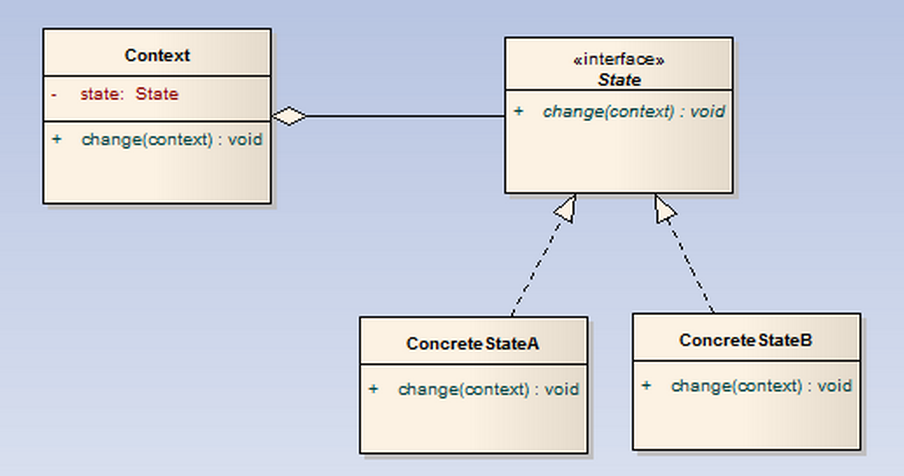
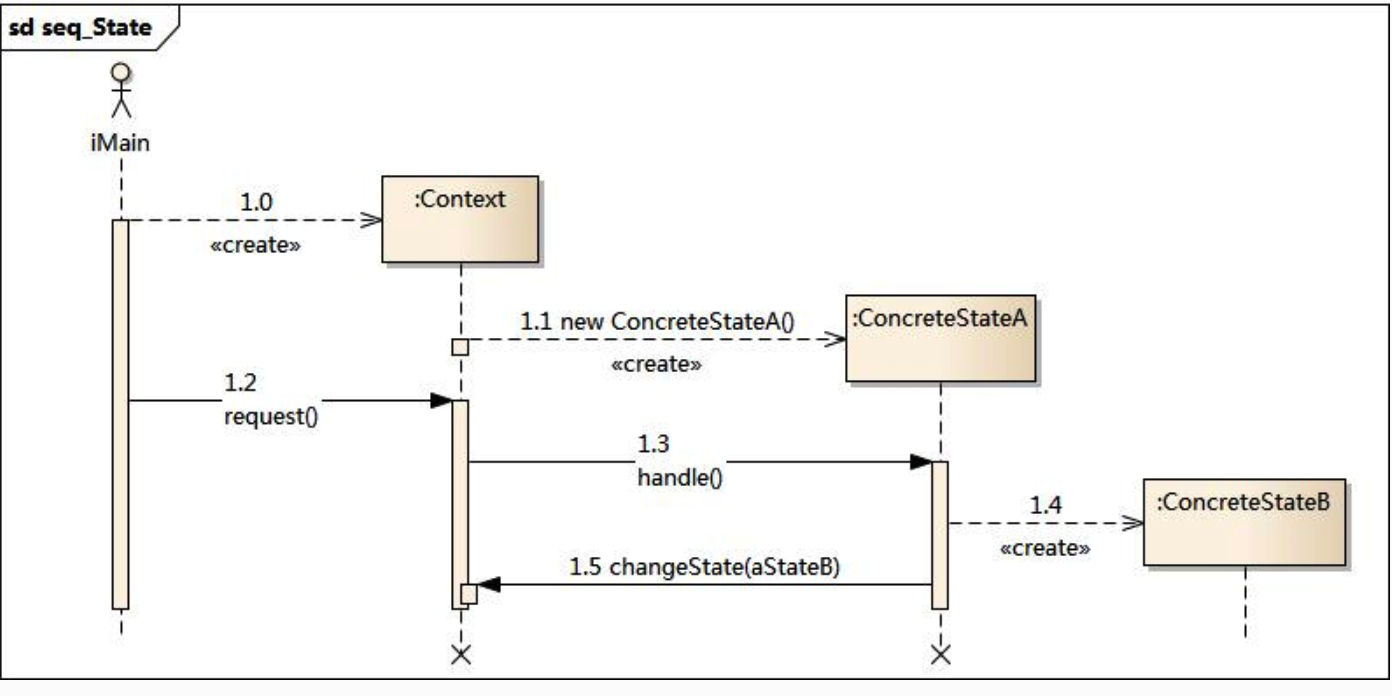
1. 状态模式：**允许一个对象在其内部状态改变时改变它的行为。对象看起来似乎修改了它的类**
2. 在很多情况下，一个对象的行为取决于一个或多个动态变化的属性，这样的属性叫做**状态**，这样的对象叫做有状态的(stateful)对象，这样的对象状态是从事先定义好的一系列值中取出的。当一个这样的对象与外部事件产生互动时，其内部状态就会改变，从而使得系统的行为也随之发生变化
3. 状态模式主要解决的是当控制一个对象状态转换的条件表达式过于复杂时的情况。把状态的判断逻辑转移到表示不同状态的一系列类当中，可以把复杂的判断逻辑简化。当一个对象的行为取决于它的状态，并且它必须在运行时刻根据状态改变它的行为时，就可以考虑使用状态模式了。
4. 核心: 用于解决系统中复杂对象的状态转换以及不同状态下行为的封装问题
5. 类图

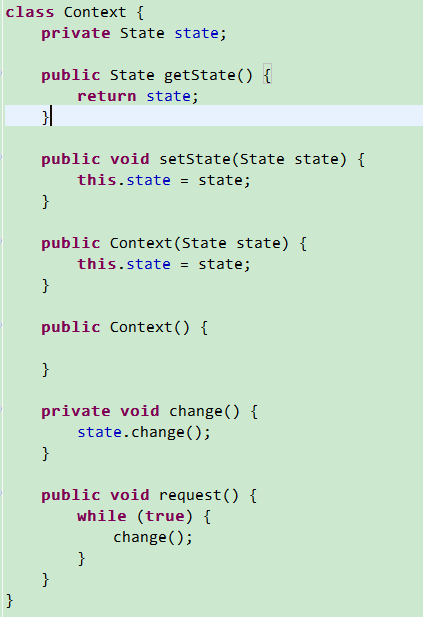


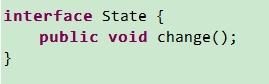
* 1. 抽象状态（State）角色：定义一个接口，用以封装环境对象的一个特定的状态所对应的行为。
  2. 具体状态（ConcreteState）角色：每一个具体状态类都实现了环境的一个状态所对的行为。
  3. 场景（Context）角色：定义客户端所感兴趣的接口，并且**保留**一个表示当前状态的具体状态类的实例。这个具体状态类的实例给出此环境对象现有的状态。

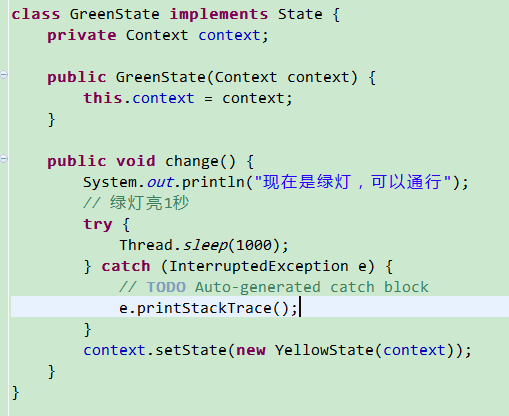
1. 时序图

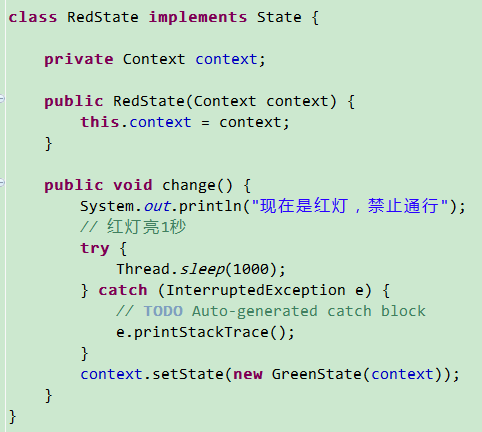


1. 例

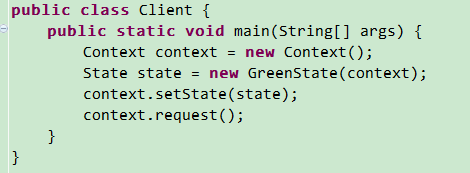




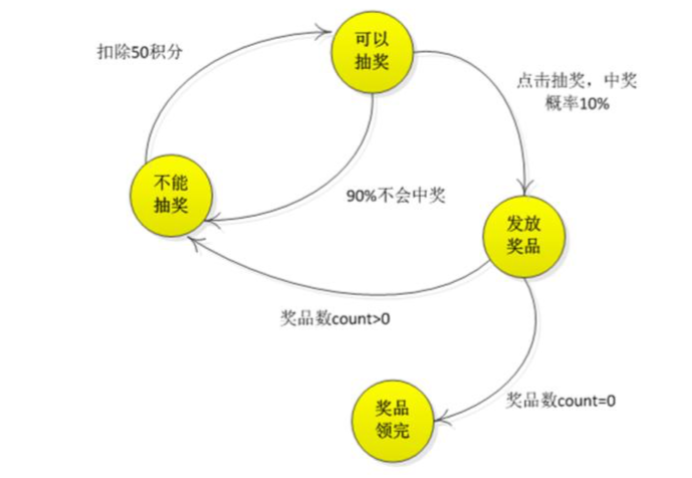




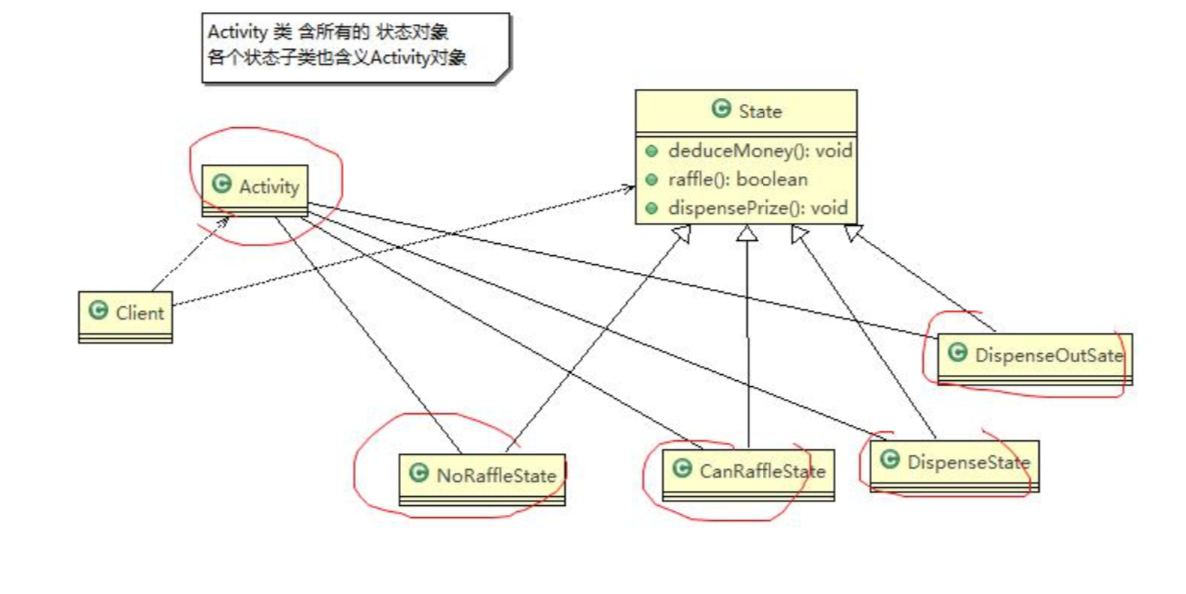




1. 状态模式优缺点
   1. 优点
      1. 封装了转换规则。
      2. 枚举可能的状态，在枚举状态之前需要确定状态种类。
      3. 将所有与某个状态有关的行为放到一个类中，并且可以方便地增加新的状态，只需要改变对象状态即可改变对象的行为。
      4. 允许状态转换逻辑与状态对象合成一体，而不是某一个巨大的条件语句块。
      5. 可以让多个环境对象共享一个状态对象，从而减少系统中对象的个数。
   2. 缺点
      1. 状态模式的使用必然会增加系统类和对象的个数。
      2. 状态模式的结构与实现都较为复杂，如果使用不当将导致程序结构和代码的混乱。
      3. 状态模式对“开闭原则”的支持并不太好，对于可以切换状态的状态模式，增加新的状态类需要修改那些负责状态转换的源代码，否则无法切换到新增状态；而且修改某个状态类的行为也需修改对应类的源代码
2. 适用场景
   1. 对象的行为依赖于它的状态（属性）并且可以根据它的状态改变而改变它的相关行为。
   2. 代码中包含大量与对象状态有关的条件语句，这些条件语句的出现，会导致代码的可维护性和灵活性变差，不能方便地增加和删除状态，使客户类与类库之间的耦合增强。在这些条件语句中包含了对象的行为，而且这些条件对应于对象的各种状态
3. 应用。状态模式在工作流或游戏等类型的软件中得以广泛使用，甚至可以用于这些系统的核心功能设计，如在政府OA办公系统中，一个批文的状态有多种：尚未办理；正在办理；正在批示；正在审核；已经完成等各种状态，而且批文状态不同时对批文的操作也有所差异。使用状态模式可以描述工作流对象（如批文）的状态转换以及不同状态下它所具有的行为
4. 实例：抽奖活动
   1. 假如每参加一次这个活动要扣除用户50积分，中奖概率是10%
   2. 奖品数量固定，抽完就不能抽奖
   3. 活动有四个状态: 可以抽奖、不能抽奖、发放奖品和奖品领完
   4. 活动的四个状态转换关系图(右图)



* 1. 类图



1. 状态模式和策略模式结构完全一样，很容易混淆，这里列举下状态模式和策略模式的区别
   1. 可以通过环境类状态的个数来决定是使用策略模式还是状态模式。
   2. 策略模式的环境类自己选择一个具体策略类，**具体策略类无须关心环境类**；而状态模式的环境类由于外在因素需要放进一个具体状态中，以便通过其方法实现状态的切换，因此**环境类和状态类之间存在一种双向的关联关系**。
   3. 使用策略模式时，客户端需要知道所选的具体策略是哪一个，而使用状态模式时，客户端无须关心具体状态，环境类的状态会根据用户的操作自动转换。
   4. 如果系统中某个类的对象存在多种状态，不同状态下行为有差异，而且这些状态之间可以发生转换时使用状态模式；如果系统中某个类的某一行为存在多种实现方式，而且这些实现方式可以互换时使用策略模式