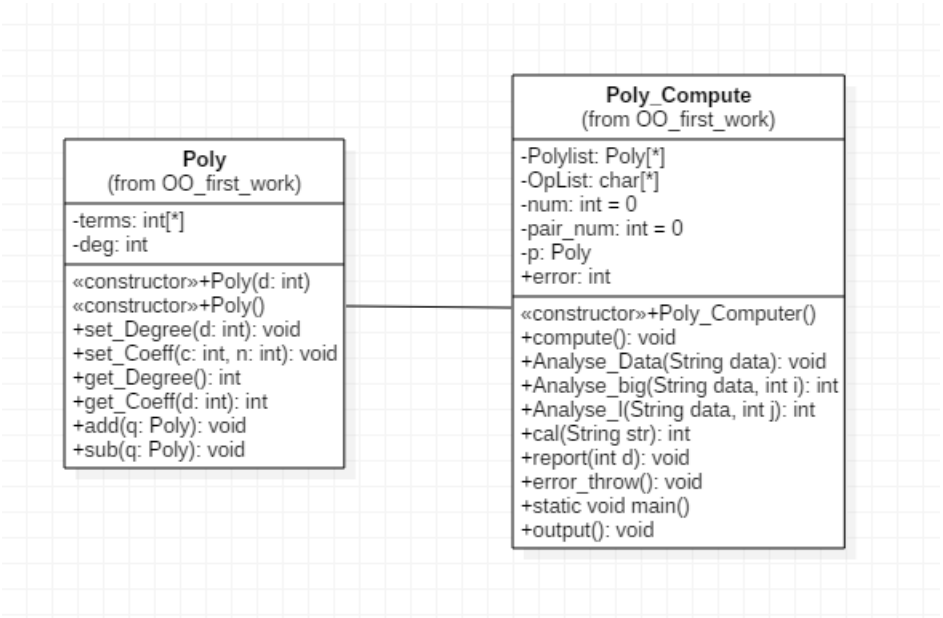
总结性博客

一、 程序结构分析

度量中属性的存取 getter 与 setter 不被计入度量方法数，代码规模计算计入了单独成行的括号。

1. 第一次作业

UML：



Poly 类 NOA：2，NOO：2； Poly_Compute 类 NOA：6，NOO：8

Poly 类属性包括系数数组和最大指数，方法除了属性存取与构造方法外有两个，方法 add()和方法 sub()没有控制分支，规模都在 10 行左右；Poly 类总代码在 50 行左右；

Poly_Compute 类属性有 6 个，包括多项式数组和运算符数组，其他为辅助计算和报错的变量；Poly_Compute 类的方法除了构造方法和 main 函数外有 8 个，方法 compute()用来计算多项式，控制分支为 2 个，规模为 12 行；方法 report()用来设置错误类型 error，控制分支为 0 个，规模为 3 行；方法 error_throw()用来输出错误类型，控制分支为 18 个，规模为 30 行；方法 output()用来输出正确结果，控制分支为 2 个，规模为 20 行；方法 Analyse_Data()、Analyse_big、Analyse_l()、cal()都是用来处理输入的多项式的，其中 Analyse_Data()控制分支为 10 个，规模为 30 行；Analyse_big()控制分支为 15 个，规模为 40 行；Analyse_l()控制分支为 10 个，规模为 50 行；cal()控制分支为 9 个，规模为 40 行；类总代码在 300 行左右；

方法间调用与类之间关系：

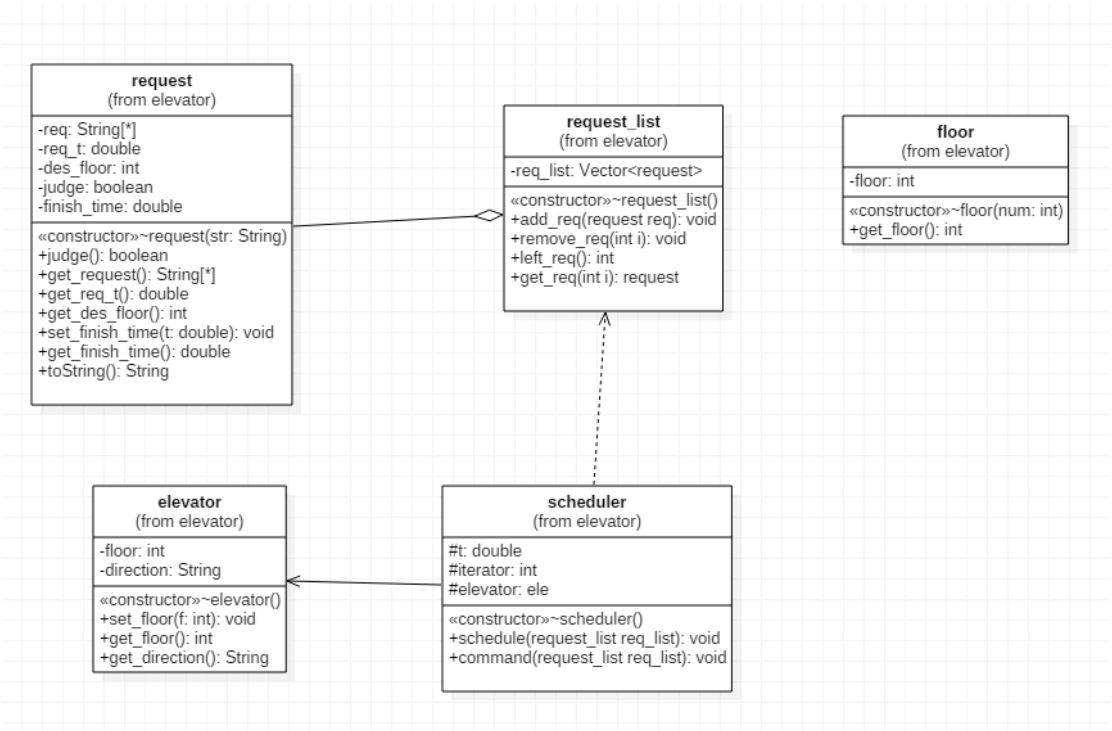
在 Poly_Compute 类中，Analyse_Data()调用 Analyse_big()，Analyse_big()调用 Analyse_l()，Analyse_l()调用 cal()，一系列调用是为了处理输入的字符串，将处理分解为大括号间，大括号内和小括号内，Analyse_Data()处理完字符串后结果会存储在 PolyList 和 OpList 中提供其他方法使用，在上述方法中都会调用 report()方法及时存储错误信息，output()在存在错误时会调用 error_throw()，error_throw()根据 report()更新的 error 值抛出错误类型；Poly_Compute 类包含 Poly 类成员，compute()方法会调用 Poly 的 add()与 sub()方法；

缺点：

1） Poly_Compute 类方法过多，代码冗余，主要原因是在处理字符串上采取的策略不佳，类中的许多方法都是为了处理字符串拆分出来的；
2） 应该使用有限状态机或者正则表达式来简化输入处理；
3） 应该将处理字符串、处理错误等部分拆分出来单独作为类。
4） Poly 类中数组初始化过于土豪，在构造函数内给数组分配了很大内存；

2. 第二次作业

UML：



request 类 NOA：5，NOO：2；request_list 类 NOA：1，NOO：4；
elevator 类 NOA：2，NOO：0；scheduler 类 NOA：3，NOO：2；
floor 类 NOA：1，NOO：1；

request 类属性有 5 个，req 为请求字符串分解后的字符数组，judge 为请求是否合法的标志，其他为请求的一些特征；方法除了属性存取和构造方法外有 2 个，judge()方法用来判断构造 request 的请求字符串是否符合要求，控制分支为 4 个，规模为 20 行；toString()重写是为了输出请求，控制分支为 2 个，规模为 8 行；类代码为 50 行左右；

request_list 类属性有 1 个，为请求队列；方法除了构造方法有 4 个，add_req()方法作用是将合法请求加入请求队列同时输出不合法请求，控制分支为 6 个，规模为 15 行；remove_req()方法作用是从请求队列移除请求，控制分支为 0 个，规模为 3 行；left_req()方法返回请求队列剩余请求总数，控制分支为 0 个，规模为 3 行；get_req()方法返回指定位置请求，控制分支为 0 个，规模为 3 行；类代码为 30 行左右；

elevator 类属性有 2 个，包括电梯所处楼层和运动方向；方法都是属性的存储，其中 set_floor()方法由于同时设置了方向含有 3 个控制分支，其他方法控制分支都为 0；类代码为 20 行左右；

scheduler 类属性有 3 个，包括了调度的时间和电梯；方法除了构造方法和 main 函数有

2 个，方法 schedule()取出指令并调度电梯和输出电梯运行状态，控制分支为 7 个，规模为 40 行；方法 comman()循环调用 schedule()，控制分支为 0 个，规模为 5 行；类代码为 60 行左右；

floor 类属性为楼层，方法为返回楼层，控制分支 0 个，类代码 10 行；

方法间调用与类之间关系：

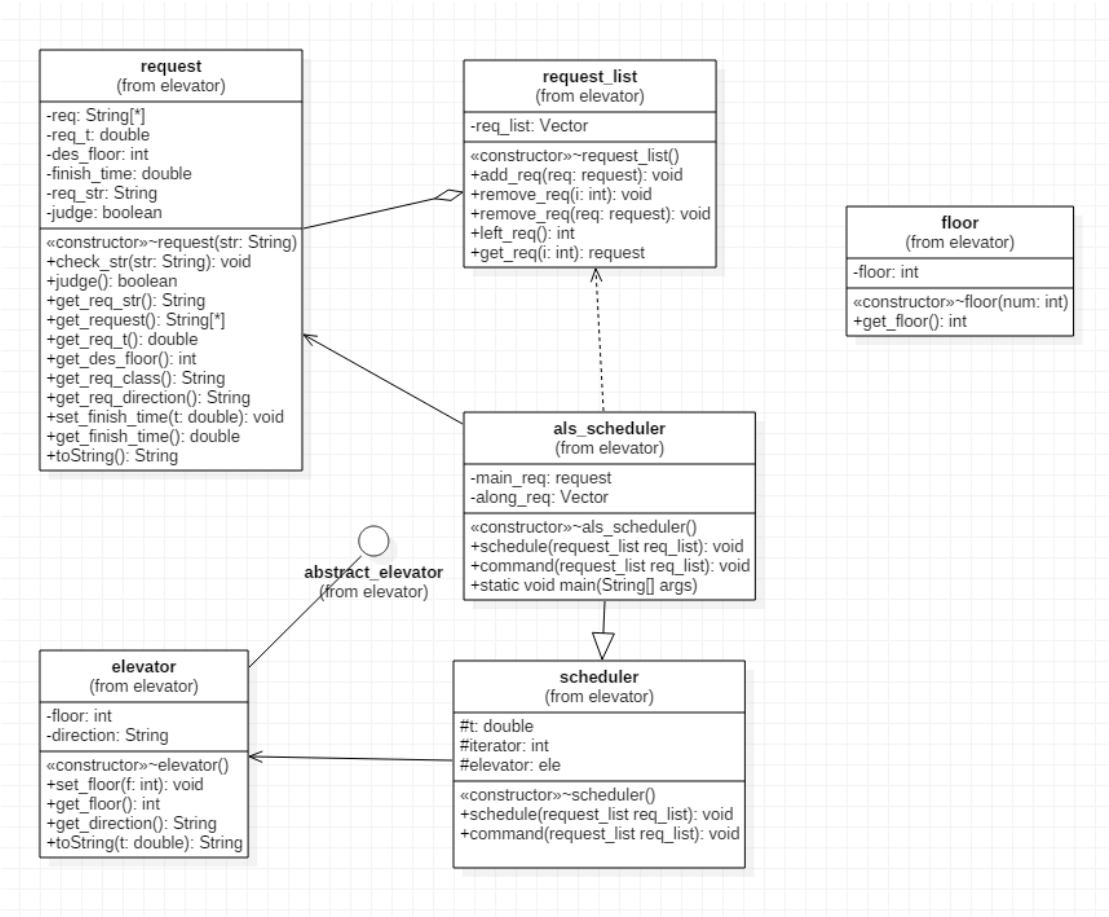
request_list 类中存储了 request 对象的队列；scheduler 类中包含了一个 elevator 对象；scheduler 类与 request_list 类通过协同进行交互；scheduler 类的 command()方法调用 schedule()方法，schedule()方法传入一个 request_list 对象，从中取出请求或删除相同请求，使用 elevator 的 set_floor()调度电梯并通过电梯的 get_floor()方法和 get_direction()方法获取电梯运行信息；

缺点：

1） 输入处理放在了 request 类的构造方法和 judge()方法中，应该单独分出类进行输入处理，但是 request 类在构造和加入队列前就进行了合法性判断确保进行调度时都是合法请求；
2） scheduler 类中的 command()方法承担的功能过少，只是起到了调用 schedule()方法的作用，而 schedule()方法承担功能过多，既要取指令，又要调度电梯并完成输出；
3） floor 类成了花瓶，完全没有发挥任何作用…

3. 第三次作业

UML：



request 类 NOA：6，NOO：3；request_list 类 NOA：1，NOO：5；

elevator 类 NOA：2，NOO：1；
scheduler 类 NOA：3，NOO：2；als_scheduler 类 NOA：2，NOO：2
floor 类 NOA：1，NOO：1；

request 类增加了属性 des_floor 请求的目标楼层，req_str 原始请求字符串；增加了一些方法存取属性的方法，增加了 check_str()方法辅助判断请求合法性，控制分支为 9 个，代码规模为 35 行；judge()方法改为仅仅返回 judge 的值，控制分支为 0 个，代码规模变为 3 行；类的代码为 80 行左右；

request_list 类属性没有变化；方法增加了一个 remove_req()的重载，控制分支为 0 个，代码规模为 3 行；类代码为 40 行左右；

elevator 类属性没有变化；方法增加了一个 toString()方法的重写，控制分支为 0 个，代码规模为 3 行；类代码为 30 行左右；

scheduler 类和 floor 类没有变化；

als_scheduler 类继承 scheduler 类，属性增加了主请求 main_req 和捎带请求队列 along_req；方法重写了父类的 command()方法和 schedule()方法，其中 command()方法控制分支为 1 个，代码规模为 5 行；schedule()方法控制分支为 42 个，代码规模为 120 行；类代码为 180 行；

方法间调用与类之间关系：

als_scheduler 类包含一个 request 对象和一个 request 对象组成的队列；als_scheduler 与 request_list 类需要相互协作完成功能；als_scheduler 类中 command()方法调用 schedule()方法，schedule()方法完成取指令与电梯调度；

缺点：

1）输入处理依旧放在 request 类的构造中，应该单独实现一个类进行输入处理；

2）als_scheduler 类中依旧存在与父类一样的问题，command 方法承担的功能过少，而 schedule()方法承担了过多的功能，不仅需要取出指令，还要识别相同指令和捎带指令，同时要完成电梯的调度和电梯状态的输出，这样 schedule()方法的规模膨胀到了 120 多行，企图完成的工作数目过多，成为了名副其实的过长方法；方法过长容易让人难以理解，应该对 schedule()方法进行拆分，将各个需要完成的功能提炼成多个精简的方法或者将长方法分解为组合方法；此外 schedule()方法中出现了相似代码块，使得方法规模扩充了将近一倍，如果想办法将相似代码块进行合并可以大大减少代码规模；schedule()方法中的控制分支也偏多；

3）als_scheduler 继承父类后出现了“被拒绝的遗产”，父类的一些属性子类完全不需要；

二、 自己程序的 bug 分析

1. 第一次作业

第一次作业公共测试集未发现 bug，互评被发现一个 bug。

互评被发现的 bug 属于 readme 存在的漏洞，本身程序处理并没有问题；但是对第一次作业必须反省的是在输入处理上过于繁琐，如果采用正则表达式更加简便，出现错误可能性也更小；

2. 第二次作业

第二次作业公共测试集被发现 5 个 bug（属于两类），互评未被发现 bug

公共测试集被发现的 bug 都是输入处理存在的问题，第一类错误是对于(FR,1,DOWN,0)和(FR,10,UP,1)之类的请求没有报错，第二类错误是对没有后括号的请求没有报错；由于电梯是傻瓜式调度，互评没有被发现 bug；

对于第一类错误，最初是进行了处理的，但是中间修改过一次代码，这个处理忘记加上了，这说明自己的测试方法并不完善，由于是中途修改的代码，在修改前进行的测试也并没有记录下来重新用来测试，导致之后的测试并没有覆盖到错误；第二类错误的发生原因在于直接使用 split()对字符串进行分割，对原始字符串并没有使用正则表达式进行判断，导致后括号缺失时依旧可以进行计算；

程序主要是在 request 类中进行请求合法性的判断，这两类错误的发生都是在 request 类中，分别在 request 的构造方法和 judge()方法中；应该写一个类单独进行输入处理，输入处理使用正则表达式更佳；

3. 第三次作业

第三次作业公共测试集未被发现 bug，互评被发现 1 个 bug

互评被发现的 bug 是：如果主请求是 FR 类型的请求，而请求的方向与电梯运动方向不同，那么当电梯到达主请求目标楼层时，程序对于和主请求结构相同的相同请求不会抛出相同请求，而会在下次执行；发生问题的类是 als_scheduler 类，其中的 schedule()方法负责取指令和电梯调度，经过检查发现是在电梯到达目标楼层时去除相同请求时没有考虑到这种情况；由于 schedule()方法过长，检查错误的难度也会相应变大，应该考虑将相似代码块合并以及对 schedule()方法进行提炼或拆分；

三、 发现别人程序 bug 采用的策略

在三次作业互评中，我发现别人程序 bug 采用的策略分三种：

一是在使用之前记录下来的测试自己程序的测试样例去测试别人的程序；

二是阅读他人程序的代码，在一些出错概率大的部分寻找可能存在的 bug；

三是从输入格式，一般情况和特殊情况三方面考虑并结合他人代码结构重新设计新的测试集；

四、 心得体会

1. 设计
   1) 避免出现过大类。过大类在 UML 图中会显得比其他类要高，通常过大类了解得太多或者完成的功能过多，比如在第一次作业中，我的 Poly_Compute 类就成了过大类，6 个属性 8 个方法明显比 Poly 类要多得多，代码冗余，功能过多，完全可以分解和迁移；
   2) 避免出现过长方法。貌似不含注释、空行、花括号，超过 10 行的方法都算长；在看过 Java 一些类的源码，又对比 github 上一些项目如 node 的源码后发现确实大多数方法并没有超过 10 行；一个方法如果要完成的工作目的多，要进行分解，可以通过多个短小的方法组合起来；
   3) 避免出现不必要的暴露和复杂的条件逻辑

2. 测试样例设计：

设计全面的测试样例对发现 bug 会有一定帮助，测试样例的设计要考虑以下几个方面：

1） 输入格式

输入格式上首先要考虑符合指导书的正确格式，然后要结合自己代码在进行输入处理时采取的策略可能存在的问题来设计错误输入，如括号缺失等；

2） 一般情况

程序需要保证一般的情况能够正确输出，同时对于指导书上提到的一般性错误要有反馈；比如第二次和第三次作业中对于时间乱序的处理，对于相同请求的处理等；

3） 特殊情况

特殊情况包括输入和逻辑的测试；输入上有输入上的空输入和超长输入等，逻辑上则是程序在处理问题时的正确性，主要是对一些边界情况和极端情况进行测试，比如第一次作业的最大输入限制，第二次和第三次作业的多种相同请求反复出现；

另外要注意重现 bug 来分析 bug 原因；