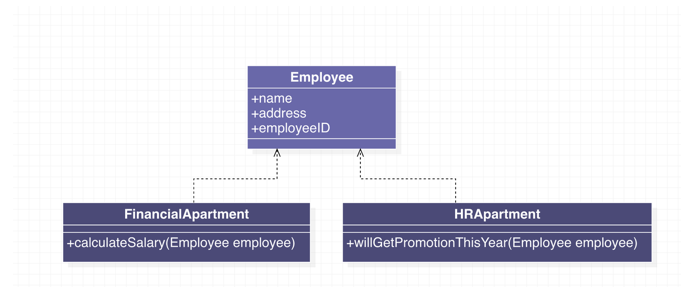
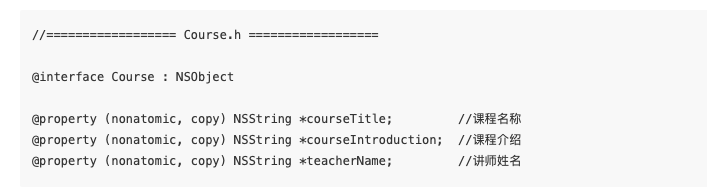
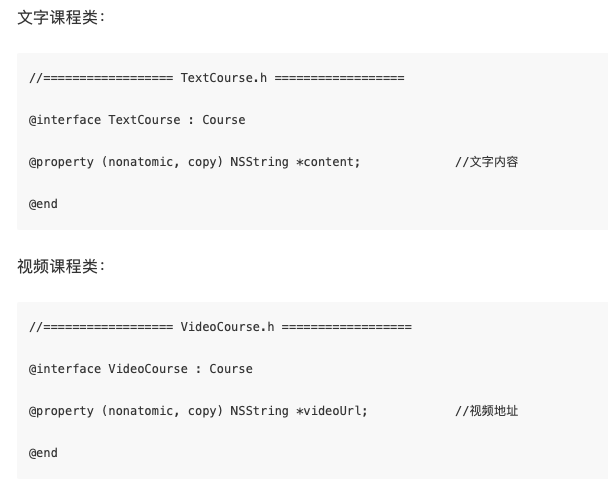
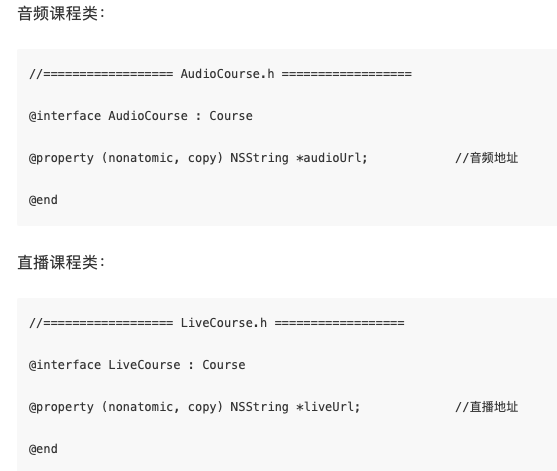
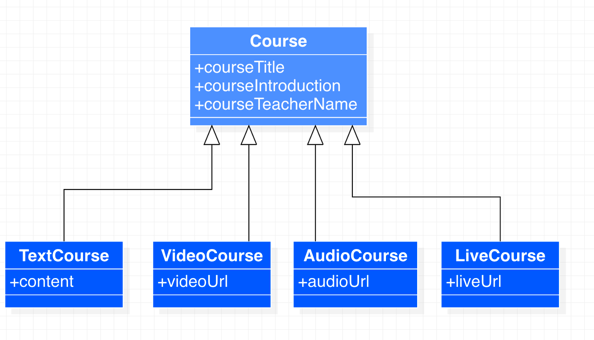
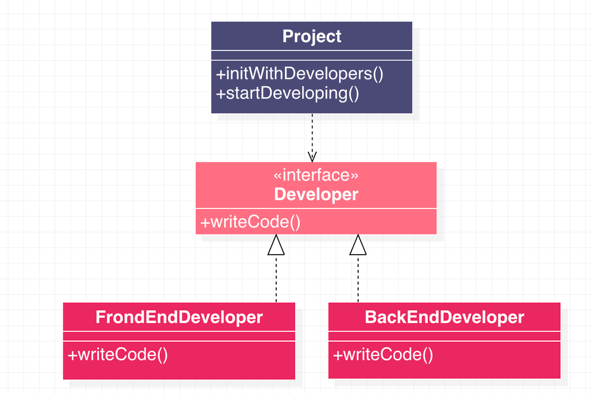
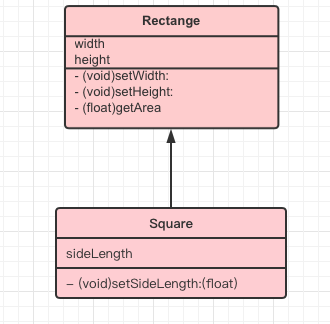
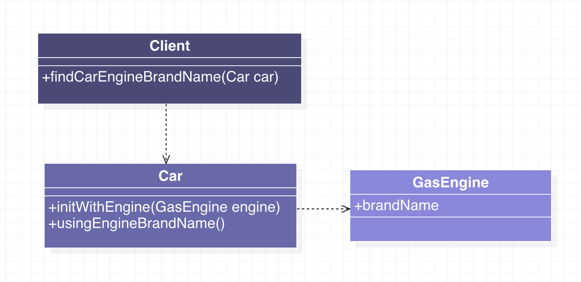
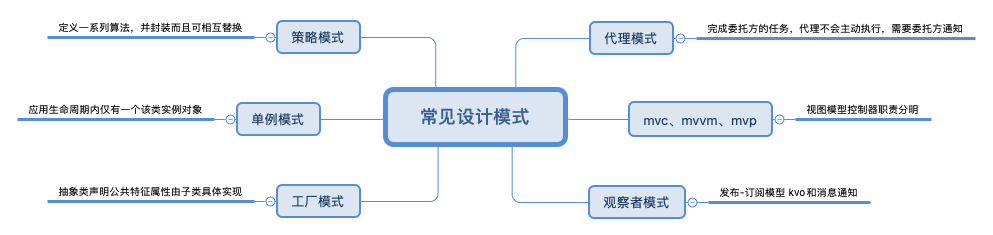
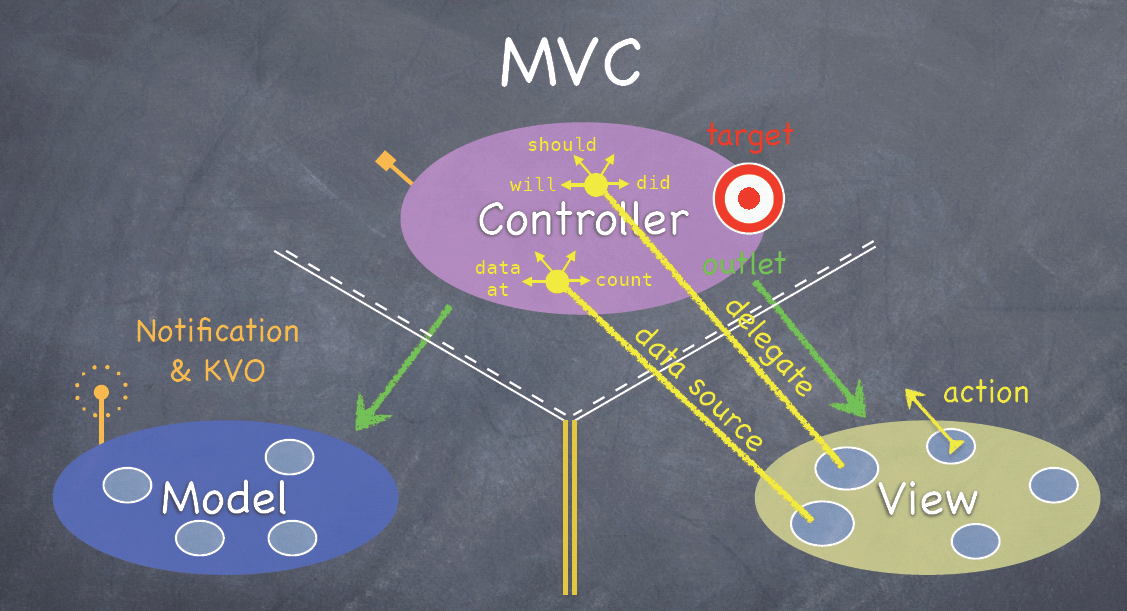
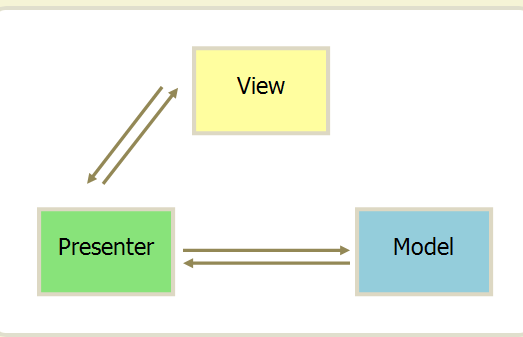
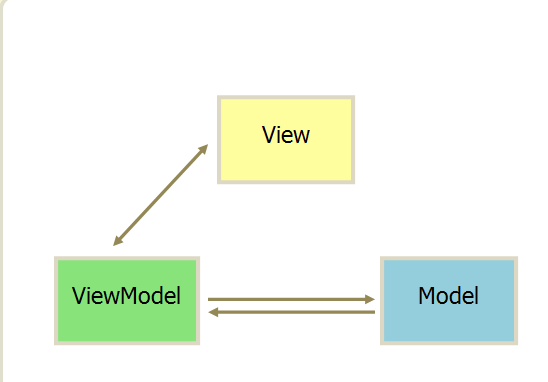
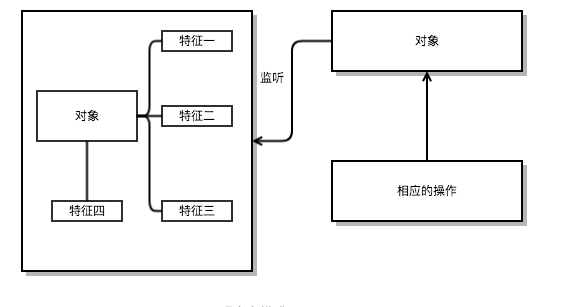
设计原则和设计模式

1. 设计原则。六大设计原则：  
     
   1、单一职责原则（一个类只做一件事）。  
    这个很好理解，就类似于微服务一样，尽量一个类只有一个职责。而且最好函数也遵循单一原则，即：一个函数只做一件事。尽量能抽象出来，如此可类似于堆积木一样方便后期复用以及便于理解和维护。  
    示例：初始需求只有员工类信息。需增加判断员工是否升职（人事部）。判断员工是否涨薪（会计部）。很多人的实现就是直接在员工类下面进行心中这两个方法。但随着公司的变大企业结构的复杂，更多的人事部和会计部门的需求增多。那么如此下去类将臃肿难以维护。正确的做法是：通过新增会计部门类和人事部门类，将员工传入后进行实现。  
     
   2、开闭原则（对修改关闭，对扩展开放。（抽象基类细化子类））   
    他的定义为：一个软件实体如类、模块和函数应该对扩展开发，对修改关闭。即用抽象构建框架，用实现扩展细节。在以不改动原有类的方式实现新需求。以实现事先抽象出来的接口（或具体类继承抽象类）的方式来实现。  
    示例代码：设计一个在线课程类，初期只有文字讲解的课程。但随着资源增多。后期增加视频课程，音频课程和直播课程。大多数人为了图方便就一个人抽象出所有这样随着需求的增多，类将非常臃肿难以维护。相对较好的设计是  
      
     
     
   UML图为：  
     
    如果以后有新的资源增加也可以在不影响现有的情况下增加，而且对于不同资源有特殊需求时其他资源也不会有冗余难懂代码。（其实阅读中的conten类就可以重新设计按开闭原则）。  
   3、接口隔离原则  
    多个特定的接口好于一个通用性的总接口。  
    不建立庞大臃肿的接口，尽量细化接口。但粒度不能太小，可能会造成接口数量过多使设计复杂化。  
    示例：典型就是tableviewDelegate和tableviewDataSource。  
   4、依赖倒置原则  
    依赖抽象，而不是依赖实现。  
    抽象不应该依赖细节；细节应该依赖抽象。  
    高层模块不能依赖低层模块，二者都应该依赖抽象。  
   理解：  
    针对接口编程，而不是针对实现编程。  
    尽量不要从具体的类派生。而是以继承或者实现接口来实现。  
    业务层属于上层模块，逻辑层和数据层归为底层。  
   示例：用代码模拟一个项目实际开发场景：前端和后端开发人员开发同一个项目。简单来将就是一个数组中有前端开发和后端开发，遍历调用writeCode方法。最粗暴的方法就是遍历时判断对象类型然后调用方法不易维护和扩展。简单方法为：一种是直接抽象开发类通过继承实现前后端开发重写writecode 一种为通过协议进行实现。  
     
    5、 里氏替换原则  
    所有引用基类的地方必须透明的使用其子类的对象，也就是说子类对象可替换其父类对象，而程序执行效果不变。  
    父类可以被子类无缝替换，且原有的功能不受任何影响。  
   示例：正方形继承长方形设计。有部分同学就是强行重写长宽的set方法这样就改变了父类的setget方方法从而父类无法被子类无缝替换。正确的方法是新增个边长然后将变长set方法中将长宽都设置为边长相等。  
      
   6、迪米特法则  
    一个对象应当对其他对象尽可能少的了解，也就是只接触那些真正需要接触的对象，实现高聚合、低耦合。  
    迪米特法则也叫做最少知道原则，一个类应该只和他的成员变量，方法的输入，返回参数重的类做交流，而不应该引入其他类。  
    示例：设计一个汽车类，包含汽车的品牌名称，引擎等成员变量。提供一个方法返回引擎的品牌名称。有引擎类、车类、引擎有个引擎名称。不好的设计是直接返回引擎类让使用者直接使用引擎类的名字。较好的解耦为如下UML直接返回名称而不是引擎对象。这样外部使用时就不需要关系引擎类。  
   

设计模式。六大常用设计模式  
  
1、代理模式   
 一种消息的传递方式，一个完整的代理模式包括：委托对象、代理对象和协议。  
 代理和block对比：多个消息传递时，delegate更直观清晰。代理更面向过程，block则更面向结果。从性能上来说，block的性能消耗略大于delegate。因为block会涉及到栈区向堆区的拷贝等操作，所以时间和空间上都大于代理。  
2、mvc、mvvm mvp模式  
mvc：经典模式  


MVP：全称model-view-presenter.即中间人模式  
  
mvvm:弱化了controller的作用，通过VM来进行数据处理和页面协调  
  
3、观察者模式   
 当对象的某个特征变化时，另一个对象做出相应的处理和操作。后者观察前者的一举一动，并及时对一些变化做相应。在iOS中的KVO（系统通过runtime实现）  
  
4、工厂模式   
 通熟易懂的就是，进去的是原材料，出来的是成品，至于怎么去生产和加工的不需要知道，只需要输送原料，便可生产产品。  
5、单例模式  
 确保整个程序生命周期内只存在一个实例对象，并提供一个类方法提供全局调用。  
 优点：只有一个实例，所以定位问题方便。 节省系统内存资源，提高程序运行效率（指针对那些会需要反复使用的数据进行单利）。  
 缺点：不能被继承，不能有子类。不易被重写或扩展（可以使用分类）。会一直占有系统内存。在整个程序运行期间都会存在。  
6、 策略模式  
 定义了一组算法，将每个算法都封装起来，并且使他们之间可以互换。  
