# 私塾在线《研磨设计模式》
## ——跟着CC学设计系列精品教程

# 本节课程概览

- 相关模式

  关于相关模式的辨析、比较、选择以及组合使用等。

# 提前说明

- 模式选择的方法

1：模式的功能——看是否能解决问题
2：模式的本质——看模式是否主要用来解决这类问题
3：模式的适用程度——看使用这个模式是否贴切，看是否需要变形
4：应用模式的复杂程度——看使用模式带来的开发复杂度是否可接受或可控
5：应用模式的代价——看使用模式的代价是否可接受或可控，如：引入过多的对象、耗费更多的内存等

- 很多模式都可以组合使用，并不局限于这里要讲的

- 有些模式功能相近，结构类似，要特别注意他们的辨析

- 掌握模式一定要从思想上、本质上、整体上去把握

# 简单工厂的相关模式

- 简单工厂和抽象工厂模式

简单工厂是用来选择实现的，可以选择任意接口的实现，一个简单工厂可以有多个用于选择并创建对象的方法，多个方法创建的对象可以有关系也可以没有关系。

抽象工厂模式是用来选择产品簇的实现的，也就是说一般抽象工厂里面有多个用于选择并创建对象的方法，但是这些方法所创建的对象之间通常是有关系的，这些被创建的对象通常是构成一个产品簇所需要的部件对象。

所以从某种意义上来说，简单工厂和抽象工厂是类似的，如果抽象工厂退化成为只有一个实现，不分层次，那么就相当于简单工厂了。

# 简单工厂的相关模式

- 简单工厂和工厂方法模式

  简单工厂和工厂方法模式也是非常类似的。

  工厂方法的本质也是用来选择实现的，跟简单工厂的区别在于工厂方法是把选择具体实现的功能延迟到子类去实现。

  如果把工厂方法中选择的实现放到父类直接实现，那就等同于简单工厂

- 简单工厂和能创建对象实例的模式

  简单工厂的本质是选择实现，所以它可以跟其它任何能够具体的创建对象实例的模式配合使用，比如：单例模式、原型模式、生成器模式等等。

# 外观模式的相关模式

- 外观模式和中介者模式

  这两个模式非常类似，但是有本质的区别。

  中介者模式主要用来封装多个对象之间相互的交互，多用在系统内部的多个模块之间；而外观模式封装的是单向的交互，是从客户端访问系统的调用，没有从系统中来访问客户端的调用。

  在中介者模式的实现里面，是需要实现具体的交互功能的；而外观模式的实现里面，一般是组合调用或是转调内部实现的功能，通常外观模式本身并不实现这些功能。

  中介者模式的目的主要是松散多个模块之间的耦合，把这些耦合关系全部放到中介者中去实现；而外观模式的目的是简化客户端的调用，这点和中介者模式也不同。

# 外观模式的相关模式

- 外观模式和单例模式

  通常一个子系统只需要一个外观实例，所以外观模式可以和单例模式组合使用，把Facade类实现成为单例。当然，也可以跟前面示例的那样，把外观类的构造方法私有化，然后把提供给客户端的方法实现成为静态的

- 外观模式和抽象工厂模式

  外观模式的外观类通常需要和系统内部的多个模块交互，每个模块一般都有自己的接口，所以在外观类的具体实现里面，需要获取这些接口，然后组合这些接口来完成客户端的功能。

  那么怎么获取这些接口呢？就可以和抽象工厂一起使用，外观类通过抽象工厂来获取所需要的接口，而抽象工厂也可以把模块内部的实现对Facade进行屏蔽，也就是说Facade也仅仅只是知道它从模块中获取的它需要的功能，模块内部的细节，Facade也不知道了。

# 适配器模式的相关模式

- 适配器模式与桥接模式

  其实这两个模式除了结构略为相似外，功能上完全不同。

  适配器模式是把两个或者多个接口的功能进行转换匹配；而桥接模式是让接口和实现部分相分离，以便它们可以相对独立的变化。

- 适配器模式与装饰模式

  从某种意义上讲，适配器模式能模拟实现简单的装饰模式的功能，也就是为已有功能增添功能。

  造成这种类似的原因：两种设计模式在实现上都是使用的对象组合，都可以在转调组合对象的功能前后进行一些附加的处理，因此有这么一个相似性。它们的目的和本质都是不一样的。

  两个模式有一个很大的不同：一般适配器适配过后是需要改变接口的，如果不改接口就没有必要适配了；而装饰模式是不改接口的，无论多少层装饰都是一个接口。

  因此装饰模式可以很容易的支持递归组合，而适配器就做不到了，每次的接口不同，没法递归。

# 适配器模式的相关模式

- 适配器模式和代理模式

  适配器模式可以跟代理模式组合使用，在实现适配器的时候，可以通过代理来调用Adaptee，这可以获得更大的灵活性

- 适配器模式和抽象工厂模式

  在适配器实现的时候，通常需要得到被适配的对象，如果被适配的是一个接口，那么就可以结合一些可以创造对象实例的设计模式，来得到被适配的对象示例。比如：抽象工厂模式、单例模式、工厂方法模式等

# 单例模式的相关模式

- 相关模式

  很多模式都可以使用单例模式，只要这些模式中的某个类，需要控制实例为一个的时候，就可以很自然的使用上单例模式。比如抽象工厂方法中的具体工厂类就通常是一个单例。

# 工厂方法模式的相关模式

- 工厂方法模式和抽象工厂模式

  这两个模式可以组合使用，具体的放到抽象工厂模式中去讲。

- 工厂方法模式和模板方法模式

  这两个模式外观类似，都是有一个抽象类，然后由子类来提供一些实现，但是工厂方法模式的子类专注的是创建产品对象，而模板方法模式的子类专注的是为固定的算法骨架提供某些步骤的实现。

  这两个模式可以组合使用，通常在模板方法模式里面，使用工厂方法来创建模板方法需要的对象。

# 抽象工厂模式的相关模式

- 抽象工厂模式和工厂方法模式

  这两个模式既有区别，又有联系，可以组合使用。

  工厂方法模式一般是针对单独的产品对象的创建，而抽象工厂模式注重产品簇对象的创建，这是它们的区别。

  如果把抽象工厂创建的产品簇简化，这个产品簇就只有一个产品，那么这个时候的抽象工厂跟工厂方法是差不多的，也就是抽象工厂可以退化成工厂方法，而工厂方法又可以退化成简单工厂，这是它们的联系。

  在抽象工厂的实现中，还可以使用工厂方法来提供抽象工厂的具体实现，也就是说它们可以组合使用。

- 抽象工厂模式和单例模式

  这两个模式可以组合使用。

  在抽象工厂模式里面，具体的工厂实现，在整个应用中，通常一个产品系列只需要一个实例就可以了，因此可以把具体的工厂实现成为单例。

# 生成器模式的相关模式

- 生成器模式和工厂方法模式

  这两个模式可以组合使用。

  生成器模式的Builder实现中，通常需要选择具体的部件实现，一个可行的方案就是实现成为工厂方法，通过工厂方法来获取具体的部件对象，然后再进行部件的装配。

- 生成器模式和抽象工厂模式

  这两个模式既相似又有区别，也可以组合使用

  先说相似性，这个在课程里面已经详细讲述了，这里就不去重复了。

  再说说区别：抽象工厂模式的主要目的是创建产品簇，这个产品簇里面的单个产品，就相当于是构成一个复杂对象的部件对象，抽象工厂对象创建完成过后就立即返回整个产品簇；而生成器模式的主要目的是按照构造算法，一步一步来构建一个复杂的产品对象，通常要等到整个构建过程结束过后，才会得到最终的产品对象。

# 生成器模式的相关模式

事实上，这两个模式是可以组合使用的，在生成器模式的Builder实现中，需要创建各个部件对象，而这些部件对象是有关联的，通常是构成一个复杂对象的部件对象，也就是说，Builder实现中，需要获取构成一个复杂对象的产品簇，那自然就可以使用抽象工厂模式来实现。这样一来，由抽象工厂模式负责了部件对象的创建，Builder实现里面就主要负责产品对象整体的构建了。

- 生成器模式和模板方法模式

这也是两个非常类似的模式。初看之下，不会觉得这两个模式有什么关联，但是仔细一思考，发现两个模式在功能上很类似。

模板方法模式主要是用来定义算法的骨架，把算法中某些步骤延迟到子类中实现。再想想生成器模式，Director用来定义整体的构建算法，把算法中某些涉及到具体部件对象的创建和装配的功能，委托给具体的Builder来实现。

# 生成器模式的相关模式

虽然生成器不是延迟到子类，是委托给Builder，但那只是具体实现方式上的差别，从实质上看两个模式很类似，都是定义一个固定的算法骨架，然后把算法中的某些具体步骤交给其它类来完成，都能实现整体算法步骤和某些具体步骤实现的分离。

当然两个模式也有很大的区别，首先是模式的目的，生成器模式是用来构建复杂对象的，而模板方法是用来定义算法骨架，尤其是一些复杂的业务功能的处理算法的骨架；其次是模式的实现，生成器模式是采用委托的方法，而模板方法是采用的继承的方式；另外从使用的复杂度上，生成器模式需要组合Director和Builder对象，然后才能开始构建，要等构建完后才能获得最终的对象，而模板方法就没有这么麻烦，直接使用子类对象即可。

- 生成器模式和组合模式

这两个模式可以组合使用。

对于复杂的组合结构，可以使用生成器模式来一步一步构建。

# 原型模式的相关模式

- 原型模式和抽象工厂模式

  功能上有些相似，都是用来获取一个新的对象实例的。

  不同之处在于，原型模式的着眼点是在如何创造出实例对象来，最后选择的方案是通过克隆；而抽象工厂模式的着眼点则在于如何来创造产品簇，至于具体如何创建出产品簇中的每个对象实例，抽象工厂模式不是很关注。

  正是因为它们的关注点不一样，所以它们也可以配合使用，比如在抽象工厂模式里面，具体创建每一种产品的时候就可以使用该种产品的原型，也就是抽象工厂管产品簇，具体的每种产品怎么创建则可以选择原型模式。

- 原型模式和生成器模式

  这两种模式可以配合使用。

  生成器模式关注的是构建的过程，而在构建的过程中，很可能需要某个部件的实例，那么很自然地就可以应用上原型模式，通过原型模式来得到部件的实例。

# 中介者模式的相关模式

- 中介者模式和外观模式

  这两个模式有相似的地方，也存在很大的不同。

  外观模式多用来封装一个子系统内部的多个模块，目的是向子系统外部提供简单易用的接口，也就是说外观模式封装的是子系统外部和子系统内部模块间的交互；而中介者模式是提供多个平等的同事对象之间交互关系的封装，一般是用在内部实现上。

  另外，外观模式是实现单向的交互，是从子系统外部来调用子系统内部，不会反着来，而中介者模式实现的是内部多个模块间多向的交互。

- 中介者模式和观察者模式

  这两个模式可以组合使用。

  中介者模式可以组合使用观察者模式，来实现当同事对象发生改变的时候，通知中介对象，让中介对象去进行与其它相关对象的交互 。

# 代理模式的相关模式

- 代理模式和适配器模式

  这两个模式有一定的相似性，但也有差异。

  这两个模式有相似性，它们都为另一个对象提供间接性的访问，而且都是从自身以外的一个接口向这个对象转发请求。

  但是从功能上，两个模式是不一样的。适配器模式主要用来解决接口之间不匹配的问题，它通常是为所适配的对象提供一个不同的接口；而代理模式会实现和目标对象相同的接口。

- 代理模式和装饰模式

  这两个模式从实现上相似，但是功能上是不同的。

  装饰模式的实现和保护代理的实现上是类似的，都是在转调其它对象的前后执行一定的功能。但是它们的目的和功能都是不同的。

  装饰模式的目的是为了让你不生成子类就可以给对象添加职责，也就是为了动态的增加功能；而代理模式的主要目的是控制对对象的访问。

# 观察者模式的相关模式

- 观察者模式和状态模式

  观察者模式和状态模式是有相似之处的。

  观察者模式是当目标状态发生改变时，触发并通知观察者，让观察者去执行相应的操作。而状态模式是根据不同的状态，选择不同的实现，这个实现类的主要功能就是针对状态的相应的操作，它不像观察者，观察者本身还有很多其它的功能，接收通知并执行相应处理只是观察者的部分功能。

  当然观察者模式和状态模式是可以结合使用的。观察者模式的重心在触发联动，但是到底决定哪些观察者会被联动，这时就可以采用状态模式来实现了，也可以采用策略模式来进行选择需要联动的观察者。

# 观察者模式的相关模式

- 观察者模式和中介者模式

观察者模式和中介者模式是可以结合使用的。

前面的例子中目标都只是简单的通知一下，然后让各个观察者自己去完成更新就结束了。如果观察者和被观察的目标之间的交互关系很复杂，比如：有一个界面，里面有三个下拉列表组件，分别是选择国家、省份/州、具体的城市，很明显这是一个三级联动，当你选择一个国家的时候，省份/州应该相应改变数据，省份/州一改变，具体的城市也需要改变。

这种情况下，很明显需要相关的状态都联动准备好了，然后再一次性的通知观察者，就是界面做更新处理，不会国家改变一下，省份和城市还没有改，就通知界面更新。这种情况就可以使用中介者模式来封装观察者和目标的关系。

在使用Swing的小型应用里面，也可以使用中介者模式。比如：把一个界面所有的事件用一个对象来处理，把一个组件触发事件过后，需要操作其它组件的动作都封装到一起，这个对象就是典型的中介者。

# 命令模式的相关模式

- 命令模式和组合模式

  这两个模式可以组合使用。

  在命令模式中，实现宏命令的功能，就可以使用组合模式来实现。前面的示例并没有按照组合模式来做，那是为了保持示例的简单，还有突出命令模式的实现，这点请注意。

- 命令模式和备忘录模式

  这两个模式可以组合使用。

  在命令模式中，实现可撤销操作功能时，前面讲了有两种实现方式，其中有一种就是保存命令执行前的状态，撤销的时候就把状态恢复回去。如果采用这种方式实现，就可以考虑使用备忘录模式。

  如果状态存储在命令对象里面，那么还可以使用原型模式，把命令对象当作原型来克隆一个新的对象，然后把克隆出来的对象通过备忘录模式存放。

# 命令模式的相关模式

- 命令模式和模板方法模式

这两个模式从某种意义上有相似的功能，命令模式可以作为模板方法的一种替代模式，也就是说命令模式可以模仿实现模板方法模式的功能。

如同前面讲述的退化的命令模式可以实现Java的回调，而Invoker智能化后向服务进化，如果Invoker的方法就是一个算法骨架，其中有两步在这个骨架里面没有具体实现，需要外部来实现，这个时候就可以通过回调命令接口来实现。

而类似的功能在模板方法里面，一个算法骨架，其中有两步在这个骨架里面没有具体实现，是先调用抽象方法，然后等待子类来实现。

可以看出虽然实现方式不一样，但是可以实现相同的功能。

# 迭代器模式的相关模式

- 迭代器模式和组合模式

  这两个模式可以组合使用。

  组合模式是一种递归的对象结构，在枚举某个组合对象的子对象的时候，通常会使用迭代器模式。

- 迭代器模式和工厂方法模式

  这两个模式可以组合使用。

  在聚合对象创建迭代器的时候，通常会采用工厂方法模式来实例化相应的迭代器对象。

# 组合模式的相关模式

- 组合模式和装饰模式

  这两个模式可以组合使用。

  装饰模式在组装多个装饰器对象的时候，是一个装饰器找下一个装饰器，下一个再找下一个，如此递归下去。那么这种结构也可以使用组合模式来帮助构建，这样一来，装饰器对象就相当于组合模式的Composite对象了。

  要让两个模式能很好的组合使用，通常会让它们有一个公共的父类，因此装饰器必须支持组合模式需要的一些功能，比如：增加、删除子组件等等。

- 组合模式和享元模式

  这两个模式可以组合使用。

  如果组合模式中出现大量相似的组件对象的话，可以考虑使用享元模式来帮助缓存组件对象，这可以减少对内存的需要。

  使用享元模式也是有条件的，如果组件对象的可变化部分的状态能够从组件对象里面分离出去，而且组件对象本身不需要向父组件发送请求的话，就可以采用享元模式。

# 组合模式的相关模式

- 组合模式和迭代器模式

  这两个模式可以组合使用。

  在组合模式中，通常可以使用迭代器模式来遍历组合对象的子对象集合，而无需关心具体存放子对象的聚合结构。

- 组合模式和访问者模式

  这两个模式可以组合使用。

  访问者模式能够在不修改原有对象结构的情况下，给对象结构中的对象增添新的功能。将访问者模式和组合模式合用，可以把原本分散在Composite和Leaf类中的操作和行为都局部化。

  如果在使用组合模式的时候，预计到今后可能会有增添其它功能的可能，那么可以采用访问者模式，来预留好添加新功能的方式和通道，这样以后在添加新功能的时候，就不需要再修改已有的对象结构和已经实现的功能了。

# 组合模式的相关模式

- 组合模式和职责链模式

  这两个模式可以组合使用。

  职责链模式要解决的问题是：实现请求的发送者和接收者之间解耦。职责链模式的实现方式是把多个接收者组合起来，构成职责链，然后让请求在这条链上传递，直到有接收者处理这个请求为止。

  可以应用组合模式来构建这条链，相当于是子组件找父组件，父组件又找父组件，如此递归下去，构成一条处理请求的组件对象链。

- 组合模式和命令模式

  这两个模式可以组合使用。

  命令模式中有一个宏命令的功能，通常这个宏命令就是使用组合模式来组装出来的。

# 模板方法模式的相关模式

- 模板方法模式和工厂方法模式

  这两个模式可以配合使用。

  模板方法模式可以通过工厂方法来获取需要调用的对象。。

- 模板方法模式和策略模式

  这两个模式的功能有些相似，但是是有区别的。

  从表面上看，两个模式都能实现算法的封装，但是模板方法封装的是算法的骨架，这个算法骨架是不变的，变化的是算法中某些步骤的具体实现；而策略模式是把某个步骤的具体实现算法封装起来，所有封装的算法对象是等价的，可以相互替换。

  因此，可以在模板方法中使用策略模式，就是把那些变化的算法步骤通过使用策略模式来实现，但是具体选取哪个策略还是要由外部来确定，而整体的算法步骤，也就是算法骨架就由模板方法来定义了。

# 策略模式的相关模式

- 策略模式和状态模式

  这两个模式从模式结构上看是一样的，但是实现的功能是不一样的。

  状态模式是根据状态的变化来选择相应的行为，不同的状态对应不同的类，每个状态对应的类实现了该状态对应的功能，在实现功能的同时，还会维护状态数据的变化。这些实现状态对应的功能的类之间是不能相互替换的。

  策略模式是根据需要或者是客户端的要求来选择相应的实现类，各个实现类是平等的，是可以相互替换的。

  另外策略模式可以让客户端来选择需要使用的策略算法，而状态模式一般是由上下文，或者是在状态实现类里面来维护具体的状态数据，通常不由客户端来指定状态。

- 策略模式和模板方法模式

  这两个模式可组合使用，如同前面示例的那样。

  模板方法重在封装算法骨架，而策略模式重在分离并封装算法实现。

# 策略模式的相关模式

- 策略模式和享元模式

  这两个模式可组合使用。

  策略模式分离并封装出一系列的策略算法对象，这些对象的功能通常都比较单一，很多时候就是为了实现某个算法的功能而存在，因此，针对这一系列的、多个细粒度的对象，可以应用享元模式来节省资源，但前提是这些算法对象要被频繁的使用，如果偶尔用一次，就没有必要做成享元了。

# 状态模式的相关模式

- 状态模式和观察者模式

这两个模式乍一看，功能是很相似的，但是又有区别，可以组合使用。

模式都是在状态发生改变的时候触发行为，只不过观察者模式的行为是固定的，那就是通知所有的观察者，而状态模式是根据状态来选择不同的处理。

从表面来看两个模式功能相似，观察者中的被观察对象就好比状态模式中的上下文，观察者模式中当被观察对象的状态发生改变的时候，触发的通知所有观察者的方法；就好比状态模式中，根据状态的变化，选择对应的状态处理。

但实际这两个模式是不同的，观察者模式的目的是在被观察者的状态发生改变的时候，触发观察者联动，具体如何处理观察者模式不管；而状态模式的主要目的在于根据状态来分离和选择行为，当状态发生改变的时候，动态改变行为。

这两个模式是可以组合使用的，比如在观察者模式的观察者部分，当被观察对象的状态发生了改变，触发通知了所有的观察者过后，观察者该怎么处理呢？这个时候就可以使用状态模式，根据通知过来的状态选择相应的处理。

# 状态模式的相关模式

- 状态模式和策略模式

  这是两个结构相同，功能各异的模式，具体的在策略模式里面讲过了，这里不再赘述。

- 状态模式和单例模式

  这两个模式可以组合使用，可以把状态模式中的状态处理类实现成单例。

- 状态模式和享元模式

  这两个模式可以组合使用。

  由于状态模式把状态对应的行为分散到多个状态对象中，会造成很多细粒度的状态对象，可以把这些状态处理对象通过享元模式来共享，从而节省资源。

# 备忘录模式的相关模式

- 备忘录模式和命令模式

  这两个模式可以组合使用。

  命令模式实现中，在实现命令的撤销和重做的时候，可以使用备忘录模式，在命令操作的时候记录下操作前后的状态，然后在命令撤销和重做的时候，直接使用相应的备忘录对象来恢复状态就可以了。

  在这种撤销的执行顺序和重做执行顺序可控的情况下，备忘录对象还可以采用增量式记录的方式，可以减少缓存的数据量。

- 备忘录模式和原型模式

  这两个模式可以组合使用。

  在原发器对象创建备忘录对象的时候，如果原发器对象中全部或者大部分的状态都需要保存，一个简洁的方式就是直接克隆一个原发器对象。也就是说，这个时候备忘录对象里面存放的是一个原发器对象的实例，这个在前面已经示例过了，这里就不赘述了。

# 享元模式的相关模式

- 享元模式与单例模式

  这两个模式可以组合使用。

  通常情况下，享元模式中的享元工厂可以实现成为单例。另外，享元工厂里面缓存的享元对象，都是单实例的，可以看成是单例模式的一种变形控制，在享元工厂里面来单例享元对象。

- 享元模式与组合模式

  这两个模式可以组合使用。

  在享元模式里面，存在不需要共享的享元实现，这些不需要共享的享元通常是对共享的享元对象的组合对象，也就是说，享元模式通常会和组合模式组合使用，来实现更复杂的对象层次结构。

# 享元模式的相关模式

- 享元模式与状态模式

  这两个模式可以组合使用。

  可以使用享元模式来共享状态模式中的状态对象，通常在状态模式中，会存在数量很大的、细粒度的状态对象，而且它们基本上都是可以重复使用的，都是用来处理某一个固定的状态的，它们需要的数据通常都是由上下文传入，也就是变化部分都分离出去了，所以可以用享元模式来实现这些状态对象。

- 享元模式与策略模式

  这两个模式可以组合使用。

  可以使用享元模式来实现策略模式中的策略对象，跟状态模式一样，在策略模式中也存在大量细粒度的策略对象，它们需要的数据同样是从上下文传入的，所以可以使用享元模式来实现这些策略对象

# 解释器模式的相关模式

- 解释器模式和组合模式

  这两个模式可以组合使用。

  通常解释器模式都会使用组合模式来实现，这样能够方便的构建抽象语法树。一般非终结符解释器就相当于组合模式中的组合对象，终结符解释器就相当于叶子对象。

- 解释器模式和迭代器模式

  这两个模式可以组合使用。

  由于解释器模式通常使用组合模式来实现，因此在遍历整个对象结构的时候，自然可以使用迭代器模式。

- 解释器模式和享元模式

  这两个模式可以组合使用。

  在使用解释器模式的时候，可能会造成多个细粒度对象，比如会有各种各样的终结符解释器，而这些终结符解释器对不同的表达式来说是一样的，是可以共用的，因此可以引入享元模式来共享这些对象。

# 解释器模式的相关模式

- 解释器模式和访问者模式

  这两个模式可以组合使用。

  在解释器模式中，语法规则和解释器对象是有对应关系的。语法规则的变动意味着功能的变化，自然会导致使用不同的解释器对象；而且一个语法规则可以被不同的解释器解释执行。

  因此在构建抽象语法树的时候，如果每个节点所对应的解释器对象是固定的，这就意味着这个节点对应的功能是固定的，那么就不得不根据需要来构建不同的抽象语法树。

  为了让构建的抽象语法树较为通用，那就要求解释器的功能不要那么固定，要能很方便的改变解释器的功能，这个时候问题就变成了，如何能够很方便的更改树形结构中节点对象的功能了，访问者模式可以很好的实现这个功能。

# 装饰模式的相关模式

- 装饰模式与适配器模式

  这是两个没有什么关联的模式，放到一起来说，是因为它们有一个共同的别名：Wrapper。

  这两个模式功能上是不一样的，适配器模式是用来改变接口的，而装饰模式是用来改变对象功能的。

- 装饰模式与组合模式

  这两个模式有相似之处，都涉及到对象的递归调用，从某个角度来说，可以把装饰看成是只有一个组件的组合。

  但是它们的目的完全不一样，装饰模式是要动态的给对象增加功能；而组合模式是想要管理组合对象和叶子对象，为它们提供一个一致的操作接口给客户端，方便客户端的使用。

# 装饰模式的相关模式

- 装饰模式与策略模式

这两个模式可以组合使用。

策略模式也可以实现动态的改变对象的功能，但是策略模式只是一层选择，也就是根据策略选择一下具体的实现类而已。而装饰模式不是一层，而是递归调用，无数层都可以，只要组合好装饰器的对象组合，那就可以依次调用下去，所以装饰模式会更灵活。

而且策略模式改变的是原始对象的功能，不像装饰模式，后面一个装饰器，改变的是经过前一个装饰器装饰过后的对象，也就是策略模式改变的是对象的内核，而装饰模式改变的是对象的外壳。

这两个模式可以组合使用，可以在一个具体的装饰器里面使用策略模式，来选择更具体的实现方式。比如前面计算奖金的另外一个问题就是参与计算的基数不同，奖金的计算方式也是不同的。举例来说：假设张三和李四参与同一个奖金的计算，张三的销售总额是2万元，而李四的销售额是8万元，它们的计算公式是不一样的，假设奖金的计算规则是，销售额在5万以下，统一3%，而5万以上，5万内是4%，超过部分是6%。

参与同一个奖金的计算，这就意味着可以使用同一个装饰器，但是在装饰器的内部，不同条件下计算公式不一样，那么怎么选择具体的实现策略呢？自然使用策略模式就好了，也就是装饰模式和策略模式组合来使用。

# 装饰模式的相关模式

- 装饰模式与模板方法模式

  这是两个功能上有相似点的模式。

  模板方法模式主要应用在算法骨架固定的情况，那么要是算法步骤不固定呢，也就是一个相对动态的算法步骤，就可以使用装饰模式了，因为在使用装饰模式的时候，进行装饰器的组装，其实也相当于是一个调用算法步骤的组装，相当于是一个动态的算法骨架。

  既然装饰模式可以实现动态的算法步骤的组装和调用，那么把这些算法步骤固定下来，那就是模板方法模式实现的功能了，因此装饰模式可以模拟实现模板方法模式的功能。

  但是请注意，仅仅只是可以模拟功能而已，两个模式的设计目的、原本的功能、本质思想等都是不一样的。

# 职责链模式的相关模式

- 职责链模式和组合模式

  这两个模式可以组合使用。

  可以把职责对象通过组合模式来组合，这样可以通过组合对象自动递归的向上调用，由父组件作为子组件的后继，从而形成链。

  这也就是前面提到过的使用外部已有的链接，这种情况在客户端使用的时候，就不用再构造链了，虽然不构造链，但是需要构造组合对象树，是一样的。

- 职责链模式和策略模式

  这两个模式可以组合使用。

  这两个模式有相似之处，如果把职责链简化到直接就能选择到相应的处理对象，那就跟策略模式的选择差不多，因此可以用职责链来模拟策略模式的功能。只是如果把职责链简化到这个地步，也就不存在链了，也就称不上是职责链了。

  两个模式可以组合使用，可以在职责链模式的某个职责的实现的时候，使用策略模式来选择具体的实现，同样也可以在策略模式的某个策略实现里面，使用职责链模式来实现功能处理。

  同理职责链模式也可以和状态模式组合使用。

# 职责链模式的相关模式

- 职责链模式和装饰模式

这两个模式相似，从某个角度讲，可以相互模拟实现对方的功能。

装饰模式能够动态的给被装饰对象添加功能，要求装饰器对象和被装饰的对象实现相同的接口。而职责链模式可以实现动态的职责组合，标准的功能是有一个对象处理就结束，但是如果处理完本职责不急于结束，而是继续向下传递请求，那么功能就和装饰模式的功能差不多了，每个职责对象就类似于装饰器，可以实现某种功能。

而且两个模式的本质也类似，都需要在运行期间动态组合，装饰模式是动态组合装饰器，而职责链是动态组合处理请求的职责对象的链。

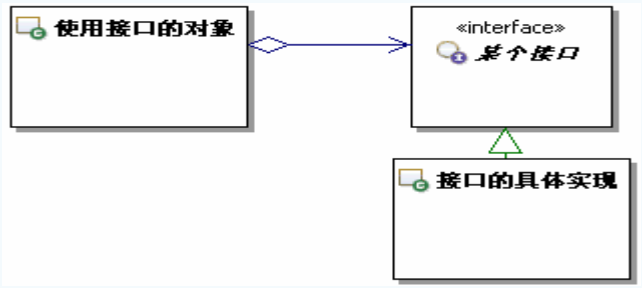
但是从标准的设计模式上来讲，这两个模式还是有较大区别的，这点要注意。首先是目的不同，装饰模式是要实现透明的为对象添加功能，而职责链模式是要实现发送者和接收者解耦；另外一个，装饰模式是无限递归调用的，可以有任意多个对象来装饰功能，但是职责链模式是有一个处理就结束。

# 桥接模式的相关模式

- 桥接模式和策略模式

这两个模式有很大的相似之处。

如果把桥接模式的抽象部分简化来看，如果暂时不去扩展Abstraction，也就是去掉RefinedAbstraction。会发现，这个时候它们的结构都类似如图所示：




通过上面的结构图，可以体会到桥接模式和策略模式是如此相似。可以把策略模式的Context视做是使用接口的对象，而Strategy就是某个接口了，具体的策略实现那就相当于接口的具体实现。这样看来的话，某些情况下，可以使用桥接模式来模拟实现策略模式的功能。

这两个模式虽然相似，也还是有区别的。最主要的是模式的目的不一样，策略模式的目的是封装一系列的算法，使得这些算法可以相互替换；而桥接模式的目的是分离抽象和实现部分，使得它们可以独立的变化。

# 桥接模式的相关模式

- 桥接模式和状态模式

  由于从模式结构上看，状态模式和策略模式是一样的，这两个模式的关系也基本上类似于桥接模式和策略模式的关系。

  只不过状态模式的目的是封装状态对应的行为，并在内部状态改变的时候改变对象的行为。

- 桥接模式和抽象工厂模式

  这两个模式可以组合使用。

  桥接模式中，抽象部分需要获取相应的实现部分的接口对象，那么谁来创建实现部分的具体实现对象呢？这就是抽象工厂模式派上用场的地方。也就是使用抽象工厂模式来创建和配置一个特定的具体实现的对象。

  事实上，抽象工厂主要是用来创建一系列对象的，如果创建的对象很少，或者是很简单，还可以采用简单工厂，可以达到一样的效果，但是会比抽象工厂来得简单。

# 桥接模式的相关模式

- 桥接模式和模板方法模式

  这两个模式有相似之处。

  虽然标准的模板方法模式是采用继承来实现的，但是模板方法也可以通过回调接口的方式来实现，如果把接口的实现独立出去，那就类似于模板方法通过接口去调用具体的实现方法了。这样的结构就和简化的桥接模式类似了。

  可以使用桥接模式来模拟实现模板方法模式的功能。如果在实现Abstraction对象的时候，在里面定义方法，方法里面就是某个固定的算法骨架，也就是说这个方法就相当于模板方法。在模板方法模式里，是把不能确定实现的步骤延迟到子类去实现；现在在桥接模式里面，把不能确定实现的步骤委托给具体实现部分去完成，通过回调实现部分的接口，来完成算法骨架中的某些步骤。这样一来，就可以实现使用桥接模式来模拟实现模板方法模式的功能了。

# 桥接模式的相关模式

使用桥接模式来模拟实现模板方法模式的功能，还有个潜在的好处，就是模板方法也可以很方便的扩展和变化了。在标准的模板方法里面，一个问题就是当模板发生变化的时候，所有的子类都要变化，非常不方便。而使用桥接模式来实现类似的功能，就没有这个问题了。

另外，这里只是说从实现具体的业务功能上，桥接模式可以模拟实现出模板方法模式能实现的功能，并不是说桥接模式和模板方法模式就变成一样的，或者是桥接模式就可以替换掉模板方法模式了。要注意它们本身的功能、目的、本质思想都是不一样的。

# 桥接模式的相关模式

- 桥接模式和适配器模式

  这两个模式可以组合使用。

  这两个模式功能是完全不一样的，适配器模式的功能主要是用来帮助无关的类协同工作，重点在解决原本由于接口不兼容而不能一起工作的那些类，使得它们可以一起工作。而桥接模式则重点在分离抽象和实现部分。

  所以在使用上，通常在系统设计完成过后，才会考虑使用适配器模式；而桥接模式，是在系统开始的时候就要考虑使用。

  虽然功能上不一样，这两个模式还是可以组合使用的，比如：已有实现部分的接口，但是有些不太适应现在新的功能对接口的需要，完全抛弃吧，有些功能还用得上，该怎么办呢？那就使用适配器来进行适配，使得旧的接口能够适应新的功能的需要。

# 访问者模式的相关模式

- 访问者模式和组合模式

  这两个模式可以组合使用。

  如同前面示例的那样，通过访问者模式给组合对象预留下扩展功能的接口，使得给组合模式的对象结构添加功能非常容易。

- 访问者模式和装饰模式

  这两个模式从表面看功能有些相似，都能够实现在不修改原对象结构的情况下修改原对象的功能。但是装饰模式更多的是实现对已有功能加强、或者修改、或者完全全新实现；而访问者模式更多的是实现给对象结构添加新的功能。

- 访问者模式和解释器模式

  这两个模式可以组合使用。

  解释器模式在构建抽象语法树的时候，是使用组合模式来构建的，也就是说解释器模式解释并执行的抽象语法树是一个组合对象结构，这个组合对象结构是很少变动的，但是可能经常需要为解释器增加新的功能，实现对同一对象结构的不同解释和执行的功能，这正好是访问者模式的优势所在，因此这在使用解释器模式的时候通常会组合访问者模式来使用。

# 临别赠言

- 不是结束而是新的开始

首先恭喜你，学到这里，说明你已经基本掌握了本课程所讲述的设计模式的内容，应该可以达到中级水平了。

但是，这并不是说你就不用再学习设计模式了，恰恰相反，要想在设计上更进一步的话，困难才刚刚开始。从中级的水平向上发展，更多的是需要思考和领悟，其难度比从入门到中级要大得多。

因此对你而言，学到这里并不是学习的结束，而是新的开始。

- 你该怎么做

1：多看：

多搜寻一些应用设计模式的实际项目、工程或是框架，参考别人的成功应用，再结合自己的经验来思考和使用。当然项目不应该太大，太大了很难完全看懂；也不能太小，太小了，没有太大实用价值，尤其是无法参考多个模式综合应用的情况，帮助就不大了。

# 临别赠言

**2：多练：**

多寻找机会，把这些设计模式在实际应用中使用，只有亲自动手去试验和使用，才能真正掌握和领会设计模式的精髓。

**3：多总结：**

认真分析每次对设计模式的使用是否得当，有什么经验和教训，是否有变形使用的情况，在不断总结中进步。

**4：反复参阅《研磨设计模式》一书：**

理论联系实际，通过实际应用反过来加深对理论的理解，以达到融会贯通这些设计模式的知识。因此，你需要反复参阅《研磨设计模式》一书，看看书上的知识，然后实践，然后再回头看书上的知识，你会有不一样的体会和领悟。

**5：多思考：**

多从设计上去思考这些设计模式，考虑它的设计意图、设计思想、解决问题的方式、实现的原理、模式的本质、以及如何变形使用等等。

只要你坚持去按照上面说的做，假以时日，必有所成。**预祝大家成功！**