## 七、建造者模式

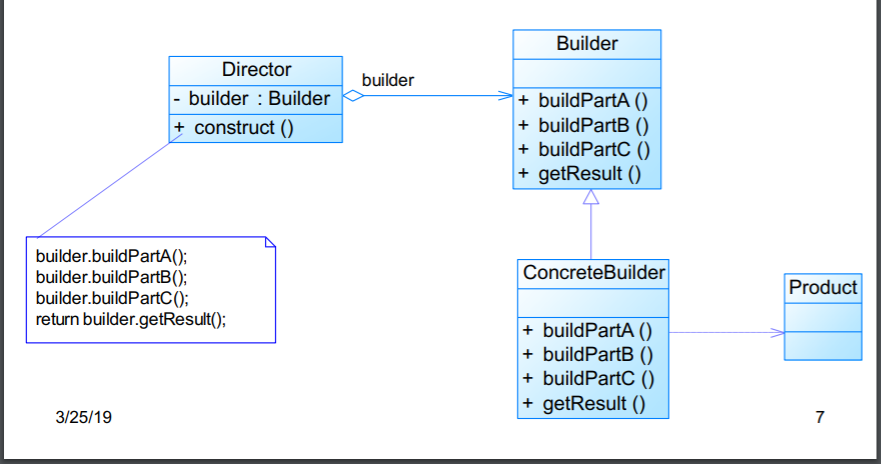
**1．动机**

建造者模式**可以将部件和其组装过程分开，一步一步创建复杂的对象。用户只需要指定复杂对象的类型就可以得到该对象，而无需指导其内部的具体构造细节。**

这些复杂对象拥有一系列成员属性，这些成员属性有些是引用了类型的成员对象。

复杂对象相当于一辆有待建造的汽车，而对象的属性相当于汽车的部件，**建造产品的过程就相当于组合部件的过程**。这个过程很复杂，因此被外部化到建造者对象中。建**造者返回给客户端的是一个已经建造完毕的完整产品对象，而用户无需关心该对象所包含的属性以及他们的组装方式。**

**2．模式结构**



Builder中有一个私有的product对象，getResult()将返回这个对象

3．模式分析

**引入了指挥者类Director，作用：**

1. **隔离客户与生产过程**
2. **负责控制产品的生成过程**

指挥者针对抽象构造者编程，客户端只需要知道ConcreteBuilder的类型即可返回一个完整的产品。

**4．模式优缺点**

优点：

1. 客户端不必指导产品内部组成的细节，将产品本身与其创建过程解耦，使得相同的构建过程可以创建不同的产品对象。
2. 添加新的具体建造者无需修改原有类库代码，Director类针对抽象的Builder编程，系统扩展方便，**符合开闭原则**。
3. 每一个具体建造者相互独立，因此可以很方便地替换具体建造者，用户使用不同的具体建造者即可得到不同的产品对象。
4. 可以更加精细地控制产品的创建过程。将复杂的产品创建步骤分解到不同的方法中，构建过程更加清晰。

缺点：

1. 创建的产品一般具有较多的共同点，组成部分相似。**若产品间的差异性很大，则不适合使用建造者模式。**
2. 若产品内部变化复杂，可能导致需要定义很多具体建造者类来实现这种变化，导致系统变得庞大。
3. 模式：建造者模式的简化
4. 省略抽象建造者角色——若只需要一个具体建造者
5. 省略指挥者角色——在省略抽象建造者的情况下，可以省略Director，由Builder扮演上述两种角色。

6. 建造者模式与抽象工厂模式的比较

建造者模式返回一个组装好的完整产品，而抽象工厂模式返回一系列相关的位于不同产品等级结构的产品，即产品族。

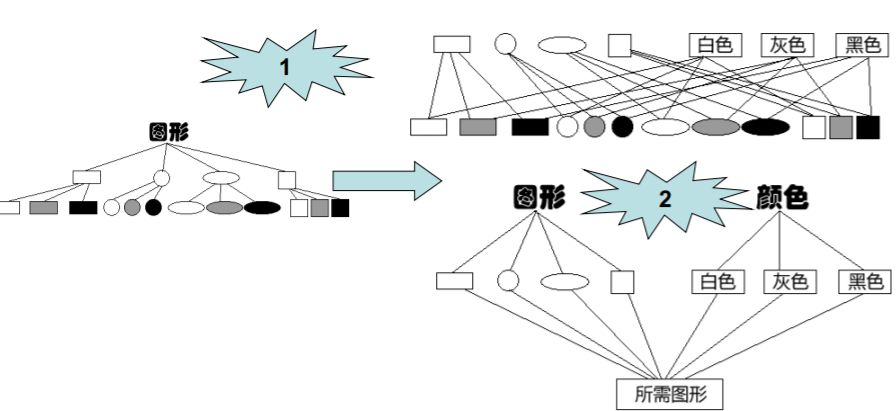
在抽象工厂模式中，由客户端指定实例化的具体工厂类，然后调用工厂方法获取产品族中一个一个的产品对象。而在建造者模式中，客户端可以不直接调用建造者相关方法，而是通过将复杂对象的建造过程委托给Director后，通过Director直接获得一个完整的产品对象。

抽象工厂——汽车配件生产工厂

建造者模式——汽车组装工厂

## 八、桥接模式 （有一题是很直观的可以使用桥接模式）

1. 模式动机：

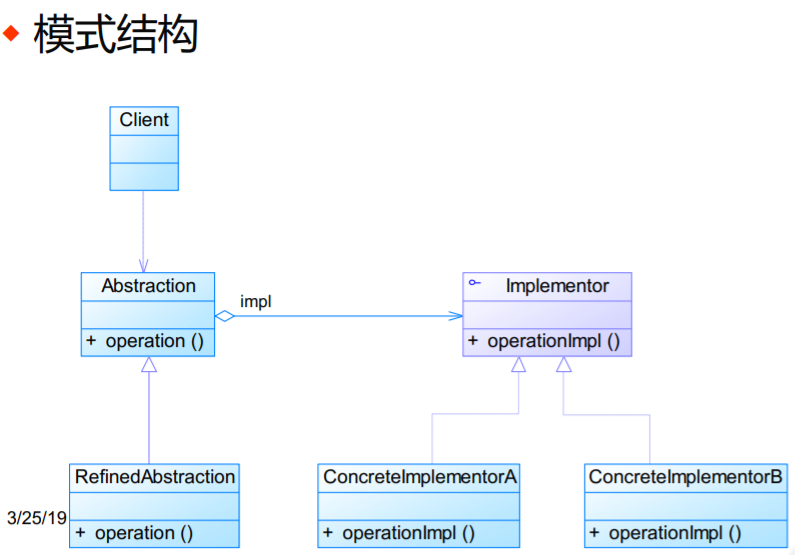


对于有两个以上变化维度的系统，第二种设计系统中的类更少，扩展更方便。

**桥接模式将继承关系转换为关联关系，降低了类与类之间的耦合减少代码量。**

2. 模式定义：bridge pattern，接口模式，handle and body （柄体模式）中handle是指什么部分？

将抽象部分（所需图形）与实现部分分离，使他们可以独立变化



3． 模式分析：抽象化和实现化脱耦

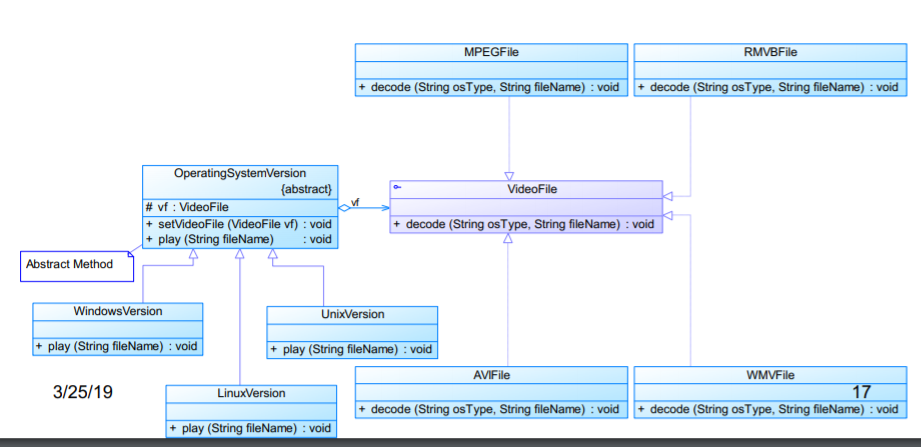
抽象化 忽略一些信息，把不同的实体当做相同的实体对待。讲对象的共同性质抽取出来形成类的过程即为抽象化过程。

实现化 针对抽象化给出的具体实现

脱耦 将抽象化与实现化之间的强关联改成弱关联。继承关系改为关联关系

4. 跨平台视频播放器

在不同的操作系统平台(Win,Linux)上播放不同格式的视频(AVI,AVI,AVI,RMVB)



**5. 优缺点：**

优点：

1. 分离抽象接口与实现部分，解耦
2. 与多层继承结构相比，桥接模式符合**单一职责原**则，复用性好，减少了系统中类的个数
3. 提高了系统的可扩充性，符合开闭原则，可以在多个变化维度中任意扩展一个维度，而不需要修改原有系统。
4. 实现细节对用户**透明**，可以隐藏实现细节。

在考试中对设计原则进行考察，在某些设计模式中间如何体现设计原则

缺点：

1. 增加系统的理解与设计难度
2. 其使用范围具有一定局限性。要求正确识别出系统中**两个独立变化的维度**

6.使用环境：

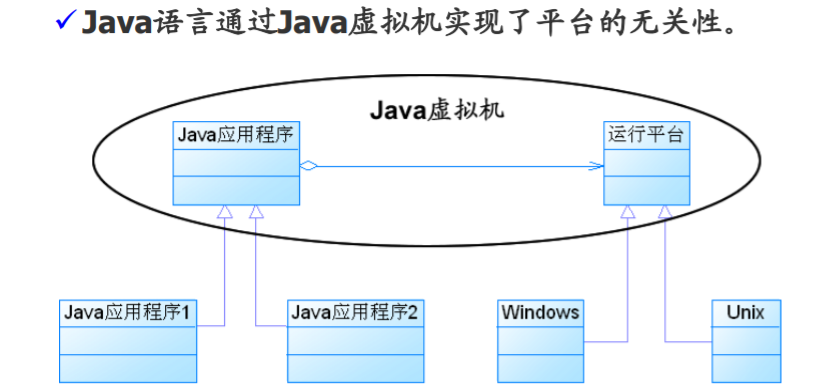
若系统需要在构件的抽象化角色和具体化角色之间增加更多的灵活性，避免在两个层次之间建立静态的继承联系。桥接模式可以使他们在抽象层建立一个关联关系。

存在两个独立维度，且它们都需要扩展。

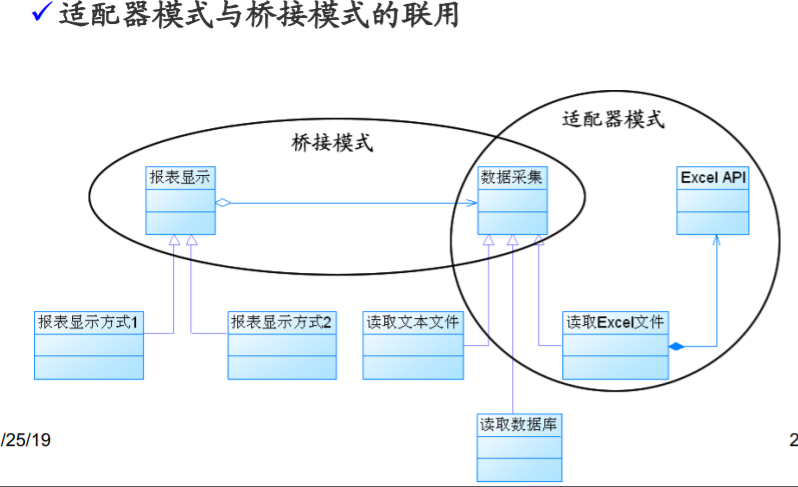
抽象化角色与实现化角色可以以继承的方式独立扩展而互不影响。

不希望使用继承或因为多层次继承容易导致系统类个数急剧增加。

7. 应用实例：



**适配器模式与桥接模式的联用——**两者处于设计的不同阶段，桥接模式用于系统的初步设计，先运用桥接模式将两个变化维度的类分别分为抽象化和实现化两种角色；此后，若系统与现有的类无法协同工作时，采用适配器模式。若涉及到大量第三方接口，适配器模式也要在设计初期考虑。

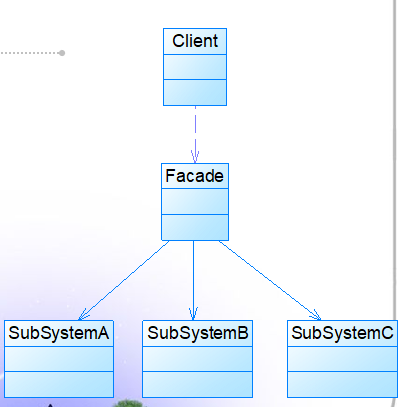


## **九、外观模式**

考察与思想有关的内容 外观模式需要掌握的：外观模式的目的（什么时候使用），功能简化和新行为时候的界限 不太容易考

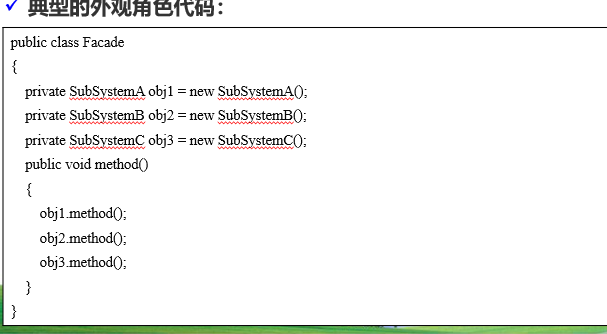
1. **模式动机：**

引入外观角色后，用户只需要直接与外观角色交互，用户与子系统之间的复杂关系由外观角色来实现，降低了系统的耦合度。



1. **原则应用：**
2. 单一职责原则。系统划分为多个子系统有利于降低系统的复杂性。外观模式中，将系统划分出多个子系统，使他们之间通信和相互依赖关系达到最小。外观对象为子系统的访问提供了一个简单又单一的入口。
3. 迪米特法则——对象对另一个对象应尽可能少地了解，不和陌生人说话。 通过引入外观类，降低了原有系统的复杂度，**同时降低客户端与子系统类的耦合度。即客户端只需要与外观对象打交道，无需了解子系统内部的细节，不需要与子系统内部的若干对象打交道。**

3. 模式分析：如上



4.

**优点：**

**对客户屏蔽子系统组件，客户处理对象数目减少，使得子系统使用起来更加容易**

**实现了子系统与客户之间的松耦合。子系统的组件变化不会影响客户类，只需调整外观类。**

**降低编译依赖性，简化系统的移植过程。可以独立地编译子系统。**

**不影响用户直接使用子系统类。**

**缺点：**

**不能很好地限制客户使用子系统类**

**违背了“开闭原则”，不引入抽象外观类的情况下，添加子系统可能要修改外观类。**

5. 使用环境：为一个复杂系统提供一个简单接口时

客户程序与多个子系统之间存在很大的依赖性

层次化的结构中使用外观模式，使用外观模式定义每层的入口，层与层之间不直接产生联系，通过外观类建立联系，降低层间耦合度。

6. 模式扩展：不要试图通过外观类为子系统增加新行为

在多个子系统之间 加入新功能，外观类肯定不能单独完成。先加入新的功能的模式，之后通过外观类来实现

7. 抽象外观类的引入：



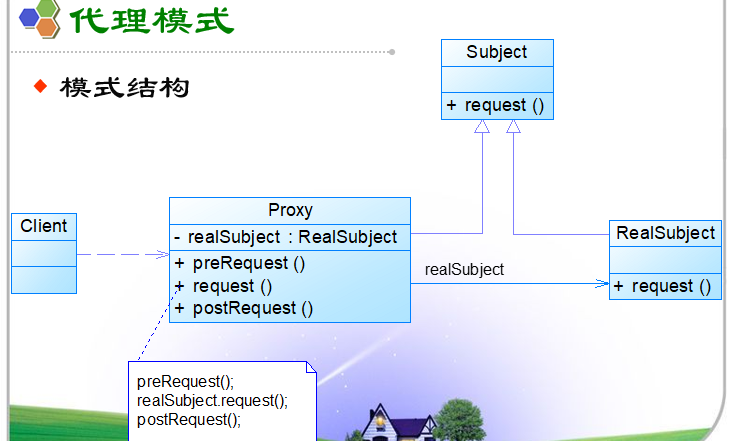
## 十、代理模式

**Proxy/Surrogate）（代理要考的内容比较多）**

**1. 模式动机：**

代理对象在客户端和目标对象之间起到中介的作用，并且可以通过代理对象去掉客户不能看到的内容和服务或者客户需要的额外服务。

定义通过引入代理对象来间接访问一个对象，由代理对象控制对原对象的引用。



2. RMI远程代理：几个步骤，如何创建RMI中间的start 和stuals

**哪个是原来的对象，哪个是客户，如何查找代理，明确谁是代理，代理的接口是那个**

6. 模式分析：

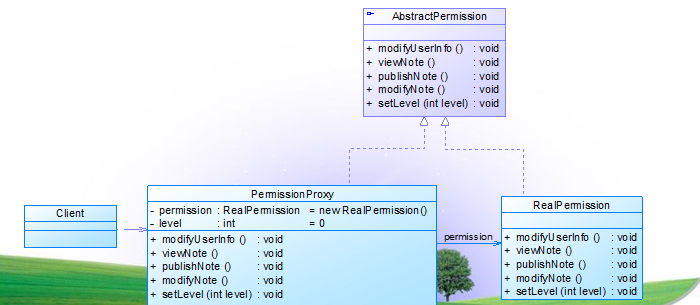


7. 应用实例：五个步骤RMI

计算器



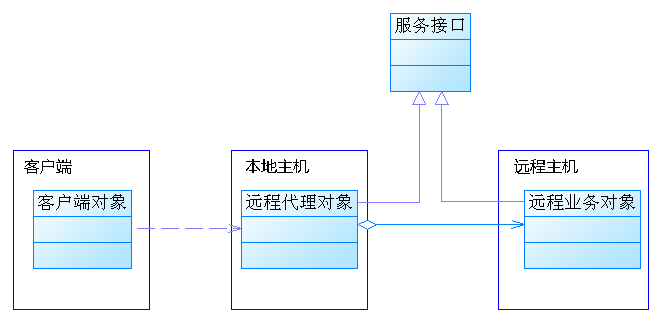
论坛权限控制代理



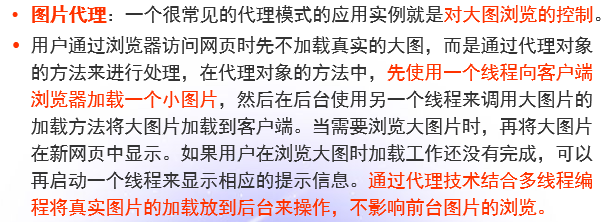
9. 考察三种代理：保护代理（权限）、远程代理、虚拟代理

10. 优点：协调调用者和被调用者，降低耦合度

1） 远程代理（RMI） Client可以访问不同地址空间的对象，可能是同主机或远程机器上的对象



2） 虚拟代理（以小的图片代替大图片，Copy-On-Write） 小对象代表大对象，减少系统资源的消耗，对系统优化并提高响应速度。真正的大对象在需要时才会被真正创建



3） 保护对象（论坛访问权限） 控制对真实对象的使用权限

缺点：复杂性。可能造成请求的处理速度变慢

11. 不考：适用环境：远程代理

12. 模式扩展：图片处理：对大图的浏览

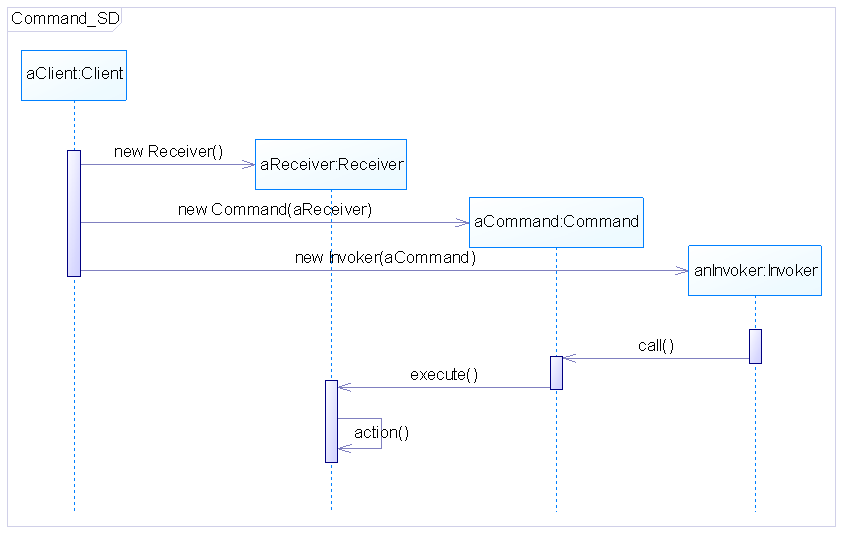
在哪种场景下，通过简单的方式识别场景，特例：远程代理（注意的点），虚拟代理（识别，具体实现有哪些需要注意的点）

## **十一、命令模式**

（Command/Action/Transaction Pattern）：考察的重点，命令模式的角色以及各自的职责（invoker/Receiver），命令模式与其他模式的联用，顺序图不需要掌握，**只需要掌握UML图（各个关系怎么画，虚线实线箭头）和状态图**

1. 模式动机：请求发送者与请求接收者解耦。客户端只需在程序运行时指定具体的请求接收者





2. 模式定义：将请求封装为一个对象，用不用的请求对客户进行参数化

支持可撤销的动作

宏命令

4. 模式分析：对命令进行封装

给一个场景，让系统执行不同的任务，使用命令模式对任务进行封装。

5. 应用：遥控器



6. 优点：降低系统的耦合性,新的命令很容易加到系统中(开闭)

7．使用环境：需要将请求调用者和请求接受者解耦

需要对不同的时间指定请求，将请求排队和执行请求

需要支持命令撤销和恢复操作

需要将一组操作组合在一起，即支持宏命令（数组来实现：组合模式）

8. 撤销操作如何实现

Undo（Redo）



命令队列和宏命令（MacroCommand，又称为组合命令，是命令模式和组合模式联用） 需要掌握

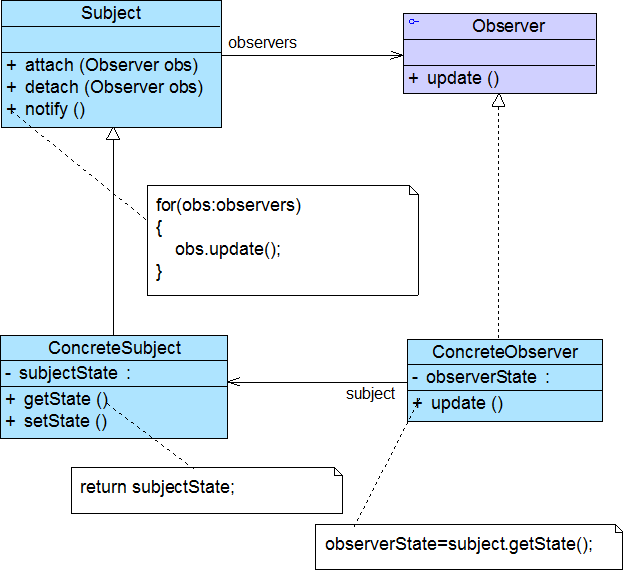
在调用宏命令的execute（）方法时，将递归调用他所包含的每个成员命令的execute（）方法



## 十二、观察者模式

（Observer/ (Publish-Subscribe)/Model-View/Dependents/Source-Listener） 重点考察！！！！

1. **模式动机：**建立对象与对象之间的依赖关系，一个对象发生改变时自动通知其他对象，其他对象将相应做出反应。一个观察目标可以对应多个观察者，可以根据需要增加和删除观察者，使得系统易于扩展。
2. 模式结构：UML中的实线和虚线 Subject中少了一个 #observers



1. 模式实例：

假设猫是老鼠和狗的观察目标，老鼠和狗是观察者，猫叫老鼠跑，狗也跟着叫，使用观察者模式描述该过程。



1. 模式

优点：符合“开闭原则”——观察者的增加和删除，主题的增加和删除

实现表示层和数据逻辑层分离

支持广播通信

缺点：将所有观察者通知会花费很多时间

若观察者与目标产生循环依赖的话，可能导致系统崩溃

没有相应机制让观察者知道目标是怎么变化的，只是知道目标发生变化了而已

1. 适用环境：一个方面依赖另一个方面 ，将这些方面封装在独立对象

一个对象改变将导致其他对象改变，不知道具体有多少对象改变

一个对象必须统治其他对象，且不知道这些对象是谁

需要在系统中创建一个触发链——**创建链式触发机制**

1. 模式扩展：JAVA语言提供的对观察者模式的支持

主题的引用有什么用处

MVC模式 要考！！！组合，策略，观察者在MVC中是如何联用的

1. 观察-通知是一个迭代的过程，可以采用组合-迭代的方式通知观察者、
2. 只要涉及到容器，都可以转换到组合模式（观察者模式、命令模式）
3. 观察者模式拓展到MVC模式 当Model改变时，View自动改变显示内容

观察目标——Model

观察者——View

中介者——Controller

1. java的观察者具体采用了哪些选择策略（推/拉/内部/外部）、原因（优缺点、JAVA语言本身的限制）



考试内容：

内部和外部实现的问题，pattern in the pattern，如何判断内部和外部实现，观察观察者的位置。内部和外部的好处和缺点，对update的方法，谁去发起update的方法

观察者怎么用，如何与其他模式联用，如MVC这样的情况（不会用MVC的例子），观察者的实现细节：推和拉，谁是Update的发起者，哪里存储这些。

## 十三、状态模式

1. 模式动机：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，UML类图 考察内容，UML顺序图，状态图
2. 模式定义：对象行为模式，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
3. 模式结构图：Context和State的关系，context保有state的引用。



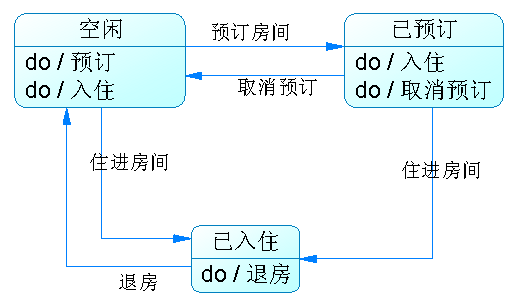
1. 模式分析：引入一个抽象类来专门表示对象的状态。

有几种情况下，在某种状态下的状态切换才是可以的（注意题目的约束，具体哪一种可以）

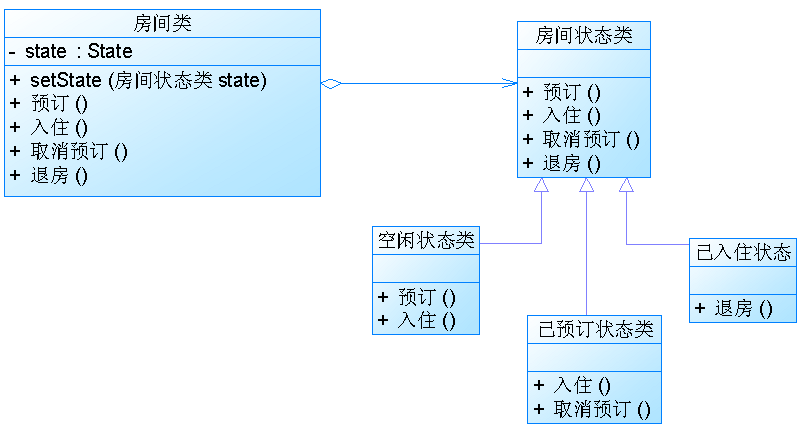
1. 状态模式：切换：通过方法进行切换（课上讲的），根据state进行状态转换，

参考享元的思想，可以用外部状态

状态图：



类图



客户端使用：

if (预定房间){

room.预定();

room.setState(new 已预订状态类);

}else …

在房间类中

class Room{

private State state;

public 预定(){

state.预定();

}

…

}

1. 优点：封装了转换规则 枚举可能的状态 可以方便的增加新状态

允许转态转换逻辑与状态对象合成一体

让多个环境对象共享一个组状态对象从而减少系统中对象的个数

缺点：增加了系统类和对象的个数？？？前后矛盾？

对开闭原则支持不好，添加新的状态，需要修改其他类（前序状态）

1. 适用环境：

对象的行为依赖于它的状态，并且可以根据她的状态改变她的相关行为

包含大量与对象状态有关的条件语句

考点：

1. 根据状态图给出更为灵活的实现
2. 状态模式与策略模式的差异，目的上有什么差异。最重要的是客户使用上的差异，即透明性上

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

策略模式允许客户根据环境或条件的不同，选择不同的具体策略来完成任务。

即状态模式中的状态转换对于用户是透明的，用户不需要关心对象的具体状态，也不需要关心状态是如何完成转换的。状态模式封装了当前的状态以及状态转换的行为。

而策略模式仅仅封装了算法，提供新算法插入到已有系统中，以及老算法从系统中退休的方便。模式本身不决定何时何地试用何种算法，算法的选择由客户端决定，因此策略模式中客户端必须明确指定试用何种具体策略，也就是策略不能对客户端透明。

3. 透明性问题——状态要对客户透明，策略模式中的策略不能对客户透明

4. 状态复用、状态转换

5. 要知道什么场景使用状态模式，状态模式在实现细节上的问题（状态转换）

## **十四、策略模式**

对模式进行简单的考察

1. 模式动机：

对独立的类封装不同的算法，对它进行动态的切换。不同的情况下指定不同的策略。与状态模式不同，给定一个环境对应一个策略，动态的替换相对少一些。

1. 模式结构：



1. 模式分析：由客户端决定什么情况下使用具体策略角色。
2. 模式实例：

排序



出游



1. 优点：对“开闭原则”的完美支持

提供了（管理相关的算法族/可以替换继承关系）的办法

避免使用多重条件转移语句

缺点：造成多种策略类，通过享元（Flyweight）模式在一定程度上减少类

1. 策略模式与状态模式的比较：手机上照片
   * 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（不透明），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（透明）。
     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。

了解策略模式的原则，设计原则在策略模式中如何体现的

## 十五、模板方法模式

1. 模式动机：

基于继承的代码复用，将相同的代码放到父类，不同的方法实现放在子类中

1. 模式定义：定义算法的骨架，将一些步骤延迟到子类。不改变一个算法的结构即可重定义算法的某些步骤
2. 模式结构：



1. 模式分析：面向对象的多态性，只有类的继承，没有对象的关联
2. 实例

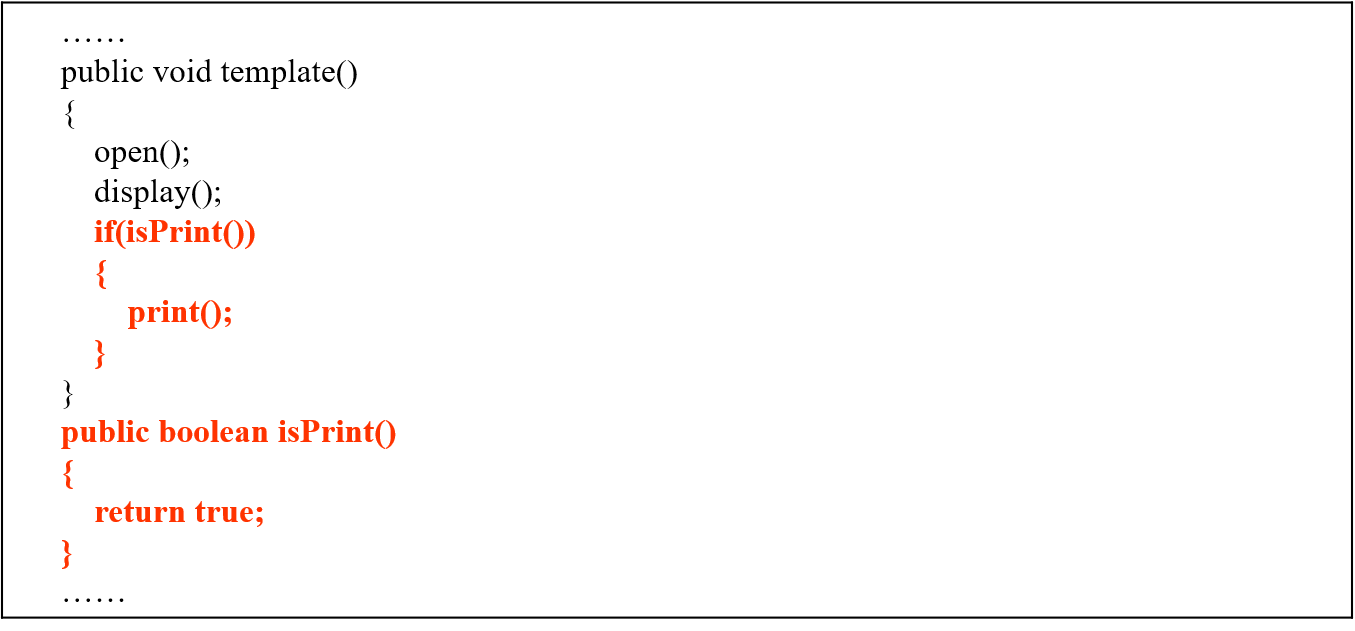
银行业务办理



1. 优点：由子类实现细节处理

由父类实现算法骨架

反向控制结构——通过钩子方法（isPrint()）实现，子类可以通过重写狗子方法可以控制父类的执行



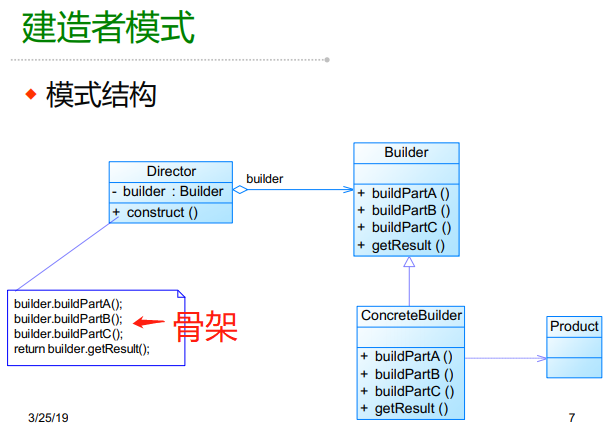
符合开闭原则。单一职责原则（子类、父类）高内聚

缺点：类的增加

1. 适用环境：
   * 1. 一次性实现一个算法的不变的部分，并将可变的行为留给子类来实现。
     2. 各子类中公共的行为应被提取出来并集中到一个公共父类中以避免代码重复。
     3. 对一些复杂的算法进行分割，将其算法中固定不变的部分设计为模板方法和父类具体方法，而一些可以改变的细节由其子类来实现。
     4. 控制子类的扩展。
2. 好莱坞原则 Don`t call us, we`ll call you

子类不显式调用父类的方法，而是通过覆盖父类的方法来实现某些具体的业务逻辑，**父类控制对子类的调用。**

1. 钩子方法：几种实现：返回值，抛出异常
2. 与其他模式联用，工厂模式 建造者中的指挥者。考察在其他模式中的体现



抽象工厂模式

