OO_Unit2总结

同步块的设置和锁的选择

这三次作业中均选择了`synchronized`关键字，未使用读写锁。既使用`synchronized`修饰方法，也用其来包裹语句块。主要在候程表类`RequestQueue`类中使用，该类中所有方法均使用`synchronized`修饰，因为该类在“生产者-消费者”模型中充当托盘的角色，会被调度器线程和电梯线程同时访问。在第三次作业中新增设的类`TransferFloor`类中也使用了`synchronized`，用来保证双轿厢同步改造。
至于处理语句之间的关系，采用`wait()`和`notifyAll`来相互配合，实现线程之间的相互合作，未使用`notify`，因为可能会有多个线程在等待，`notify`可能会唤醒一个不满足当前条件的线程，而其他满足条件的线程会继续等待，可能会导致程序错误或者死锁。

调度器设计

调度器与线程之间的交互

调度器类负责对乘客的请求进行调度，让乘客乘坐合适的电梯。为了实现调度器能够与线程之间产生交互，利用候程表类`RequestQueue`建立主请求表`mainRequestQueue`和为每个电梯线程分发请求的副请求表列表`subRequestQueues`，输入线程负责从输入中解析请求，并将请求传递给`mainRequestQueue`，而调度器负责从`mainRequestQueue`中获取请求，包括乘客请求、临时调度请求、双轿厢改造请求，然后将这个请求传递给`subRequestQueues`，由`subRequestQueues`负责将请求传递给具体的电梯线程，然后电梯线程根据具体的请求开始运作。

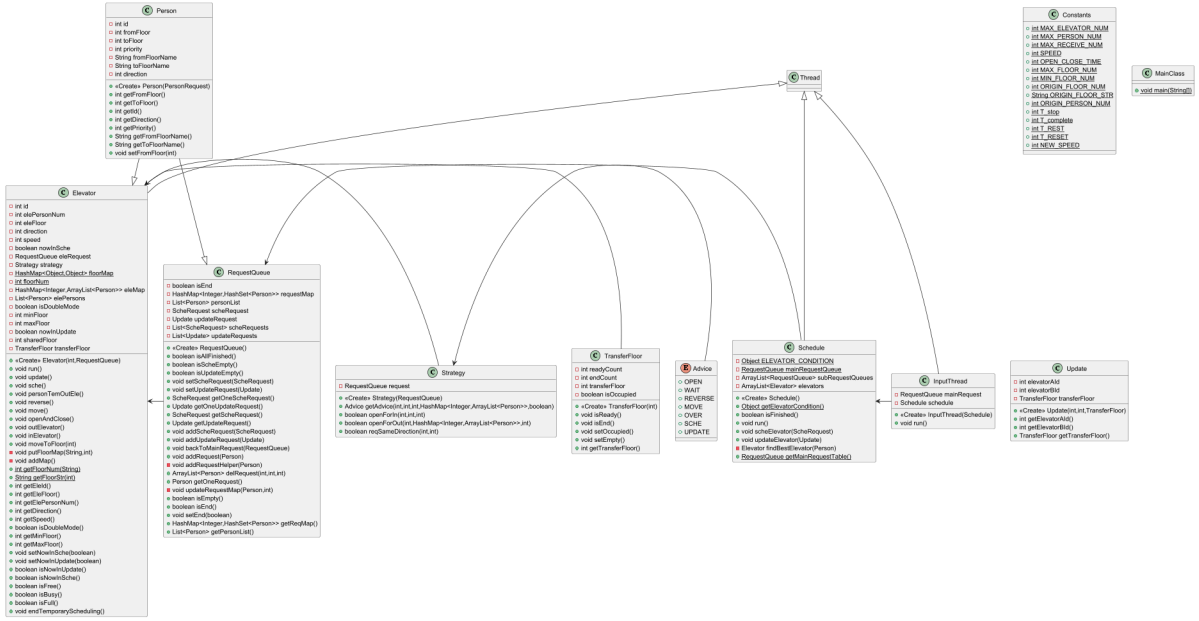
调度策略

第一次作业指定了乘客要乘坐的电梯，因此不需要我们进行调度。
第二次作业不再指定电梯，需要我们自行设计调度方案。我并没有使用影子电梯，因为太复杂了，也没有使用随机分配和平均分配（mod 6），而是采用最容易想到的方式。首先，过滤掉正在处于临时调度状态的电梯和请求数达到上限的电梯（10），然后在剩余的电梯中寻找离乘客最近的电梯，如果最近的电梯大于1部，则选择乘客数最少的电梯。这个调度策略让我在第二次作业正确的样例中得分在95~96，还可以接受，但是现在来看，可以增加一个限制：应该选择离乘客最近的且运行方向与乘客前往方向相同的电梯，这样可以节省一些耗电量。
第三次作业则是对电梯的运行范围进行了限制，由于改造后的电梯运行速度变快，所以在调度的时候并没有对单轿厢电梯和双轿厢电梯进行区分。首先，过滤掉正在处于临时调度状态的电梯和正在处于双轿厢改造的电梯以及请求数达到上限的电梯（10），然后优先选择乘客起始楼层在运行范围之内的电梯，在选择离乘客最近的电梯。

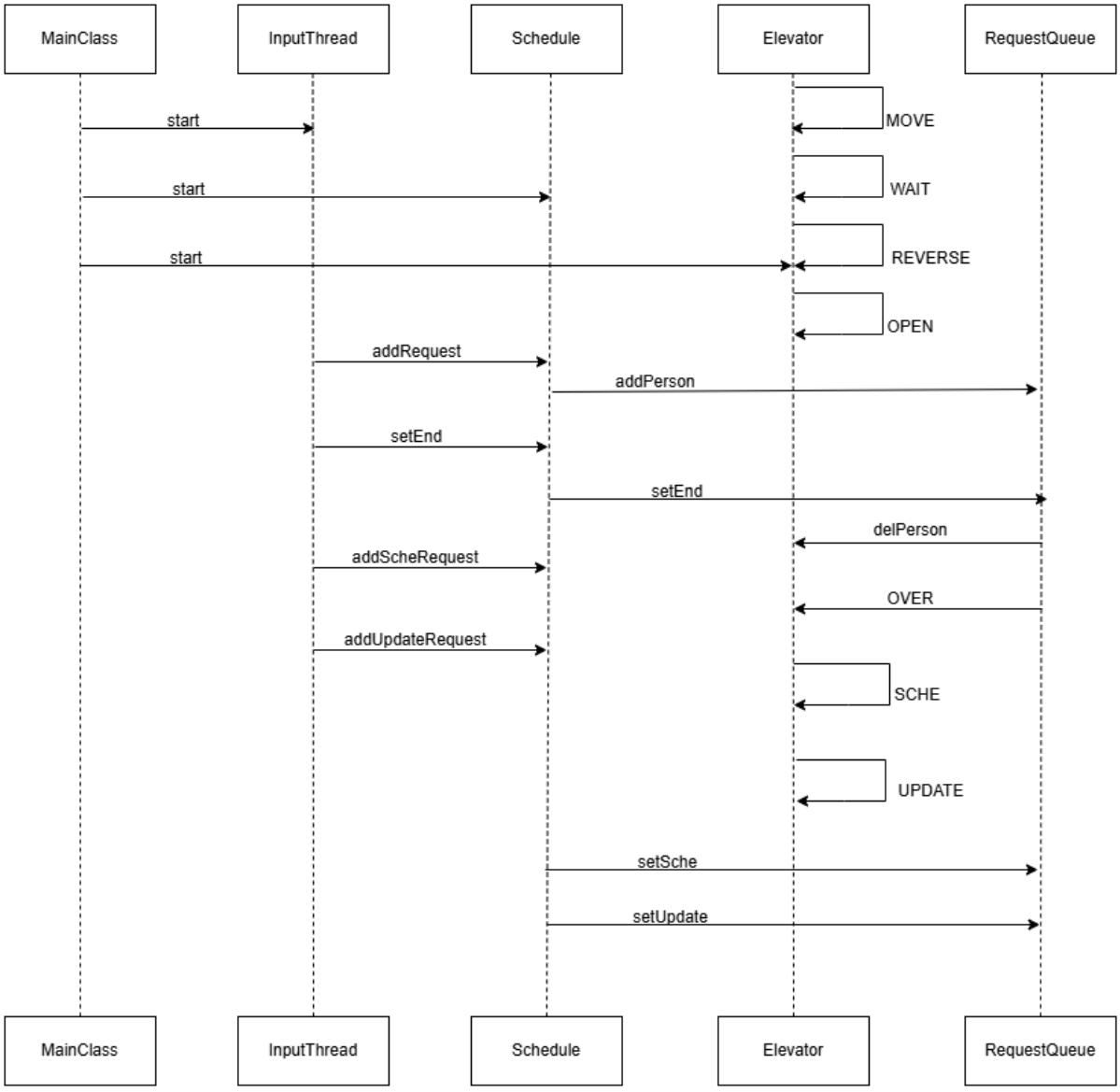
架构设计

后两次作业均在第一次作业的基础上迭代而成，采用LOOK运行策略，设置MOVE、OPEN、REVERSE、WAIT、OVER五个动作，新增加的临时调度和双轿厢改造也都看成是电梯的动作：SCHE和UPDATE。

程序的UML类图如下：



协作图如下：



稳定和易变的内容

稳定的部分：

- “生产者-消费者”模式没变，各个线程各司其职
- 程序的基本架构没变，都是由生产者产生请求，然后电梯将这些请求转化为一系列动作，使用LOOK算法实现电梯运行。

易变的部分：

- 电梯动作发生变化，增加SCHE和UPDATE
- 调度策略变化，从不需要调度策略到需要自行分析性能较好的调度策略

双轿厢改造

第三次作业要求实现电梯双轿厢改造，包括两个方面：两个电梯同时进行改造和两个轿厢不碰撞。

同时进行改造

为了实现两部电梯同时进行改造，新建一个`TransferFloor`类，并在要进行改造的两部电梯中共享同一个该类的实例，为了保证线程安全，该类中所有方法均使用`synchronized`关键字修饰。在该类中实现两个用于实现同步改造的方法：`isReady()`和`isEnd()`，实现如下：

```
    public synchronized void isReady() {
        readyCount++;
        if (readyCount == 2) {
            notifyAll();
        } else {
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
    public synchronized void isEnd() {
        endCount++;
        if (endCount == 2) {
            notifyAll();
        } else {
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
```

当电梯进行`UPDATE`这个动作时，先将乘客清空，然后执行共享实例`transferFloor`的`isReady()`方法，`readyCount`开始计数，先开始进行改造的电梯会因为`readyCount`小于2而`wait()`，当两部电梯都开始改造后，`readyCount`等于2，将`wait()`的电梯唤醒，两部电梯便同时开始改造。同时为了保证只输出一次`UPDATE-BEGIN`，我们可以只让A电梯输出。
显然，电梯同步结束改造和同步开始该做逻辑相同，不再赘述。

```
    public void update() {
        //清空乘客
        transferFloor.isReady();
        if (id == elevatorAId) {
            TimableOutput.println("UPDATE-BEGIN-" + elevatorAId + "-" +
elevatorBId);
        }
        //更新电梯参数
        transferFloor.isEnd();
        if (id == elevatorAId) {
            TimableOutput.println("UPDATE-END-" + elevatorAId + "-" +
elevatorBId);
        }
    }
```

双轿厢防撞

仍然使用共享对象`transferFloor`实现，在`TransferFloor`类中设置布尔信号`isOccupied`，在电梯要进入换乘楼层之前会先判断`isOccupied`是否真，也就是电梯是否被占用，如果为真，则该电梯会进行`wait()`，反之则会进入换乘楼层同时将`isOccupied`设为真。当电梯离开换乘楼层时会将`isOccupied`设为假并唤醒正在等待的电梯，以便其他电梯可以进入换乘楼层。
同时为了避免出现一部电梯长时间呆在换乘楼层导致另一部电梯无法移动的情况，需要在电梯的`MOVE`动作中进行特判，如果电梯当前所在楼层是换乘楼层，则会自动远离共享楼层一层。

```
    public synchronized void setOccupied() {
        if (isOccupied) {
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        isOccupied = true;
    }
    public synchronized void setEmpty() {
        isOccupied = false;
        notifyAll();
    }
```

Bug分析

出现过的bug

第一次作业强测和互测均无bug。
第二次作业出现了两个bug。第一个是电梯开始进行临时调度后依旧会Receive，原因是我为电梯设置了一个临时调度状态的布尔属性，为真时说明电梯已经处于临时调度状态，方便调度器进行调度的时候忽略正在进行临时调度的电梯，但是我将这个属性设置的太晚，直到电梯accept到sche请求后清空完乘客后才设置为真，在一些极端情况下Receive的输出再输出`SCHE-BEGIN`之后，解决方法是当电梯开始临时调度这个动作之后随即将这个状态设置为真。
第二个bug与我的调度策略有关，我将一部电梯同一时间可接受的最多请求数量设置为6，再加上其他限制（比如临时调度），当不存在合适的电梯时该乘客会进行等待，且每秒会刷新一次，直到出现合适的电梯出现。由于请求数量设置的过少，结果在强测的强力数据之下，部分乘客等待时间过长导致程序出

现了CTLE，将一部电梯的最多请求数量设置为10，即便这部电梯已经满员也可以接收到请求便可以解决这一问题。
第三次作业出现的bug是死锁。刚开始做第三次作业时我的想法是两部电梯互相传递对方的信息来实现同步改造，不出所料地是我的程序成功的陷入死锁卡死，在观摩了大量前人的博客之后换成现在共享`TransferFloor`类之后这个bug得到了解决。

deBug方法

1. 在代码关键位置使用print大法输出日志，根据输出的电梯情况对bug进行勘误，但是看其他大佬提到使用print会影响多线程程序的运行，所以更多地使用第二种方法。
2. IntelliJ IDEA自带一个多线程分析工具——转储线程，这个工具在处理死锁情况时非常有效，这个功能可以抓取当前时刻程序中各个线程的状态，再配合第一种方法可以很好的进行deBug。

心得体会

线程安全

这是我第一次接触多线程编程，经过这三次作业的折磨，也算是对多线程有了初步的了解。这一单元要保证的线程安全，无非要实现同步互斥和防止死锁。实现同步互斥可以使用读写锁的方法，也可以使用`synchronized`关键字一路走到头，但是`synchronized`关键字并不是无脑加的，只需要在需要进行同步互斥的代码块使用`synchronized`，否则不仅会导致性能下降，还有可能会导致死锁。

层次化设计

这一单元的三次作业均是迭代写成，没有经历第一单元时的中途重构，这得益于第一次作业的好的架构，或者说是本单元第一次上机时代码有一个好的架构，因为我是模仿着上机代码的"生产者-消费者"模型写的，调度器线程和输入线程作为生产者，候程表作为托盘，电梯线程作为消费者，三者合作实现作业要求，而后两次作业增加的临时调度和双轿厢改造都可以看成和开门或者移动一样的电梯动作，着重修改电梯线程即可。
也希望在下一单元我也可以有一个好的架构，避免重构。