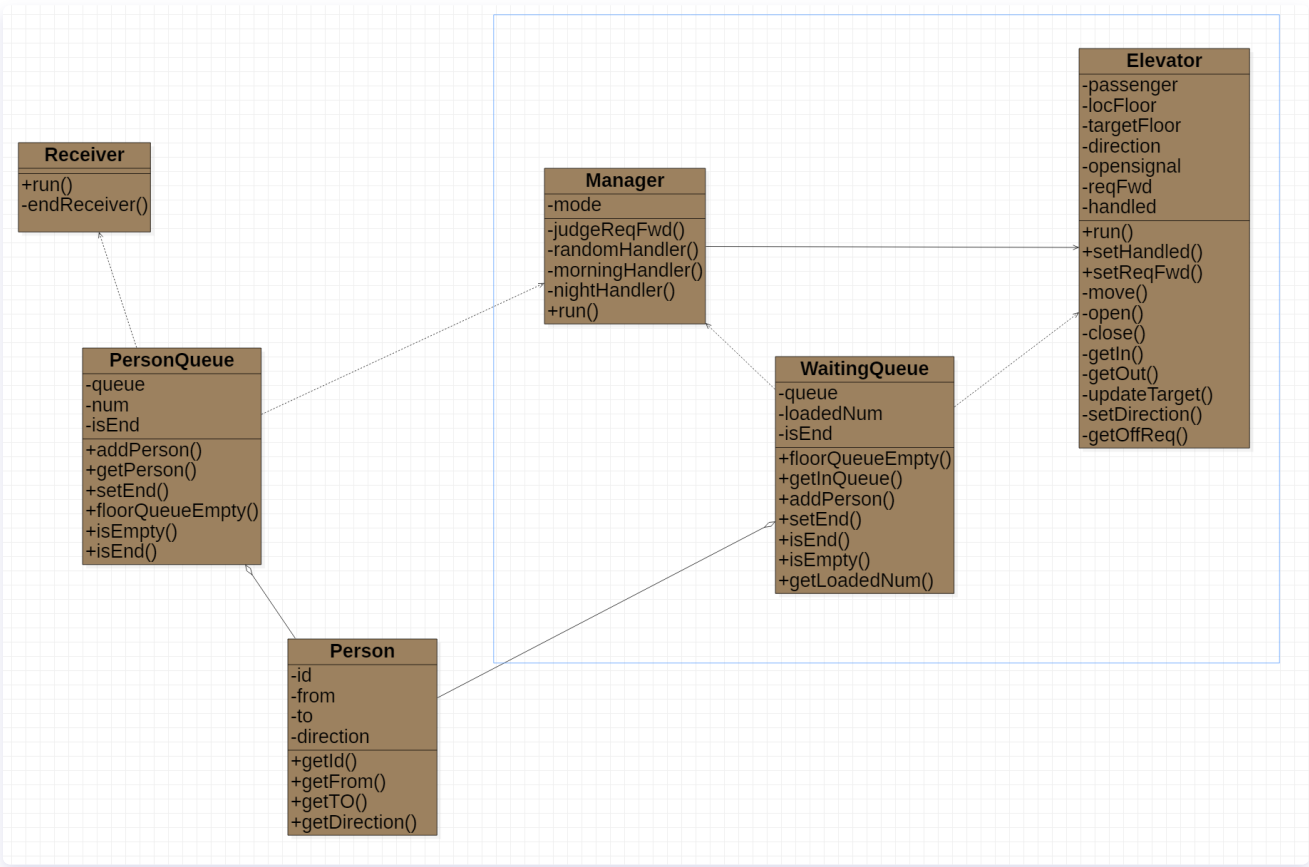
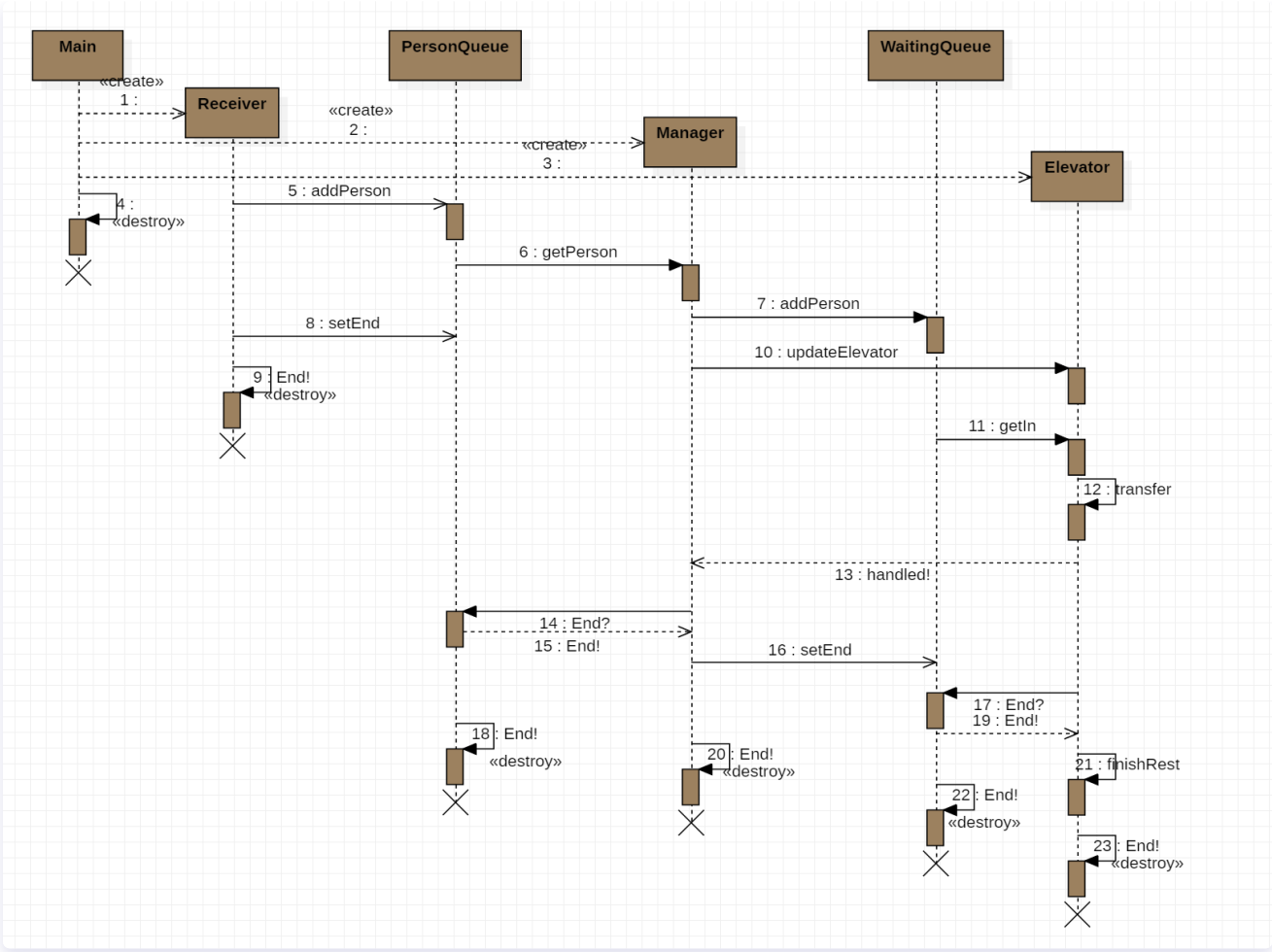
unit 2 summary

一、第一次作业

1.UML



2.Sequence Diagram



3.同步块设置与锁处理

采用了**生产者-消费者模式**，用共享对象来连接不同的线程。

- 第一次作业中，我有三个线程: `Receiver` , `Manager` , `Elevator` ,用 `PersonQueue` `WaitingQueue` 分别来连接 `Receiver` 和 `Manager` ， `Manager` 和 `Elevator`
  整个程序中的加锁操作只针对有可能发生数据竞争的连个共享对象： `PersonQueue` , `WaitingQueue`
  加锁是为了保证之后位于同步块内的代码执行时是原子性的，不会被另外的线程所干扰，防止共享对象发生数据冲突
- 第一次作业只涉及到一部电梯的运行，因此可以准确的把握电梯的运行状态，而为了把握准确状态加锁需要尽量细致，即几乎各种操作都需要加锁，所以干脆直接写了一个很大的同步块
  具体来说：
  `Receiver` 需要将输入的请求放进 `PersonQueue` , `Manager` 需要将请求从 `PersonQueue` 移到 `Elevator` ，当加人、减人、设置结束信号和查询人数等操作时需要对操作对象加锁(因为得到的电梯状态可以尽量准确），相关操作过程需放在同步块中

4.调度器设计分析

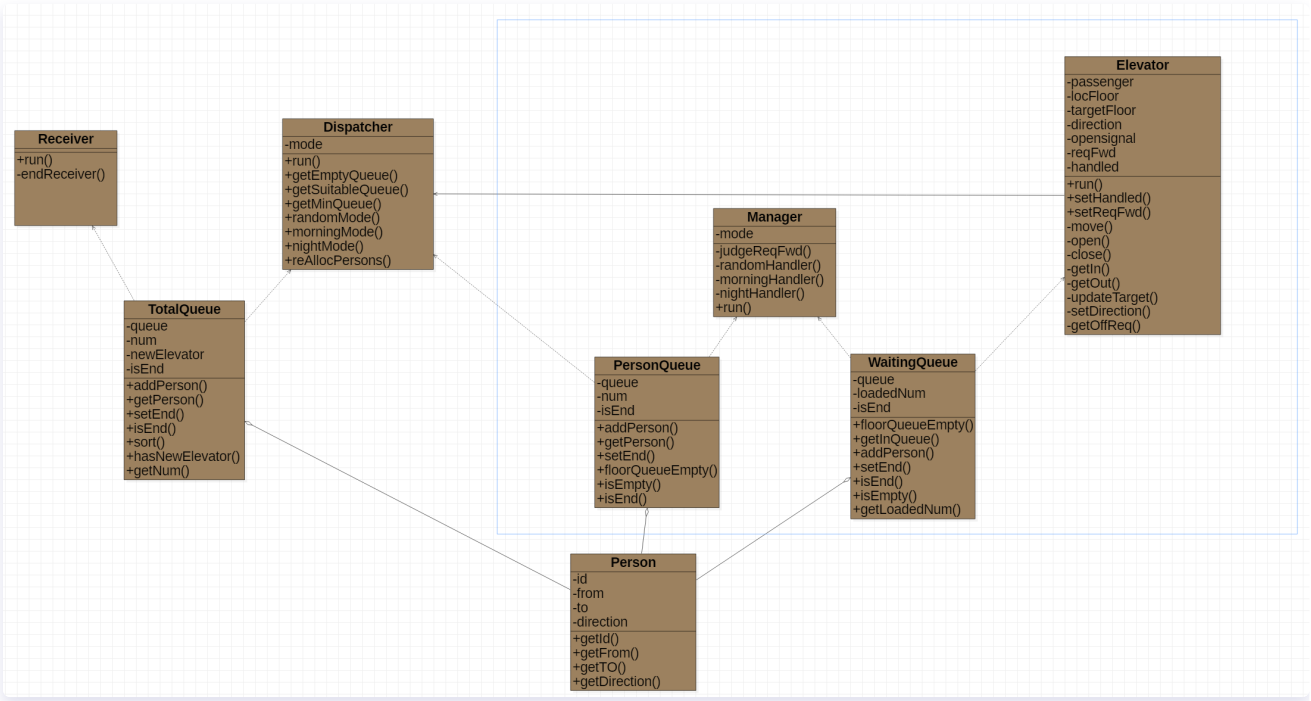
本次作业中为了后面的可扩展性，除了输入线程和电梯线程两个必要线程，还增加了一个处理线程 `Manager` （可以看作是本次作业的调度器），他负责把输入线程发来的请求进行处理，选择当前能符合电梯捎带策略的请求，然后把请求发给电梯的等待队列，再由电梯负责**消化**请求
由于只涉及到一部电梯，整个架构比较简单，为了能最大化电梯性能，需要保证处理器的分配是绝对正确的，那么就需要处理器能够得到**绝对精准**的当前电梯状态，所以我让处理线程和电梯线程两个线程的行动一直保持同步，即电梯每运动一次，处理器就读取一次电梯的最新状态，并根据最新的电梯状态进行请求分配，分配完成后电梯**才能**继续往正确的方向运动，也即是说，但请求未分配完成时，电梯的运行需要处理器来支撑，当请求都分配完成后，电梯能自行处理已经分配到电梯等待队列的所有剩余请求，处理线程的生命周期小于等于电梯线程的生命周期
输入请求完成后，会向 `PersonQueue` 发送一个结束的标志， `Manager` 读到之后传递结束标志到 `WaitingQueue` 并自行结束， `Elevator` 读到结束标志后，处理完剩余请求后自行结束线程，也就是说各个线程的结束都是靠着**标志+特定状态**才能结束的

5.可扩展性分析

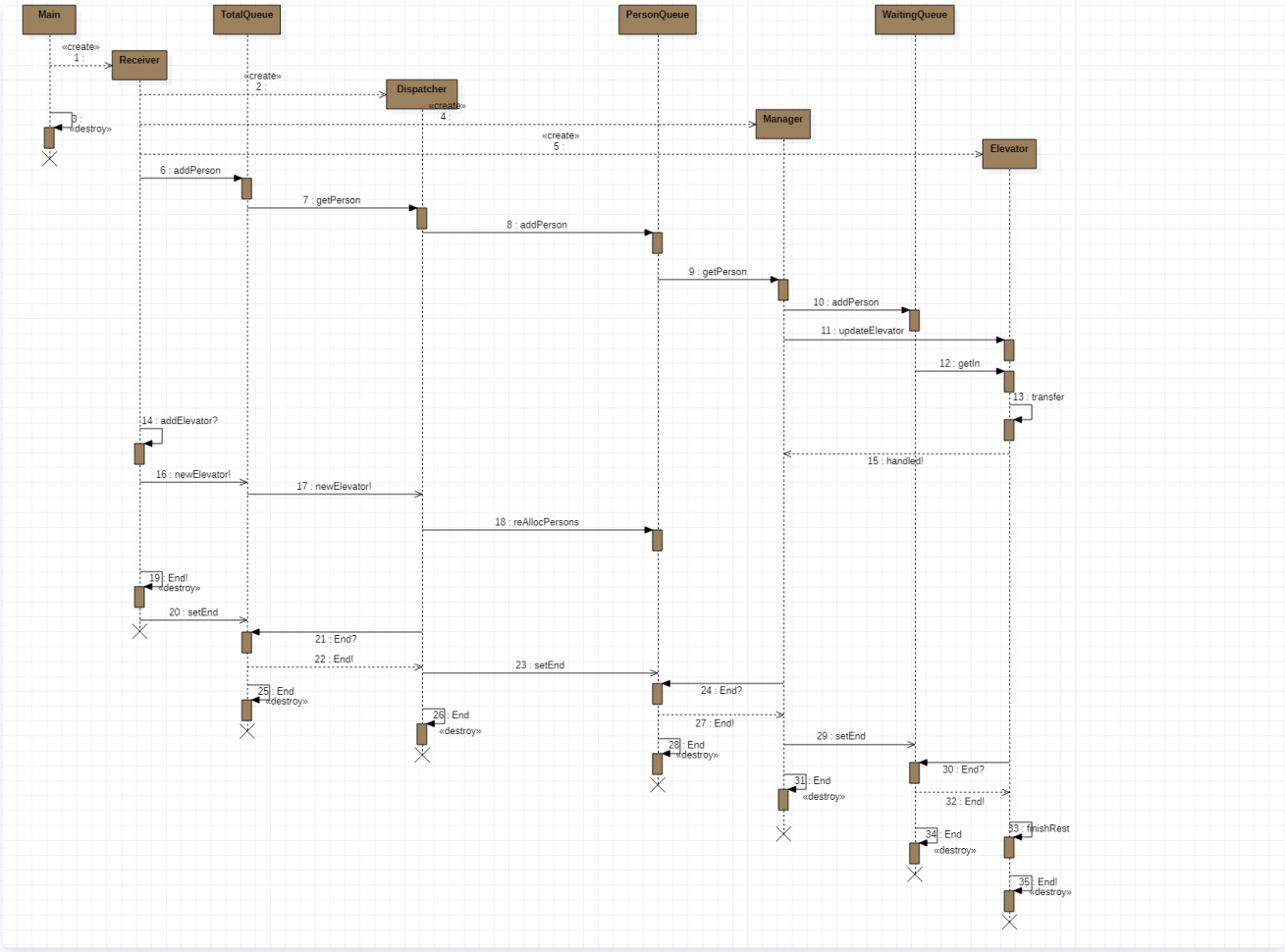
由于考虑了第二次作业可能会出现的多部电梯同时运行，需要调度器进行分配的情况，提前进行了调度器的部署，而不是普通的一个生产者，一个托盘，一个消费者这种简单的架构，具有良好的可扩展性

二、第二次作业

1.UML



2.Sequence Diagram



3.同步块设置与锁处理

相比第一次作业，第二次作业只是新增的需求是：**支持多部相同的电梯同时工作**，考虑到架构的扩展性，那么在第一次作业的基础上我就只用加一个更大的调度器，把之前 `PersonQueue`，`Manager`，`WaitingQueue`，`Elevator` 看成一个电梯模块（UML图蓝色框中的部分），包含两个线程以及两个共享对象，整个电梯与外界的交互就只有 `PersonQueue` 这个共享对象，只要调度器把请求发给 `PersonQueue` 电梯就能自行完成后面的一系列操作。

对对象的加锁仍然采用了synchronized为共享对象上锁的方式，同步块在某些**不需要确切状态**的地方可以尽可能小，防止过多的阻塞

所以第二次作业同步块的设置与锁处理主要针对 `TotalQueue`，`PersonQueue` 这两个共享对象（其他与第一次作业相同）

`TotalQueue` 其实就相当于第一次作业 `PersonQueue` 所有加锁操作完全相同，当增加电梯的时候需要将已经分发出去的 `PersonQueue` 我没有进行无脑加锁，而是具体考虑数据冲突产生的原因，能不加锁就不加锁：

加人，减人，设置结束信号必须加锁，判断当前人数是否为0可以不用加锁，这个共享对象人数减少的唯一原因是 `Manager` 转移请求造成的，这样在不加锁的情况下如果查询到的结果是空那么现在的状态就一定是空，如果非空，那么即使加锁查询得到的结果也是非空，也就是说在查询共享对象人数是否为0时**不加锁不影响正确性**，另外在新增电梯之后需要重新把 `PersonQueue` 中的请求新收回到 `TotalQueue` 中，按照分配策略重新分配，需要先拿到前者的锁再拿到后者得锁，但**不会产生循环依赖**

4..调度器设计分析

此次作业涉及到多部电梯的同时运行，由于第一次的架构可扩展性较好，本次作业只增加了一个 `Dispatcher` 调度器以及一个 `TotalQueue` 即可完成相关功能需求。

把第一次作业的**同步**双线程看成一个**电梯模块**，根据分而治之的思想，只要能新增一个大的调度器，将总请求队列发给各个**电梯模块**，那么**电梯模块**就能各自处理分配给自己的请求了

`TotalQueue` 用于连接 `Dispatcher` 和 `Receiver` 两个线程，之后 `Dispatcher` 会根据电梯的运行状态进行请求的分发，并且保证每次都分发完毕，`TotalQueue` 里必须为空，以**最大程度的减少电梯模块和调度器的沟通，减少过多阻塞导致的性能损失。**

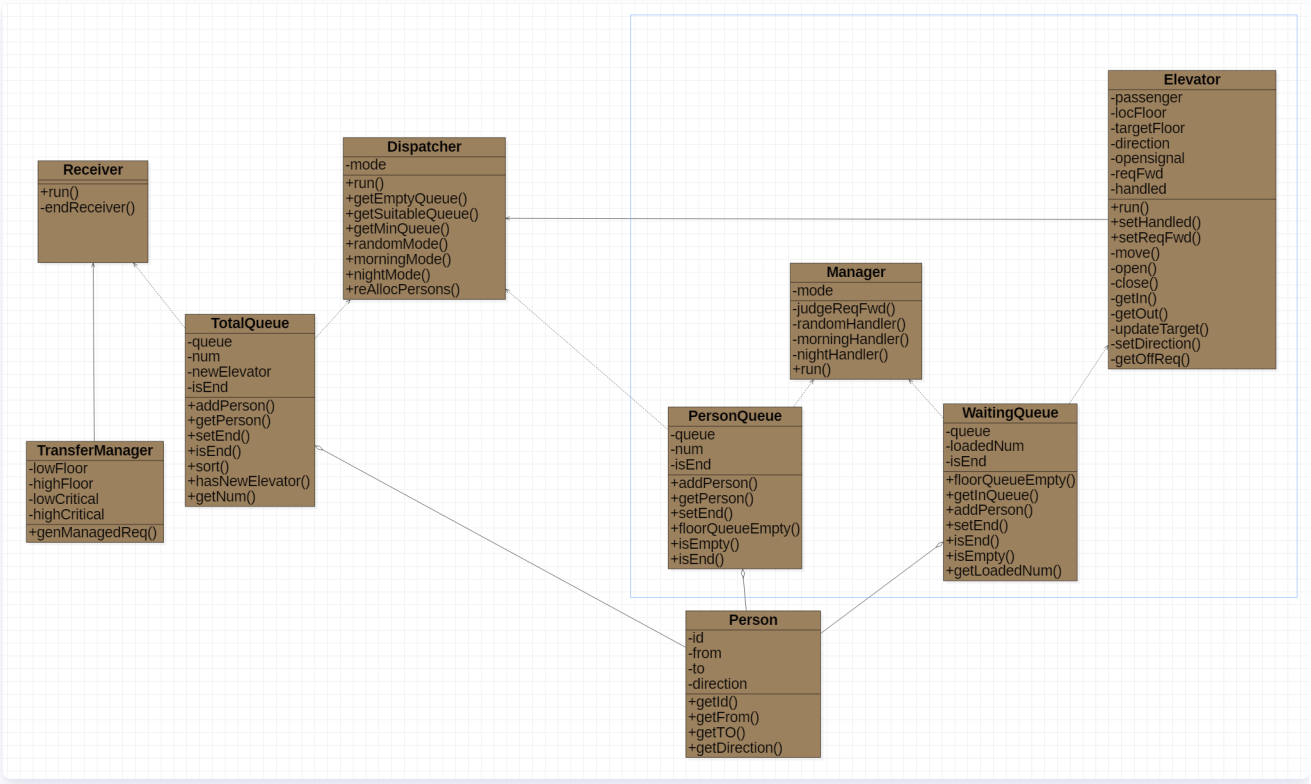
但是这样会有一个弊端，就是不能得到电梯的准确信息，只能得到一个比较模糊的状态，比如说当调度器要得到电梯当前所在楼层的时候，有可能电梯是在a楼，在调度器读取之后，电梯又进行了运动，譬如b楼，那么此时调度器所得到的状态就是错误的，或者说是模糊的，如果根据此信息来分配请求，就可能出现分配给对应电梯之后电梯不能及时处理的情况，也就是**错分**但如果要得到准确状态，`Dispatcher` 就需要时刻关心电梯的最新状态，具体实现来就是电梯线程阻塞，告诉调度器，等调度器调度完之后才能继续运行（第一次作业一部电梯的情况），但多部电梯下与调度器的频繁交互会造成巨大的性能损失，比如A电梯在与调度器交互，那么B电梯就没法运行，得等A的交互完成之后，B才有机会与调度器交互，可能5台电梯以内性能损失不是很明显，但是电梯一旦多起来，性能损失就非常大，权衡之下，选择容忍一些分配**不完全正确**的情况，舍弃部分局部最优，选择全局最优。

5.可扩展性分析

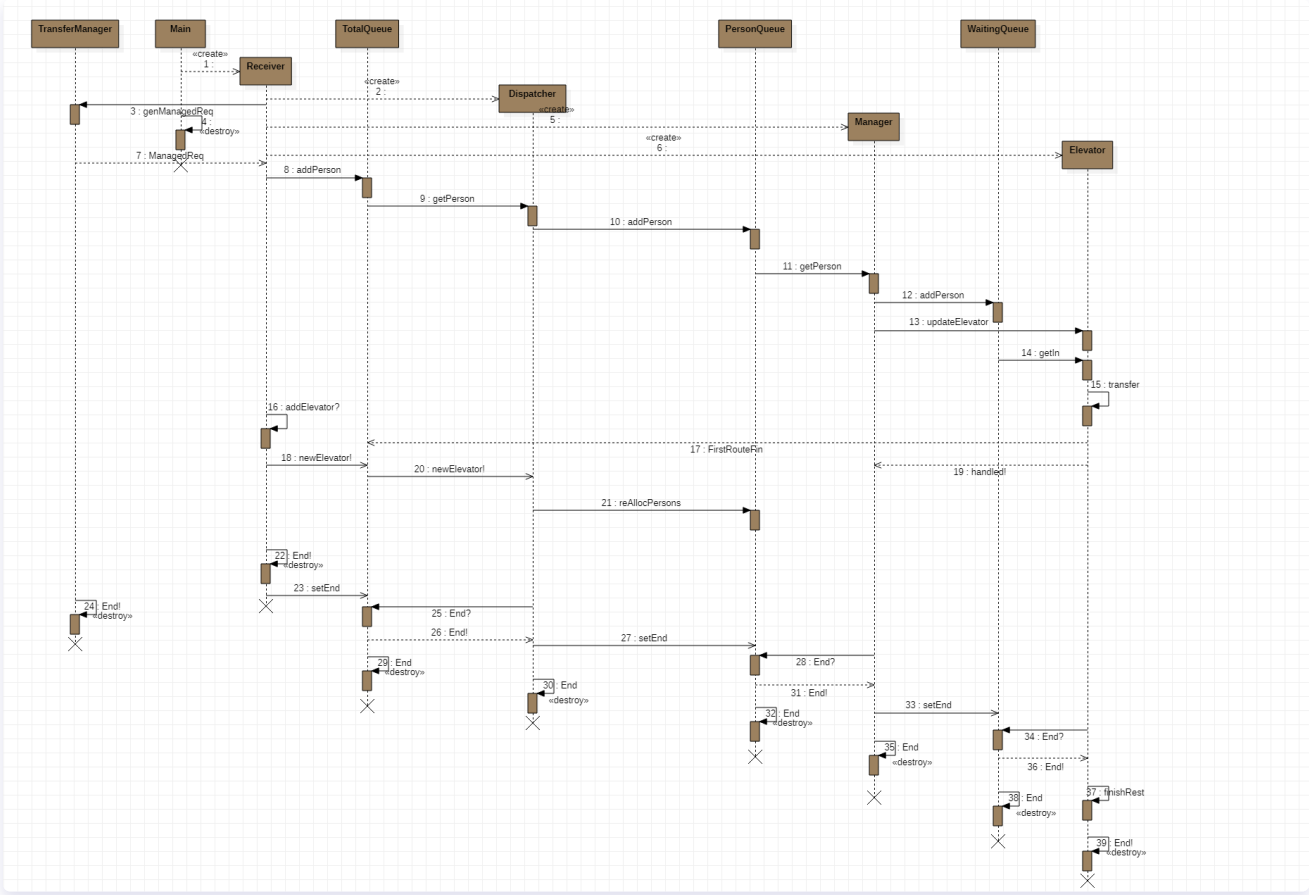
采用了分而治之的策略，对于已经搭好的架构不再轻易改动，因此第一次作业到第二次作业的过渡十分顺滑，做完第二次作业，在老师的提示下，我也思考了可能出现各种奇形怪状的电梯的情况，此时的想法是电梯本身一些无关紧要的参数直接通过创建电梯实例时传入相关参数即可，对于某些电梯出现楼层限制的话需要换乘，而这种情况可以通过分割请求的方式实现，总的来说可扩展性尚可

三、第三次 作业

1.UML



2.Sequence Diagram



3.同步块设置与锁处理

要实现动态换乘，个人认为必须要得到电梯的准确状态，但个人的架构是不能得到电梯的准确状态的，因此只能实现静态换乘，那么就需要分析

由于可扩展性较好，第三次作业在第二次作业的基础上仅仅新增了一个 `TransferManager` 类，将静态的表当做属性存在里面，当出现一个请求时就扔进去处理，把处理过的请求扔出来，放在 `TotalQueue` 中，这样就实现了与第二次作业一模一样的转换（当然还有电梯参数的传入）

因此大部分原有同步块与锁处理都与第二次作业完全相同，不用做什么改动

唯一的区别就是电梯线程也能访问 `TotalQueue` 了，仔细分析可知不会发生**循环依赖**的问题，而电梯内部的 `passenger` 列表表示当前处于电梯内部的请求，不是共享对象，不用加锁，只需要对 `TotalQueue` 加锁即可，另外虽然电梯频繁的锁 `TotalQueue` 对象，但是 `Dispatcher` 从中拿取请求的时候是一次性拿出并分配的，也就是说占有对象的时间非常少，所以不太影响性能

4.调度器设计分析

有一点要注意，就是**如何保证分割后请求执行的先后顺序**，在我的静态表中，最多进行1次换乘：

当然不排除某些情况2次甚至更多的换乘在理论上会更优，但是换乘的成本除了开关门的代价外，还需要考虑到不同电梯的容量以及当扔下请求后，请求是否能在尽量短的时间内被另外要换乘的电梯接走，综合考虑下我决定最多只换乘1次

具体做法是：

对于Person类新增 `valid` 标志， `priority` 标志， `nextRoute` 属性（下一个 `Person` 实例）

如果是前半程，那么 `valid` 标志是true，如果是后半程则为false，当一个请求出电梯时，会将nextRoute存的后半程的valid置位true，并notify TotalQueue（效果上神似又有新的请求从外部到达），这样不用不断加锁把请求传来传去，避免了线程安全问题特别是死锁的产生

另外由于本次性能分除了电梯运行时间还要考虑每个请求的等待时间，于是我设了 `priority` 表示优先，当后半程和多个前半程或者不换乘的请求都符合分配要求时，优先分配后半程，以减少请求的等待时间

所以调度器所有功能基本相同，新增的功能就是在分配请求时需要考虑每个请求的 `valid` 以及 `priority` 标志

5.可扩展性分析

- 电梯限制更苛刻：由于静态表是自己通过~~不太够用的脑子~~在纸上算完之后记在 `TransferManager` 类里的，如果相关参数稍微一变，那么将面临此类大改的窘境，现在能想到的就是**动态生成静态表**，即根据参数自行生成静态表，这样姑且能解决电梯限制简单的情况，但是如果新增更多更复杂的电梯限制，恐怕将非常难办

- 出现负楼层：考虑到现有架构下-1到1是会被算作2层的，为了实现统一，将所有负楼层向上移动1层，即-1层在输入后就处理为0层等，改动很小的情况下就能满足需求
- 出现VIP，VVIP，VVVIP：由于之前已经设置了请求的优先级，根据不同的优先级设置`priority`即可实现
- 某种情况下封锁某台电梯：当接受到指令之后通过信号对电梯实行封锁，如果电梯内部的请求允许处理完而不中途放下，那么很好实现，如果中途放下而要求继续处理，那么就要改动核心架构了，不满足开闭原则，不能很好实现
  总的来说可扩展性尚可，对于简单的新增需求能比较好的扩展，对于复杂的新增需求扩展性一般

四、bug分析

三次作业中**公测和互测都没有遇到bug**，所以就分析一下自己写完代码初步测试时遇到的一些bug

- 死锁问题：刚刚接触多线程，对于锁的持有不熟悉，导致出现了死锁，主要表现就是**循环依赖**，通常是同时访问多个共享对象，并且在访问时**没有注意**到拿锁的先后顺序以及**其他线程可能的拿锁情况**导致死锁，这种bug如果想通过运行数据并debug来解决是非常困难同时也是非常浪费时间的（~~虽然de完之后对于处理多线程bug的能力有了显著提高，对于锁有了更深刻的理解~~）但这种架构上的问题最好还是通过理清思路，多画图想清楚架构中个线程的关系来解决
- 异常问题：在处理ArrayList时边remove边遍历导致出现了异常；数组越界问题
- 死循环问题：在遍历时没有考虑某些特殊情况，导致遍历的过程中出现死循环，同时还误以为是死锁，判断错误不准从而浪费了更多的时间
- 轮询问题：没有很好地应用wait notifyAll进行线程间的交互，某些情况下自己认为不存在轮询，但是在某个操作中可能存在，通过JProfier观察CPU使用情况能很快地解决
- 数据更新问题：一定要确保wait并被唤醒后所持有的数据是最新的，换句话说不能再wait之前保存其他线程的相关数据，要在醒来之后去访问相关数据

五、hack策略

1.采用的策略

由于个人没有做评测机，在三次互测中都只能自行构造数据并投入测试，构造的数据一般会针对一些**临界情况**

- 第一、二次作业中morning，night能否正确结束线程
- 第二次作业中，程序对于扎堆请求的处理速度
- 第三次作业中对于认为**不换乘也能完成任务**的程序，构造特殊的必须要进行换乘才能在规定时间内完成的样例

2.有效性

由于是自行构造临界数据，再加上多线程本身bug难以复现，导致出现在本地跑出问题，提交之后却没有hack到的情况
最终hack结果是：

- 第一次：**0**
- 第二次：**1**
  morning模式下不该电梯还没开人就出来了
- 第三次：**3**
  对于night和morning模式存在死锁
  没换乘导致超时

相比评测机无脑黑盒测试，手动构造还是太弱了，但找一些边界数据去测试还是有点用的

3.采用了什么策略发现线程安全问题

由于个人没有做评测机，在发现线程安全问题这个问题上比较难处理
感觉只有评测机通过高强度高并发的测试才能拿找出一些隐藏很深的线程安全问题，这种问题绝对是没法通过自己手动构造手动跑能找出来的~~（不然早就修了）~~，手动构造能做的就是看边界数据能不能hack到一些线程没法正常结束这类的线程安全问题

4.与第一单元测试策略的差异之处

第二单元是多线程，第一单元是单线程
这就是最大的差异，所有的差异都源于此。对于多线程由于**高并发**，线程执行顺序不定，就会导致资源访问与修改顺序不确定，在对共享资源没有良好的保护机制下，会出现各种各样的问题，而我们测试的时候往往是在一个资源竞争不激烈的环境下进行的，但是一到公测这种资源竞争激烈的环境下，可能就会出各种自己测试遇不到的问题~~（当然也可能没遇到，但其是还是有问题）~~，带来

的结果是一千次测试中一次测出问题，那就是有问题，虽然再测两千次不一定能复现第二次……综上，第二单元的bug如果想完全通过测试来解决是不现实的，测试只能de明显的bug，**要保证程序的概率正确还得在设计层面解决**，当然这也在一定程度上增加了hack阻力

但是第一单元不同，由于是单线程，所有的执行结果都是确定的，bug是能百分之百复现的，如果**能通过覆盖性测试，并保证覆盖的有效性**那么程序是绝对没有问题的

六、心得体会

1.线程安全

- 所谓多线程，必然有多个线程同时运行，多个线程同时运行必然会涉及到多个线程对于有限个资源的访问，必然将导致出现共享对象，对于同一个对象的同时操作必然又将导致线程安全问题。所以必须通过加锁来实现操作的原子性，但是又不能无脑加锁，否则多线程形同虚设，那么对于这个锁的大小就很有考究了，这就要求我们的每一次加锁，自己都能清楚地认识到为什么加锁，加锁之前都要问问自己不加锁会产生问题吗？通过三次作业的锻炼，自己对于线程安全的理解，对于如何加锁有了显著的提升

2.层次化设计

分而治之，化繁为简

经过三次作业，我打从心底里重新认识到了这句话的意思。

面对多个线程的协同操作，如果不能把各个过程分开，瞻前顾后，耦合度十分大，那么不仅写程序困难，debug更是雪上加霜~~（别问我怎么知道的）~~，但是如果能把各个部分划开，自己只负责处理自己应该处理的事，那么做起来就轻松多了

3.迭代感受

- 在做第一次作业时，由于对多线程了解不够深刻，各种操作、语言也理解不到位，再加上~~讨论区没有大佬提供多线程debug思路~~，我就只能在控制台自行一口气投放数据，辅之肉眼debug&&画图论证框架正确性，期间产生了死锁、轮询、死循环等各种应该在第二第三次遇到的多线程问题，导致完成作业十分困难。
- 第二次作业，由于第一次有较好的架构，完成十分顺畅，~~写完子直接提交竟然没有bug~~
- 第三次作业由于动了之前的架构，产生了轮询，自己没意识到，再加上不知道RE也可能是轮询导致的，浪费了一定的时间

  个人感觉第一次作业的难度远大于第二第三次作业难度之和，（也有可能是自己把第二次作业的活挪到第一次去做了？）

  总的来说,第二单元体验还是不错,个人在多线程的认识上得到很大的拔高,在短期内多线程的debug能力以及理解也得到很大的进步!