一开始上OO课程时，关于这门课程的传言有很多，在知乎上也用很多的学长、学姐把这门课程批判一番，最令广大同学深恶痛绝的就是OO的互测扣分制度，用很多学长对此有很大的意见。曾经在知乎上见到有人评价说：“北航OO实行一种“宰了别人救活自己”的十分可怕的课程制度。这门课的老师不排除想“弄出个大新闻”的“噱头”效应。但是，如果通过之前一年的实验，这位老师真的可以培养出学生鸡蛋里挑骨头的能力其实也是合理的（这是将来市场竞争以及博弈的需要）。不过，如果我是这位老师，我依然要修正这种方法，因为鼓励揭发的方法是反人文的。而且必须跟学生说明我的初衷并能进一步换取学生的理解而不是激化矛盾。在学生的角度，学生在那种恶性竞争的机制下回搜肠刮肚似的找错误，以谋取更多的成绩。老师和学生之间在这里有一种天然的不平等，很多的学生就是有心也觉得无力去反抗，因为成本太大。听说，这位老师还在课上将自己的学生跟什么北京学院的学生比较来讽刺自己的学生，我真心觉得这位老师应该好好学学教育学的基本知识。老师应该鼓励学生提出自己的意见，对于敢于提出异议的学生期末应该予以加分。因为这是一个一本万利的事。我觉得这位老师以及计算机学院的各位领导们应该仔细思考一下这些问题。我航作为一个全国知名大学且有志建设成世界一流大学，世界一流大学都没有用过这种坑人制度来对本科生进行教学的。只希望你们可以警醒，不要让怀部长和徐校长的愿望泡汤。”评论的言辞激烈，义愤填膺。在这里我姑且不讨论OO的弊病，虽然它如此饱受争议，但是经过一个学期的学习，我对面向对象程序设计这门课程还是有非常大的收获。

在最初几次的OO作业里，我深刻地认识到了在任何程序设计中对输入的严格控制是多么的重要，如果输入不符合规范却还是通过了输入检查，无疑会给程序带来巨大的隐患。而在前几次作业里不知道多少同学因为ctrl+z被爆了crash。永远不要相信用户的输入是符合规范的，这是我得到的深刻教训。因此，我们在编写程序时一定要严防死守，采用正则表达式等方法，严格限定输入，一旦用户的输入不正确，应当立即报错，最好还能告诉用户输入的输入信息究竟错在了哪个地方。

接着就是多线程的问题，多线程问题很难，其中包含了线程的协作、并发、互斥、线程安全等等十分困难的问题。用多线程只有一个目的，那就是更好的利用cpu的资源，因为所有的多线程代码都可以用单线程来实现。多线程指的是这个程序（一个进程）运行时产生了不止一个线程。并行指多个cpu实例或者多台机器同时执行一段处理逻辑，是真正的同时。并发则是指通过cpu调度算法，让用户看上去同时执行，实际上从cpu操作层面不是真正的同时。并发往往在场景中有公用的资源，那么针对这个公用的资源往往产生瓶颈。线程安全经常用来描绘一段代码，它指在并发的情况之下，该代码经过多线程使用，线程的调度顺序不影响任何结果。这个时候使用多线程，我们只需要关注系统的内存，cpu是不是够用即可。反过来，线程不安全就意味着线程的调度顺序会影响最终结果，如不加事务的转账代码。同步在Java中指的是通过人为的控制和调度，保证共享资源的多线程访问成为线程安全，来保证结果的准确。在保证结果准确的同时，提高性能，才是优秀的程序，而且线程安全的优先级高于性能。Java语言的synchronized关键字，当它用来修饰一个方法或者一个代码块的时候，能够保证在同一时刻最多只有一个线程执行该段代码。在多线程的作业中，我有大量使用了synchronized关键词。这个关键词可不简单，当两个并发线程访问同一个对象object中的这个synchronized(this)同步代码块时，一个时间内只能有一个线程得到执行。另一个线程必须等待当前线程执行完这个代码块以后才能执行该代码块。然而，当一个线程访问object的一个synchronized(this)同步代码块时，另一个线程仍然可以访问该object中的非synchronized(this)同步代码块。尤其关键的是，当一个线程访问object的一个synchronized(this)同步代码块时，其他线程对object中所有其它synchronized(this)同步代码块的访问将被阻塞。当一个线程访问object的一个synchronized(this)同步代码块时，它就获得了这个object的对象锁。结果，其它线程对该object对象所有同步代码部分的访问都被暂时阻塞。但是，synchronized也有自己的不足，首先它是一个重型锁，也就是说它可以满足线程方面的要求但是它的大量使用可能会导致性能方面的问题。在分布式开发中，锁是线程控制的重要途径。Java为此也提供了2种锁机制，synchronized和lock。synchronized：在需要同步的对象中加入此控制，synchronized可以加在方法上，也可以加在特定代码块中，括号中表示需要锁的对象。lock：需要显示指定起始位置和终止位置。一般使用ReentrantLock类做为锁，多个线程中必须要使用一个ReentrantLock类做为对象才能保证锁的生效。且在加锁和解锁处需要通过lock()和unlock()显示指出。所以一般会在finally块中写unlock()以防死锁。synchronized和lock性能上也有较大的区别。synchronized是托管给JVM执行的，而lock是java写的控制锁的代码。synchronized原始采用的是CPU悲观锁机制，即线程获得的是独占锁。独占锁意味着其他线程只能依靠阻塞来等待线程释放锁。而在CPU转换线程阻塞时会引起线程上下文切换，当有很多线程竞争锁的时候，会引起CPU频繁的上下文切换导致效率很低。而Lock用的是乐观锁方式。所谓乐观锁就是，每次不加锁而是假设没有冲突而去完成某项操作，如果因为冲突失败就重试，直到成功为止。乐观锁实现的机制就是CAS操作（Compare and Swap）。我们可以进一步研究ReentrantLock的源代码，会发现其中比较重要的获得锁的一个方法是compareAndSetState。这里其实就是调用的CPU提供的特殊指令。现代的CPU提供了指令，可以自动更新共享数据，而且能够检测到其他线程的干扰，而 compareAndSet() 就用这些代替了锁定。这个算法称作非阻塞算法，意思是一个线程的失败或者挂起不应该影响其他线程的失败或挂起的算法。synchronized原语和ReentrantLock在一般情况下没有什么区别，但是在非常复杂的同步应用中，请考虑使用ReentrantLock，特别是遇到下面2种需求的时候。某个线程在等待一个锁的控制权的这段时间需要中断。ReentrantLock的lock机制有2种，忽略中断锁和响应中断锁，这给我们带来了很大的灵活性。比如：如果A、B这2个线程去竞争锁，A线程得到了锁，B线程等待，但是A线程这个时候实在有太多事情要处理，就是一直不返回，B线程可能就会等不及了，想中断自己，不再等待这个锁了，转而处理其他事情。这个时候ReentrantLock就提供了2种机制，第一，B线程中断自己（或者别的线程中断它），但是ReentrantLock不去响应，继续让B线程等待，你再怎么中断，我全当耳边风（synchronized原语就是如此）；第二，B线程中断自己（或者别的线程中断它），ReentrantLock处理了这个中断，并且不再等待这个锁的到来，完全放弃。ReentrantLock这个类还提供了2种竞争锁的机制：公平锁和非公平锁。这2种机制的意思从字面上也能了解个大概：即对于多线程来说，公平锁会依赖线程进来的顺序，后进来的线程后获得锁。而非公平锁的意思就是后进来的锁也可以和前边等待锁的线程同时竞争锁资源。对于效率来讲，当然是非公平锁效率更高，因为公平锁还要判断是不是线程队列的第一个才会让线程获得锁。

后来的几次作业都是关于规格设计方面的。规格设计的好坏直接影响了后续代码实现阶段代码的质量。而且设计阶段的bug在代码实现阶段，是很难用高超的编程技巧弥补的。一个好的程序不是靠程序员高超的编程技巧实现的，而是由严格的规章制度所产生的。只有严格的按照需求设计规格、实现代码、进行测试，才能做成一个好的程序。

最后谈一谈我对这门课程的一些看法。我觉得由于评测方式的原因，老师在课堂上讲的规范化的设计方法往往得不到有效实施。例如在写JSF时，有的同学为了少写方法的JSF，更主要的是为了少扣分，将方法故意写成面条代码，因为每个方法的JSF扣分有上限，所以规范化设计的话，需要分解成多个方法，最大扣分值多，但是面条代码却只有一个方法，最大扣分值很少。除此之外，还有同学尽量少写类甚至只写一两个类以同样方式逃避不变式的检查。除此之外，恶意扣分机制一直说要实行但却从未实行。