成的，却因为想着优化一下而优化出bug了，而且因为本周事情较多，主要在准备蓝桥杯，所以并没有进行充分的测试，也就造成了bug。这提示我在以后的优化中一定要确保程序的正确性，希望以后不要再犯这样的错误，吃一堑长一智。

第二单元感想

终于来到博客周，第二单元终于要结束了！

私以为本单元是整个oo课的难度巅峰，总体来说还是非常有挑战性的，但是同样收获也非常多，我们不仅学习了多线程、锁、线程安全、读写锁、原子变量等等新知识点，还接洽了**OS**的**进程与线程**的知识，可见多线程的学习是cs学生的**重中之重**，因此本单元的学习相信会给我打下一个坚实的多线程基础。