线程及其同步设计结构：
使用继承 Thread 类或者实现 Runnable 接口来实现多线程的建立。

采用互斥控制：使得任何时刻只允许一个线程获得访问/执行权限。
synchronized(obj) {…} ：任意时刻只允许一个线程对对象 obj 进行操作
synchronized method(…){…} 任意时刻只允许一个线程调用方法 method

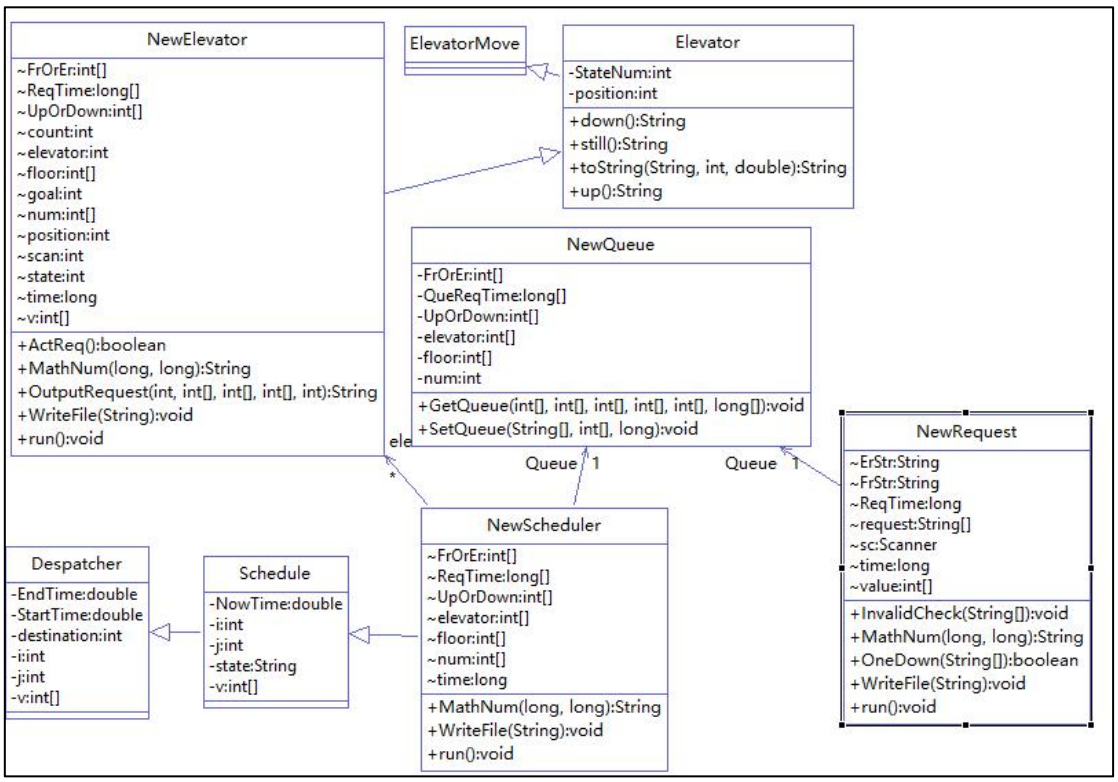
任何线程在临界代码区(critical section)执行时
可能有多个其他线程在等待进入执行
所以执行结束前通过 notify/notifyAll 来让 JVM 调度一个线程进入并执行。

通过锁来实现进程同步及互斥：
J 锁对象 e 的同步方法(“synchronized m(…){}”)
使用锁对象 e 来控制的同步代码段(“synchronized (e){…}”)

利用 wait()来让一个线程在某些条件下暂停运行。例如，在生产者消费者模型中，生产者线程在缓冲区为满的时候，消费者在缓冲区为空的时候，都应该暂停运行。如果某些线程在等待某些条件触发，那当那些条件为真时，可以用 notify 和 notifyAll 来通知那些等待中的线程重新开始运行。不同之处在于，notify 仅仅通知一个线程，并且我们不知道哪个线程会收到通知，然而 notifyAll 会通知所有等待中的线程。

如果 read-modify-write 和 check-then-act 计算只涉及单一变量，就可以通过 java.util.concurrent.atomic 包中提供的 Atomic****类型来确保计算的原子性。

第三次电梯：（1）类图



（2）分析程序设计结构

类的个数：7

Despatcher 类（40 行）：属性 6，方法 1（Paln()30 行），

Elevator 类（80 行）：属性 3，方法 9

NewElevator 类（200 行）：属性 12，方法 6（NewElevator()4 行，run()120 行，ActReq()10 行，MathNum()5 行，WriteFile()21 行，OutPutRequest()12 行）

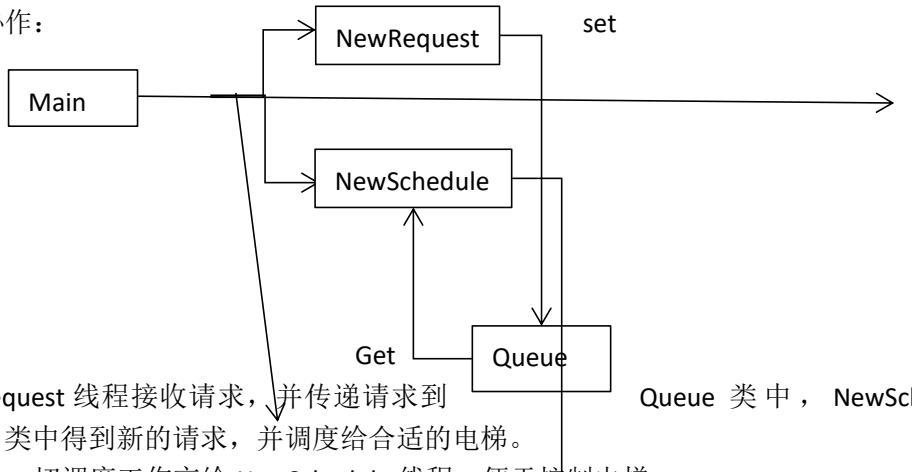
NewQueue 类（87 行）：属性 5，方法 2（GetQueue()38 行，SetQueue()31 行）

NewRequest 类（156 行）：属性 8，方法 6（NewRequest()4 行，InvalidCheck()34 行，MathNum()5 行，OneDown()8 行，run()18 行，main()23 行）

NewSchedule 类（169 行）：属性 8，方法 4（NewSchedule()5 行，MathNum()5 行，run()107 行，WriteFile()22 行）

Schedule 类（215 行）：属性 5，方法 2（OutputRequest()11 行，Plan()200 行）

线程协作：

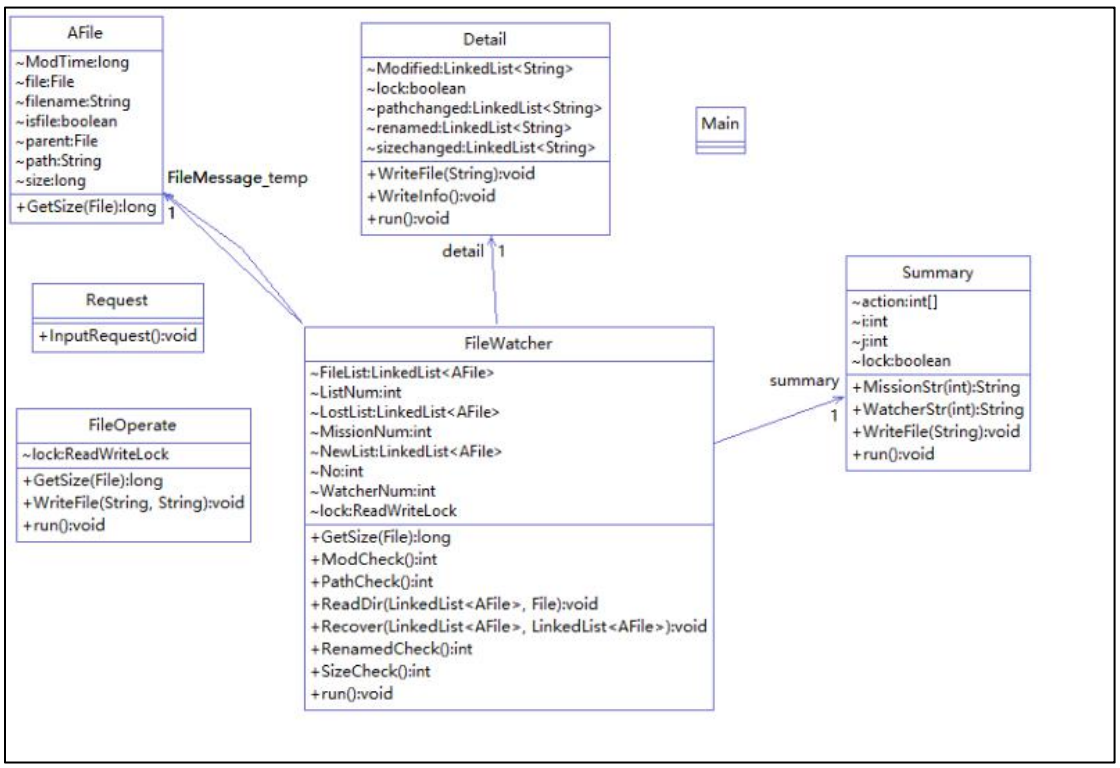


NewRequest 线程接收请求，并传递请求到 Queue 类中，NewSchedule 从 Queue 类中得到新的请求，并调度给合适的电梯。

优点：一切调度工作交给 NewSchedule 线程，便于控制电梯。

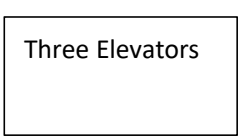
缺点：NewSchedule 线程所做的工作过于多，导致 NewSchedule 类代码冗长。

IFTTT:（1）类图：



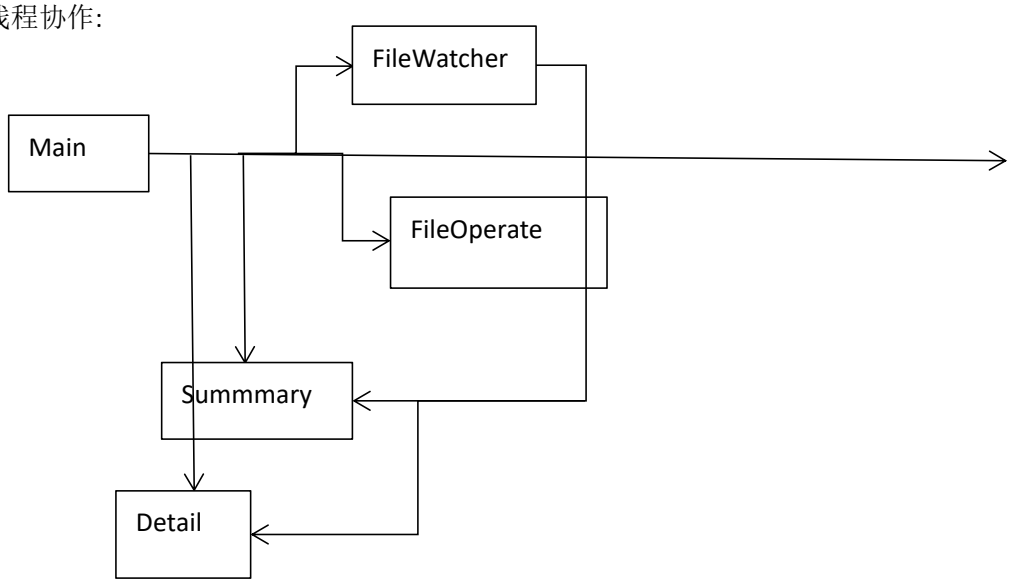
（2）分析程序设计结构

类的个数：7.



AFile 类（45 行）：属性 7,方法 2（Afile()10 行，GetSize()20 行）
Detail 类：（114 行）：属性 5，方法 4（add()28 行，run()23 行，WriteFile()21 行，WriteInfo()19 行）
FileOperate 类（125 行）：属性 1，方法 4（FileOperate()3 行，run()64 行，WriteFile()21 行，GetSize()19 行）
FileWatcher 类（510 行）：属性 11，方法 9（FileWatcher()35 行，run()56 行，RenamedCheck()98 行，PathCheck()97 行，ModCheck()49 行，SizeCheck()81 行，Recover()10 行，GetSize()19 行，ReadDir19 行）
Main 类（110 行）：属性 0，方法 1（main()96 行）。
Request 类（11 行）：属性 0，方法 1（InputRequest()6 行）
Summary 类（91 行）：属性 4，方法 5（add()5 行，run()26 行，WatcherStr()9 行，WriteFile()21 行，MissionStr()8 行）

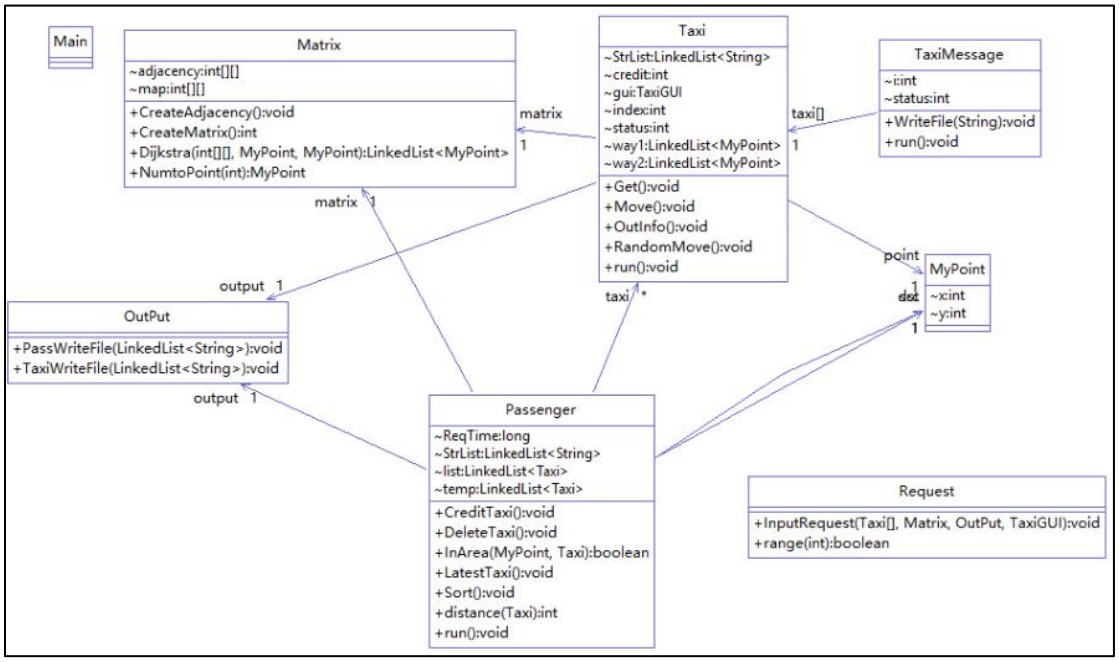
线程协作:



FileOperate 线程负责对文件进行访问、修改，FileWacher 线程监控文件变化状态并及时向 Summary 或 Detail 线程传递信息。

优点：各线程分工明确，不会造成某个类过于大。
缺点：多个线程，如果采取的锁模式没有考虑周到，可能造成效率低下

出租车：（1）类图



（2）分析程序设计结构

类个数：9

gui 类

Main 类（39 行）：属性 0，方法 1（main()38 行）

Matrix 类（240 行）:属性 3，方法 4（CreateAdfacency()35 行，CreateMatrix()61 行，Dijkstra()97 行，NumtoPoint()10 行）

MyPoint 类（10 行）：属性 2，方法 1（MyPoint()4 行）

OutPut 类（61 行）：属性 0，方法 2（PassWriteFile()24 行，TaxiWriteFile()24 行）

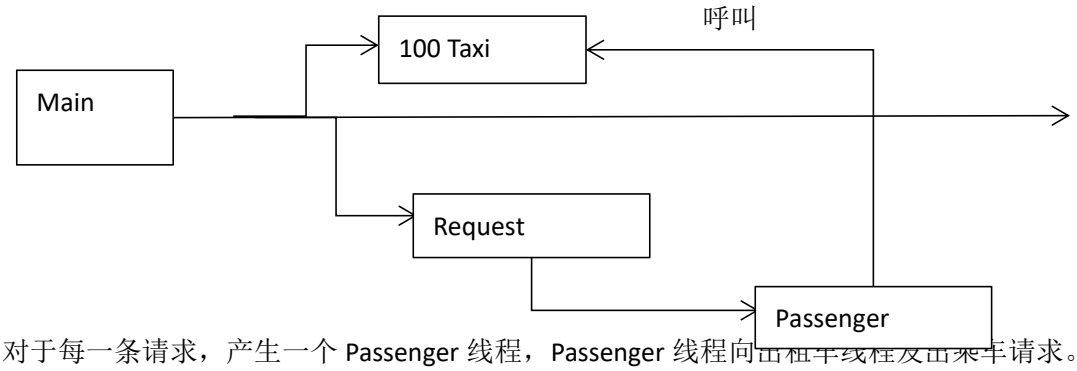
Passenger 类（209 行）：属性 9，方法 8（Passenger()11 行，CreditTaxi()12 行，DeleteTaix()7 行，distance(6 行)，InArea()9 行，LatestTaxi()11 行，run()82 行，Sort()29 行）

Request 类（74 行）：属性 0，方法 2(InputRequest()54 行，range()8 行)

Taxi 类（206 行）：属性 10，方法 6（Taxi()7 行，Get()7 行，Move()62 行，OutIlfo()7 行，RandomMove(49 行)，run()45 行）

TaxiMessage 类（78 行）：属性 3，方法 3（TaxiMessage()5 行，run()36 行，WriteFIle()21 行）

线程协作：



对于每一条请求，产生一个 Passenger 线程，Passenger 线程向出租车线程发出乘车请求。

分析程序 bug：

这三次作业，我的 bug 主要有以下几点，第一，第三次电梯时，我的请求队列实际上没有加对锁，这使得程序出现了一系列的时对时错的 bug，除此之外，我还犯了一些粗心的毛病，比如对正则表达式的理解还不充分，导致一些错误的输入没有报错。在 IFTTT 那一次的作业中，我的程序只要监控 C 盘或者 D 盘这样的目录，就有可能 crash，这是因为系统盘中有一些隐藏的临时系统文件夹，如 System Volume Information(系统卷标信息)等。

如何测试：

主要针对临界值，如第三次电梯时测试可以用多层捎带请求测试，测试出租车可以使用同时发出的请求来测试线程的安全问题。

心得体会：

在多线程编程时，共享的资源最好都加上锁，毕竟在不考虑性能的情况下，这是最安全的。编写多线程程序时，对于哪些资源是临界资源一定要在设计开始时就设计好，否则容易遗漏加锁。多个线程读写操作时，可以使用 java 的读写锁，避免使用 synchronized 方法导致的读之间互斥，以及性能低下。