



LANZHOU UNIVERSITY OF TECHNOLOGY

结课作业

学 号 1817700346

学生姓名 张 容

专业班级 软件三班

学 院 软件学院

[1.设计模式（代码范围）： 3](#_Toc42180679)

[1.1适配器模式 3](#_Toc42180680)

[1.2桥接模式 3](#_Toc42180681)

[1.3建造者模式 3](#_Toc42180682)

[1.4命令模式 3](#_Toc42180683)

[1.5组合模式 4](#_Toc42180684)

[1.6装饰模式 4](#_Toc42180685)

[1.7解释器模式 4](#_Toc42180686)

[1.8外观者模式 5](#_Toc42180687)

[1.9工厂模式 5](#_Toc42180688)

[1.10享元模式 5](#_Toc42180689)

[1.11责任链模式 5](#_Toc42180690)

[1.12中介者模式 6](#_Toc42180691)

[1.13备忘录模式 6](#_Toc42180692)

[1.14观察者模式 6](#_Toc42180693)

[1.15原型模式 6](#_Toc42180694)

[1.16代理模式 6](#_Toc42180695)

[1.17抽象工厂模式 7](#_Toc42180696)

[1.18单例模式 7](#_Toc42180697)

[1.19状态模式 7](#_Toc42180698)

[1.20策略模式 7](#_Toc42180699)

[1.21模板模式 7](#_Toc42180700)

[1.22访问者模式 8](#_Toc42180701)

[2.设计模式（应用的优缺点）： 9](#_Toc42180702)

[2.1单例模式 9](#_Toc42180703)

[2.2工厂方法模式 9](#_Toc42180704)

[2.3抽象工厂模式 9](#_Toc42180705)

[2.4模板方法模式 10](#_Toc42180706)

[2.5建造者模式 10](#_Toc42180707)

[2.6代理模式 10](#_Toc42180708)

[2.7原型模式（通过实现Cloneable接口） 10](#_Toc42180709)

[2.8中介者模式 11](#_Toc42180710)

[2.9命令模式 11](#_Toc42180711)

[2.10责任链模式 11](#_Toc42180712)

[2.11装饰器模式 12](#_Toc42180713)

[2.12策略模式 12](#_Toc42180714)

[2.13适配器模式 12](#_Toc42180715)

[2.14迭代器模式 12](#_Toc42180716)

[2.15组合模式 13](#_Toc42180717)

[2.16观察者模式 13](#_Toc42180718)

[2.17门面模式 13](#_Toc42180719)

[2.18备忘录模式 14](#_Toc42180720)

[2.19访问者模式 14](#_Toc42180721)

[2.20状态模式 14](#_Toc42180722)

[2.21解释器模式（现在使用较少） 15](#_Toc42180723)

[2.22享元模式 15](#_Toc42180724)

[2.23桥梁模式 15](#_Toc42180725)

# 1.设计模式：

## 1.1适配器模式

/\*\*

\* 适配器模式

\* 在计算机编程中，适配器模式（有时候也称包装样式或者包装）将一个类的接口适配成用户所期待的。

\* 一个适配允许通常因为接口不兼容而不能在一起工作的类工作在一起，做法是将类自己的接口包裹在一个已存在的类中。

\*

\*/

## 1.2桥接模式

 /\*\*

 \*  桥接模式

 \*  在软件系统中，某些类型由于自身的逻辑，它具有两个或多个维度的变化，那么如何应对这种“多维度的变化”？这就要使用桥接模式

 \*

 \*/

## 1.3建造者模式

 /\*\*

 \* 建造者模式 将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。

 \* 1 builder：为创建一个产品对象的各个部件指定抽象接口。

 \* 2 ConcreteBuilder：实现Builder的接口以构造和装配该产品的各个部件，定义并明确它所创建的表示，并 提供一个检索产品的接口。 3

 \*    Director：构造一个使用Builder接口的对象。 4

 \* Product：表示被构造的复杂对象。ConcreteBuilder创建该产品的内部表示并定义它的装配过程

 \* 包含定义组成部件的类，包括将这些部件装配成最终产品的接口。

 \*/

## 1.4命令模式

 /\*

 \*

 \* 命令模式 “行为请求者”与“行为实现者”通常呈现一种“紧耦合”。但在某些场合，比如要对行为进行“记录、撤销/重做、事务”等处理，

 \* 这种无法抵御变化的紧耦合是不合适的。

 \* 在这种情况下，如何将“行为请求者”与“行为实现者”解耦？将一组行为抽象为对象，实现二者之间的松耦合。这就是命令模式（Command Pattern）

 \*

 \* 1.降低对象之间的耦合度。 2.新的命令可以很容易地加入到系统中。 3.可以比较容易地设计一个组合命令。 4.调用同一方法实现不同的功能

 \*

 \*/

## 1.5组合模式

 /\*

 \*

 \* 组合模式 组合模式，将对象组合成树形结构以表示“部分-整体”的层次结构，组合模式使得用户对单个对象和组合对象的使用具有一致性。

 \* 有时候又叫做部分-整体模式，

 \* 它使我们树型结构的问题中，模糊了简单元素和复杂元素的概念，客户程序可以像处理简单元素一样来处理复杂元素,从而使得客户程序与复杂元素的内部结构解耦。

 \* 组合模式让你可以优化处理递归或分级数据结构

 \* 。有许多关于分级数据结构的例子，使得组合模式非常有用武之地。关于分级数据结构的一个普遍性的例子是你每次使用电脑时所遇到的

 \* :文件系统。文件系统由目录和文件组成

 \* 。每个目录都可以装内容。目录的内容可以是文件，也可以是目录。按照这种方式，计算机的文件系统就是以递归结构来组织的。如果你想要描述这样的数据结构

 \* ，那么你可以使用组合模式Composite。

 \*/

## 1.6装饰模式

 /\*

 \*

 \*装饰模式

 \*装饰模式是在不必改变原类文件和使用继承的情况下，动态地扩展一个对象的功能。它是通过创建一个包装对象，也就是装饰来包裹真实的对象。

 \*/

## 1.7解释器模式

 /\*\*

 \* 解释器模式 意图：给定一个语言，定义它的文法表示，并定义一个解释器，这个解释器使用该标识来解释语言中的句子。

 \* 主要解决：对于一些固定文法构建一个解释句子的解释器。

 \* 何时使用：如果一种特定类型的问题发生的频率足够高，那么可能就值得将该问题的各个实例表述为一个简单语言中的句子

 \* 。这样就可以构建一个解释器，该解释器通过解释这些句子来解决该问题。 如何解决：构件语法树，定义终结符与非终结符。

 \*/

## 1.8外观者模式

 /\*\*

 \* 外观模式

 \* 外观模式（Facade），为子系统中的一组接口提供一个一致的界面，定义一个高层接口，这个接口使得这一子系统更加容易使用。

 \*

 \*/

## 1.9工厂模式

 /\*\*

 \* 工厂模式 工厂模式是我们最常用的实例化对象模式了，是用工厂方法代替new操作的一种模式。

 \* 著名的Jive论坛,就大量使用了工厂模式，工厂模式在Java程序系统可以说是随处可见。因为工厂模式就相当于创建实例对象的new，我们经常要根据类Class生成实例对象，如A

 \* a=new A() 工厂模式也是用来创建实例对象的，所以以后new时就要多个心眼，是否可以考虑使用工厂模式，虽然这样做，可能多做一些工作，

 \* 但会给你系统带来更大的可扩展性和尽量少的修改量。

 \*/

## 1.10享元模式

 /\*\*

 \* 享元模式

 \*

 \* 它使用共享物件，用来尽可能减少内存使用量以及分享资讯给尽可能多的相似物件；它适合用于只是因重复而导致使用无法令人接受的大量内存的大量物件。

 \* 通常物件中的部分状态是可以分享。常见做法是把它们放在外部数据结构，当需要使用时再将它们传递给享元。

 \* 如果一个应用程序使用了大量的对象，而这些对象造成了很大的存储开销的时候就可以考虑是否可以使用享元模式。

 \* 例如,如果发现某个对象的生成了大量细粒度的实例，并且这些实例除了几个参数外基本是相同的

 \* ，如果把那些共享参数移到类外面，在方法调用时将他们传递进来，就可以通过共享大幅度单个实例的数目。

 \*/

## 1.11责任链模式

 /\*\*

 \* 责任链模式

 \*

 \* 责任链模式是一种设计模式。在责任链模式里，很多对象由每一个对象对其下家的引用而连接起来形成一条链。请求在这个链上传递，直到链上的某一个对象决定处理此请求。

 \* 发出这个请求的客户端并不知道链上的哪一个对象最终处理这个请求，这使得系统可以在不影响客户端的情况下动态地重新组织和分配责任。

 \*

 \*/

## 1.12中介者模式

 /\*\*

 \* 中介者模式 Mediator模式也叫中介者模式，是由GoF提出的23种软件设计模式的一种。Mediator模式是行为模式之一，在Mediator模式中，

 \* 类之间的交互行为被统一放在Mediator的对象中，对象通过Mediator对象同其他对象交互，Mediator对象起着控制器的作用。

 \*

 \*/

## 1.13备忘录模式

 /\*\*

 \* 备忘录模式

 \* 在不破坏封闭的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态。这样以后就可将该对象恢复到原先保存的状态。

 \*/

## 1.14观察者模式

 /\*\*

 \* 观察者模式

 \* 一个目标物件管理所有相依于它的观察者物件，并且在它本身的状态改变时主动发出通知。这通常透过呼叫各观察者所提供的方法来实现。

 \* 此种模式通常被用来实现事件处理系统。

 \*/

## 1.15原型模式

 /\*\*

 \* 原型模式

 \* 用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象。

 \*/

## 1.16代理模式

 /\*\*

 \* 代理模式

 \* 真实的角色就是实现实际的业务逻辑，不用关心其他非本职责的事务，通过后期的代理完成一件完成事务，附带的结果就是编程简洁清晰。

 \* (2).代理对象可以在客户端和目标对象之间起到中介的作用，这样起到了中介的作用和保护了目标对象的作用。 (3).高扩展性

 \*/

## 1.17抽象工厂模式

 /\*\*

 \* 抽象工厂模式 抽象工厂模式是所有形态的工厂模式中最为抽象和最具一般性的一种形态。抽象工厂模式是指当有多个抽象角色时，使用的一种工厂模式。

 \* 抽象工厂模式可以向客户端提供一个接口

 \* ，使客户端在不必指定产品的具体的情况下，创建多个产品族中的产品对象。根据里氏替换原则，任何接受父类型的地方，都应当能够接受子类型

 \* 。因此，实际上系统所需要的，

 \* 仅仅是类型与这些抽象产品角色相同的一些实例，而不是这些抽象产品的实例。换言之，也就是这些抽象产品的具体子类的实例。工厂类负责创建抽象产品的具体子类的实例。

 \*/

## 1.18单例模式

 /\*\*

 \* 单例模式

 \*单例模式是一种常用的软件设计模式。在它的核心结构中只包含一个被称为单例的特殊类。通过单例模式可以保证系统中一个类只有一个实例

 \*/

## 1.19状态模式

 /\*\*

 \* 状态模式

 \* 状态模式主要解决的是当控制一个对象状态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中，可以把复杂的判断逻辑简化。

 \*/

## 1.20策略模式

 /\*\*

 \*策略模式

 \*策略模式定义了一系列的算法，并将每一个算法封装起来，而且使它们还可以相互替换。策略模式让算法独立于使用它的客户而独立变化。

 \*/

## 1.21模板模式

 /\*\*

 \* 模板模式

 \*  模板方法模式（Template Method

 \* Pattern），定义一个操作中的算法骨架，而将一些实现步骤延迟到子类当中。模板方法使得子类可以在不改变算法结构的情况下，重新定义算法中的某些步骤。

 \* 模板方法模式是比较简单的一种设计模式

 \* ，但是它却是代码复用的一项基本的技术，在类库中尤其重要，它遵循“抽象类应当拥有尽可能多的行为，应当拥有尽可能少的数据”的重构原则

 \* 。作为模板的方法要定义在父类中，在方法的定义中使用到抽象方法，而只看父类的抽象方法是根本不知道怎样处理的，实际做具体处理的是子类，在子类中实现具体功能，

 \* 因此不同的子类执行将会得出不同的实现结果，但是处理流程还是按照父类定制的方式。这就是模板方法的要义所在，制定算法骨架，让子类具体实现。

 \*/

## 1.22访问者模式

 /\*\*

 \* 访问者模式

 \*

 \* 1.Visitor 抽象访问者角色，为该对象结构中具体元素角色声明一个访问操作接口。该操作接口的名字和参数标识了发送访问请求给具体访问者的具体元素角色，

 \* 这样访问者就可以通过该元素角色的特定接口直接访问它。 2.ConcreteVisitor.具体访问者角色，实现Visitor声明的接口。

 \* 3.Element 定义一个接受访问操作(accept())，它以一个访问者(Visitor)作为参数。

 \* 4.ConcreteElement具体元素，实现了抽象元素(Element)所定义的接受操作接口。

 \* 5.ObjectStructure结构对象角色，这是使用访问者模式必备的角色。它具备以下特性

 \* 能枚举它的元素；可以提供一个高层接口以允许访问者访问它的元素；如有需要，可以设计成一个复合对象或者一个聚集（如一个列表或无序集合）。

 \*

 \* 1、 一个对象结构包含很多类对象，它们有不同的接口，而你想对这些对象实施一些依赖于其具体类的操作。

 \* 2、需要对一个对象结构中的对象进行很多不同的并且不相关的操作，而你想避免让这些操作“污染”这些对象的类。Visitor模式使得你可以将相关的操作集中起来定义在一个类中。

 \* 3、 当该对象结构被很多应用共享时，用Visitor模式让每个应用仅包含需要用到的操作。

 \* 4、定义对象结构的类很少改变，但经常需要在此结构上定义新的操作

 \* 。改变对象结构类需要重定义对所有访问者的接口，这可能需要很大的代价。如果对象结构类经常改变，那么可能还是在这些类中定义这些操作较好。

 \*/

# 2.设计模式（应用的优缺点）：

## 2.1单例模式

优点：

只有一个实例，减少了内存开支；

可以避免对系统资源的多重占用；

可以在系统中设置全局的访问点，优化和共享资源访问；

缺点：

没有接口，扩展困难；

对测试开发不利；

应用场景：

要求生成唯一序列号的场景；

需要一个共享访问点；

创建一个对象需要消耗过多的资源时

需要定义大量的静态常量和静态方法时（也可直接声明为static的方式）；

## 2.2工厂方法模式

优点：

良好的封装性，代码结构清晰；

扩展非常好；

屏蔽产品类；

应用场景：

是new一个对象的替代品；

需要灵活的，可扩展的框架时；

使用在测试驱动开发的框架下；

## 2.3抽象工厂模式

优点：

封装性；

产品族内部的约束为非公开状态；

缺点：

产品族扩展困难；

## 2.4模板方法模式

优点：

封装不变部分，扩展可变部分，把不变的算法封装到父类实现，可变的部分则通过继承来扩展；

提取公共部分代码，便于维护；

行为由父类控制，子类实现;

缺点：

子类对父类产生影响，子类执行的结果影响了父类的结果;

应用场景：

多个子类有公有的方法，且逻辑相同时；

重要，复杂的算法，可以把核心算法设计为模板方法；

重构时，把相同的代码抽取到父类，然后通过钩子函数结束其行为；

## 2.5建造者模式

优点：

封装性，使得客户端不必知道产品内部的组成细节，我们不用关心每一个具体的模型内部是如何实现的。

建造者独立，容易扩展

便于控制细节风险，由于具体的建造者是独立的，因此可以对建造过程逐步细化，而不对其他的模块产生任何影响;

建造者模式的应用场景：

相同的方法，不同的执行顺序，会产生不同的结果时；

多个部件或零件，都可以装配到一个对象中，但产生的运行结果又不相同时，如Android中的AlertDialog的构造;

产品类非常复杂，或产品类的的调用顺序不同产生不同的效果；

## 2.6代理模式

优点：

职责清晰，其实的角色就是实现实际的业务的逻辑，不用关心其他非本职责的事务；

高扩展性，具体主题角色随时都会发生变化，但只要它实现了接口，我们的代理类就可以在完全不做任何修改的情况下使用；

## 2.7原型模式（通过实现Cloneable接口）

优点：

性能优良，原型模式是在内存二进制流的拷贝，要比直接new一个对象性能好，特别是要在循环体内产生大量对象时，

避免构造函数的约束，直接是在内存中拷贝的，构造函数是不会执行的。

应用场景：

类初始化需要消化非常多的资源时

性能和安全要求的场景，通过 new产生一个对象需要非常繁琐的数据准备和访问权限时；

一个对象多个修改者的场景，一个对象需要提供给多个对象访问，而且各个调用者都可以修改其值时；

注意地方：浅拷贝与深拷贝

Java的Object类提供的clone方法只是拷贝本对象，其对象内部的数组、引用对象等都不拷贝，其他的原始类型如int,char等都会被拷贝，拷贝后的对象与原生对象共享内部元素的地址（浅拷贝），如果拷贝后的对象修改了原生对象的数组，则原生对象也会看到修改。如果需要进行深拷贝，则需要在复写的clone方法里对私有的类变量（内部数组，引用对象）进行独立的拷贝。并且使用final关键字修饰的变量不能被拷贝；

## 2.8中介者模式

优点：

减少了类间的依赖，把原有的一对多的依赖变成了一对一的依赖；

缺点：

中介者会膨胀得很大，而且逻辑复杂；原本N个对象的依赖关系转换为中介者与对象的依赖关系；

## 2.9命令模式

优点：

类间解耦，调用者与接收者之间没有任何依赖关系，调用者实现功能时不需要了解到底是哪个接收者执行，只需调用Command抽象类的execute方法就可以了；

可扩展性，Command的子类可以非常容易扩展，并且调用者和高层模块不产生严重的代码耦合；

缺点：

Command类膨胀厉害，如果有N个命令，则Command类的子类就为N个；

应用场景：如Android中各种事件的处理；

## 2.10责任链模式

优点：

请求与处理分开，请求者可以不用知道是谁处理的，处理者可以不用知道请求的全貌；

缺点：

性能问题，每个请求都是从链头遍历到链尾的，当这个责任链比较长时，遍历开销会比较大；

应用场景:

如Android事件的传递机制；

## 2.11装饰器模式

优点：

装饰类和被装饰类可以独立发展，而不会互相耦合；

装饰模式是继承关系的一个替代方案，不管装饰多少层，最终返回的也还是那个对象；

装饰模式可以动态地扩展一个实现类的功能；

缺点：

多层的装饰比较复杂，当使用多层装饰出现问题时，排查问题的工作量比较大

应用场景：

需要扩展一个类的功能，或给一个类增加附加功能；

需要为一批兄弟类进行改装或加装功能；

## 2.12策略模式

优点：

算法可以自由切换，只要实现抽象策略，它就成为策略家庭的一个成员；

避免使用多重条件判断，

扩展性良好，在现有的系统中增加一个策略太容易，只要实现接口就可以了;

缺点：

策略类数量多，每一个策略都是一个类，复用的可能性很小；

所有的策略类都需要对外暴露，上层模块必须知道有哪些策略，然后决定使用哪一个策略；

应用场景：

多个类只有在算法或行为上稍有不同的场景；

算法需要自由切换的场景；

需要屏蔽算法规则的场景；

## 2.13适配器模式

优点：

让两个没有任何联系的类在一起运行；

增加了类的透明性；

提高了类的复用度；

灵活性好，当不需要适配器时，只要删掉这个适配器就可以了，

应用场景：

修改一个已经投产的接口时，

Android中各种Adapter，

## 2.14迭代器模式

迭代器模式是为解决遍历容器中的元素而诞生的，没有人会单独写一个迭代器，使用Java提供的Itreator就可以满足要求了；

## 2.15组合模式

优点：

高层模块调用简单，高层模块不需要关心自己处理的是单个对象还是整个组合结构，

节点可以自由增加；

缺点：

调用时会直接使用实现类，不符合面向接口编程思想；

应用场景：

维护和展示部分-整体关系的场景，如树型菜单，文件和文件夹的管理；

只要是树型结构，就要考虑使用组合模式；

## 2.16观察者模式

优点：

观察者与被观察者之间是抽象耦合，不管是增加观察者还是被观察者都非常容易扩展；

建立一套触发机制；

缺点：

一个被观察者，多个观察者，开发与调度会比较复杂，在Java中消息的通知默认是顺序执行，其中一个观察者卡壳，会影响整体的执行效率，一般要考虑采用异步的方式；

应用场景：

关联行为场景，如Android中数据变化会引起UI的变化；

事件多级触发场景；

跨系统的消息交换场景；

## 2.17门面模式

优点：

减少系统的相互依赖，所有的依赖都是与门面对象的依赖，与子系统无关。

提高了灵活性；

提高了安全性，想让你访问子系统的哪些业务就开通哪些逻辑；

缺点：

不符合开闭原则，当出现bug后，只能通过修改门面角色的代码来修复；

应用场景：

为一个复杂的模块或子系统提供一个供外界访问的接口，如Android的Context类只是一个抽象类，所有的功能都是在ContextImpl类实现的，我们不会察觉到ContextImpl的存在，只需要调用Context就可以了；

子系统相对独立，外界对子系统的访问只要黑箱操作即可；

预防低水平开发人员带来的风险，被限定在指定的子系统开发；

## 2.18备忘录模式

应用场景：

需要保存和恢复数据的相关状态场景；

提供一个可回滚的操作场景；

需要监控的副本场景中；

数据库连接的事务管理就是用的备忘录模式；

注意事项：

备忘录的生命期，要主动管理它的生命周期，建立就要使用，不使用就删除；

备忘录的性能，不要在频繁建立备份的场景中使用备忘录模式；（对象的创建是需要消耗资源的）

## 2.19访问者模式

优点：

符合单一职责原则，具体元素角色负责数据的加载，而访问者类则负责数据的呈现；

优秀的扩展性，

灵活性非常高；

缺点：

具体元素对访问者公布细节，访问者要访问一个类就必须要求这个类公布一些方法和数据；

具体元素变更比较困难；具体元素角色的增加、删除、修改都是比较困难；

违背了依赖倒置原则，访问者依赖的是具体的元素，而不是抽象的元素；

应用场景 ：

一个对象结构包含很多类对象，它们有不同的接口，而你想对这些对象实施一些依赖于其具体类的操作；

需要对一个对象结构中的对象进行很多不同并且不相关的操作，而你想避免让这些操作”污染“这些对象的类；

业务规则要求遍历多个不同的对象；

## 2.20状态模式

优点：

结构清晰，避免了过多的switch...case或if...else语句的使用；

遵循设计原则，每个状态就是一个子类；

封装性非常好，将状态变换放置到类的内部来实现；

缺点：

子类会太多，也就是类膨胀，有多少个状态，就会有多少个子类；

应用场景：

行为随状态改变而改变的场景，如权限设计；

条件、分支判断语句的替代者，通过扩展子类实现条件的判断处理；

状态的个数最好不要超过5个；

## 2.21解释器模式（现在使用较少）

优点：

扩展性好，

缺点：

解释器模式会引起类膨胀；

采用了递归调用方法；

## 2.22享元模式

优点：

大大减少应用程序创建的对象，降低程序内存的占用;

缺点：

提高了系统复杂性，需要分离出内部和外部状态；

应用场景：

系统中存在大量的相似对象；

需要缓冲池的场景；

细粒度的对象都具有较接近的外部状态；且内部状态与环境无关

## 2.23桥梁模式

优点：

抽象与实现分离；

优秀的扩充能力；

实现细节对客户透明；

应用场景：

不希望或不适用继承的场景；

接口或抽象类不稳定的情况；

重要性要求较高的场景；