**创建型模式**，共五种：**工厂方法模式、抽象工厂模式、单例模式、建造者模式**、原型模式。

**结构型模式**，共七种：**适配器模式、装饰器模式、代理模式**、**外观模式**、桥接模式、**组合模式**、享元模式。

**行为型模式**，共十一种：**策略模式**、模板方法模式、**观察者模式**、迭代子模式、**责任链模式**、**命令模式**、备忘录模式、状态模式、访问者模式、中介者模式、**解释器模式**。

**其他如：Ioc**

**1、开闭原则（Open Close Principle）**

开闭原则就是说**对扩展开放，对修改关闭**。在程序需要进行拓展的时候，不能去修改原有的代码（工厂方法与简单工厂），实现一个热插拔的效果。所以一句话概括就是：为了使程序的扩展性好，易于维护和升级。想要达到这样的效果，我们需要使用接口和抽象类，后面的具体设计中我们会提到这点。

**2、里氏代换原则（Liskov Substitution Principle）**

里氏代换原则(Liskov Substitution Principle LSP)面向对象设计的基本原则之一。 里氏代换原则中说，**任何基类可以出现的地方，子类一定可以出现**。 LSP是继承复用的基石，只有当衍生类可以替换掉基类，软件单位的功能不受到影响时，基类才能真正被复用，而衍生类也能够在基类的基础上增加新的行为。里氏代换原则是对“开-闭”原则的补充。实现“开-闭”原则的关键步骤就是抽象化。而基类与子类的继承关系就是抽象化的具体实现，所以里氏代换原则是对实现抽象化的具体步骤的规范。

**3、依赖倒转原则（Dependence Inversion Principle）**

这个是开闭原则的基础，具体内容：真对接口编程，**依赖于抽象而不依赖于具体**。

**4、接口隔离原则（Interface Segregation Principle）**

这个原则的意思是：使用多个隔离的接口，比使用单个接口要好。**一个类对另一个类的依赖应该建立在最小的接口上**，**不应该将多个没有关系的接口耦合在一起。**还是一个降低类之间的耦合度的意思，从这儿我们看出，其实设计模式就是一个软件的设计思想，从大型软件架构出发，为了升级和维护方便。所以上文中多次出现：降低依赖，降低耦合。

**5、**[**单一职责原则**](https://baike.baidu.com/item/%E5%8D%95%E4%B8%80%E8%81%8C%E8%B4%A3%E5%8E%9F%E5%88%99) **（Single Responsiblity Principle SRP）**

**6、迪米特法则（最少知道原则）（Demeter Principle）**

为什么叫最少知道原则，就是说：**一个实体应当尽量少的与其他实体之间发生相互作用**，使得系统功能模块相对独立。

**7、合成复用原则（Composite Reuse Principle）**

原则是尽量使用合成/聚合的方式，而不是使用继承。

前五个：solid

**（工厂模式）：(一个类提供多种不同对象)**

**为了屏蔽屏蔽一个类的实例化（实例化参数、步骤类名等）**

**工厂方法模式：A**

特点：四个主要角色，一个工厂接口，一个产品接口，多个工厂类，多个产品类。

工厂接口只是接口，由工厂类决定实例化什么对象。

一个工厂类对应一个产品

<https://en.wikipedia.org/wiki/Factory_method_pattern> 看uml

**抽象工厂模式：A**

特点：四个主要角色，一个工厂接口，**多个**产品接口，多个工厂类，多个产品类

区别于工厂方法：一个工厂可能对应多个产品，比如华为工厂生产华为pad和华为手机，

比工厂方法多了一个产品族的概念。一个工厂可以生产多个产品。

两种工厂方法模式不同于简单工厂方法模式，在新增产品时只需要新增，不需要修改。

<https://en.wikipedia.org/wiki/Abstract_factory_pattern> 看uml

**单例模式： A**

**特点：一个类只有一个实例 场景：实例化工具类**

**建造者模式 A**

工厂模式都是为了生产单个产品，建造者是为了生成一个复杂的商品，屏蔽底层的零件构造。

<http://www.jianshu.com/p/e16d1ac9a335>

**适配器模式 A**

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（**改变对象接口使之适应系统**）。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

有不符合标准的类A和标准接口的B。类A不存在一个方法，或存在方法不符合要求。通过新增一个adpater类，通过adpater extends A implments B实现适配。

**装饰器模式**

<http://www.runoob.com/design-pattern/decorator-pattern.html>

为了增加功能

**代理模式 A**

某个类A里有几个需要代理的对象B、C，A.xxx(),其实是调用B、C的XXX方法。

解决：在直接访问对象时带来的问题

**组合模式**

以树形模式组合一组对象

<http://www.runoob.com/design-pattern/composite-pattern.html>

**外观模式**

将系统各个类的方法在一个同一的类中提供接口，屏蔽系统复杂性

<http://www.runoob.com/design-pattern/facade-pattern.html>

**策略模式**

一个策略对象根据策略调用不同的类的方法。

<http://www.runoob.com/design-pattern/strategy-pattern.html>

**观察者模式**

定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。

一个类中有一个对象为观察者列表，当对象修改，通知所有观察者做响应操作。

<http://www.runoob.com/design-pattern/observer-pattern.html>

**责任链模式**

如果一个对象不能处理该请求，那么它会把相同的请求传给下一个接收者，依此类推。

**命令模式A**

**请求以命令的形式包裹在对象中**，并传给调用对象。调用对象寻找可以处理该命令的合适的对象，并把该命令传给相应的对象，该对象执行命令。

Odp，类似于controller实现。

<http://blog.csdn.net/jialinqiang/article/details/8752111>

客户（非主体客户）、命令继承自命令接口（订单），请求者（服务员），接收者（厨师）

一个请求来，生成一个命令调用类，调用类调用相关接口处理请求。

public interface Command

{

public void execute();

}

//命令

public class ConcreteCommand implements Command

{

// 持有接收者的引用

private Receiver receiver;

public ConcreteCommand(Receiver receiver)

{

this.receiver = receiver;

}

@Override

public void execute()//将命令的执行委托给接收者

{

receiver.doAction();

}

}

//接收者 真正执行逻辑的

public class Receiver

{

public void doAction()

{

System.out.println("doAction");

}

}

//调度（调用者，服务员）者，invoker内部可以实现队列、日志等

public class Invoker

{

private Command command;

public Invoker(Command command)

{

this.command = command;

}

public void doInvokerAction()

{

command.execute();

}

}

public class Client

{

public static void main(String[] args)

{

Receiver receiver = new Receiver();

Command command = new ConcreteCommand(receiver);

Invoker invoker = new Invoker(command);

invoker.doInvokerAction();

}

}

**解释器模式**。

语法解释器构架，Gof是这样定义的：给定一个语言，定义它的文法的一种表示，并定义一个解释器，这个解释器使用该表示来解释语言中的句子。

实例应用：正则表达式

<http://www.runoob.com/design-pattern/interpreter-pattern.html> pass

**单例模式**

1、单例类只能有一个实例。  
2、单例类必须自己创建自己的唯一实例。  
3、单例类必须给所有其他对象提供这一实例。

1. **public** **static** Singleton getInstance() {
2. private static Singleton instance = new Singleton();
3. **if** (singleton == **null**) { //dlc问题 http://blog.csdn.net/fw0124/article/details/42737171
4. **synchronized** (Singleton.**class**) {
5. **if** (singleton == **null**) {
6. singleton = **new** Singleton();
7. }
8. }
9. }
10. **return** singleton;
11. }
12. **public** **class** Singleton {
13. **private** **static** **class** LazyHolder {
14. **private** **static** **final** Singleton INSTANCE = **new** Singleton();
15. }
16. **private** Singleton (){}
17. **public** **static** **final** Singleton getInstance() {
18. **return** LazyHolder.INSTANCE;
19. }
20. }

//线程安全，不需同步

工厂模式：

作用：

目的是为了解耦，还可以隐藏类名，也不需要准备构造方法的参数。

简单new一个对象，如果许多地方都有要修改时则都需要修改并编译。

几种实现方式

参见17.2

代理

Class simpleProxy{

Private Interface proxied;

Public SimpleProxy(Interface proxied){

This.proxied = proxied;

}

Public void doSomethid(){

Proxied.doSomethind();

}

}

动态代理

Interface **proxy** = (Interface)Proxy.**newProxyInstance**(

Interface.class.getClassLoader();//**一个类加载器**

New Class[]{Interface.class}//**代理实现的列表接口 ，这里只希望代理Interface**

New DynamicProxyHandler(real)) //InvocationHandler的一个实现);

Proxy.doSomething()

**InvocationHandler**主要实现 **invoke**(Object proxy,Method method,Object [] args)和它的构造器

//实现

构造方法(要代理的对象){

Proxied = 要代理的对象;

}

invoke(Object proxy,Method method,Object [] args){

method.invoke(proxied,args);

}

MVC

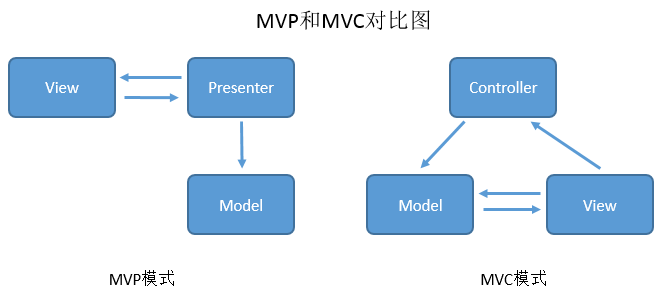
模型(model)－视图(view)－控制器(controller)的缩写

Model（模型）是应用程序中用于处理应用程序数据逻辑的部分。 通常模型对象负责在数据库中存取数据。

View（视图）是应用程序中处理数据显示的部分。

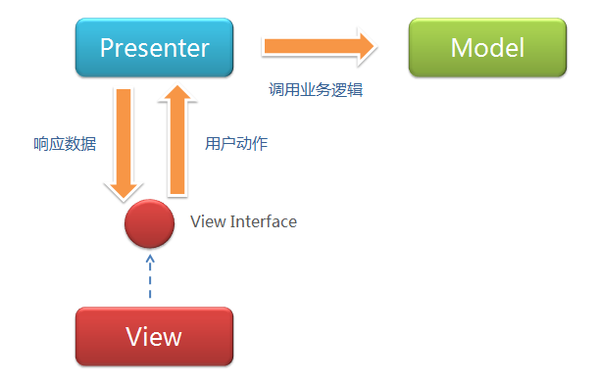
Controller（控制器）是应用程序中处理用户交互的部分。 通常控制器负责从视图读取数据，控制用户输入，并向模型发送数据。

MVC的一般流程是这样的：View（界面）触发事件--》Controller（业务）处理了业务，然后触发了数据更新--》不知道谁更新了Model的数据--》Model（带着数据）回到了View--》View更新数据（非实际应用中的mvc）



MVP:（实际开发中 mvc和mvp一致）

MVP是把MVC中的Controller换成了Presenter（呈现），目的就是为了完全切断View跟Model之间的联系，由Presenter充当桥梁，做到View-Model之间通信的完全隔离。



Models分为ps、ds和dao

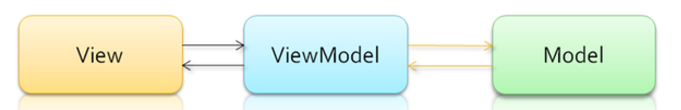
Controller层:Controller层负责具体的业务模块流程的控制

Dao层持久化

Service分为ds和ps，ps针对单一接口来调用ds层的具体业务逻辑实现方法。Ds层可由不同接口复用。

MVVM

Model-View-ViewModel



ViewModel大致上就是MVP的Presenter和MVC的Controller了，而View和ViewModel间没有了MVP的界面接口，而是直接交互，**用数据“绑定”的形式让数据更新的事件不需要开发人员手动去编写特殊用例，而是自动地双向同步**。

在MVVM中，数据和业务逻辑处于一个独立的View Model中，ViewModel只要关注数据和业务逻辑，不需要和UI或者控件打交道。由数据自动去驱动UI去自动更新UI，UI的改变又同时自动反馈到数据，数据成为主导因素，这样使得在业务逻辑处理只要关心数据，方便而且简单很多。