1.单例模式（Singleton Pattern）

定义：Ensure a class has only one instance, and provide a global point of access to it.（确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例。）

通用代码：（是线程安全的）

```
public class Singleton {
     private static final Singleton singleton = new Singleton();
//限制产生多个对象
     private Singleton(){
     }
     //通过该方法获得实例对象
     public static Singleton getSingleton(){
             return singleton;
     }
     //类中其他方法，尽量是 static
     public static void doSomething(){
     }
}
```

使用场景：

- 要求生成唯一序列号的环境；
- 在整个项目中需要一个共享访问点或共享数据，例如一个 Web 页面上的计数器，可以不用把每次刷新都记录到数据库中，使用单例模式保持计数器的值，并确保是线程安全的；
- 创建一个对象需要消耗的资源过多，如要访问 IO 和数据库等资源；
- 需要定义大量的静态常量和静态方法（如工具类）的环境，可以采用单例模式（当然，也可以直接声明为 static 的方式）。

线程不安全实例：

```
public class Singleton {
     private static Singleton singleton = null;
     //限制产生多个对象
     private Singleton(){
     }
     //通过该方法获得实例对象
     public static Singleton getSingleton(){
             if(singleton == null){
                    singleton = new Singleton();
```

```
            }
            return singleton;
    }
}
```

解决办法：
在 getSingleton 方法前加 synchronized 关键字，也可以在 getSingleton 方法内增加 synchronized 来实现。**最优的办法是如通用代码那样写。**

2.工厂模式

定义：Define an interface for creating an object,but let subclasses decide which class to instantiate.Factory Method lets a class defer instantiation to subclasses.（定义一个用于创建对象的接口，让子类决定实例化哪一个类。工厂方法使一个类的实例化延迟到其子类。）

Product 为抽象产品类负责定义产品的共性，实现对事物最抽象的定义；
Creator 为抽象创建类，也就是抽象工厂，具体如何创建产品类是由具体的实现工厂 ConcreteCreator 完成的。
具体工厂类代码：

```
public class ConcreteCreator extends Creator {
public <T extends Product> T createProduct(Class<T> c){
            Product product=null;
            try {
                  product =
(Product)Class.forName(c.getName()).newInstance();
            } catch (Exception e) {
                  //异常处理
            }
            return (T)product;
    }
}
```

简单工厂模式：
一个模块仅需要一个工厂类，没有必要把它产生出来，使用静态的方法

多个工厂类：
每个人种（具体的产品类）都对应了一个创建者，每个创建者独立负责创建对应的产品对象，非常符合单一职责原则

代替单例模式：
单例模式的核心要求就是在内存中只有一个对象，通过工厂方法模式也可以只在内存中生产一个对象

延迟初始化：
ProductFactory 负责产品类对象的创建工作，并且通过 prMap 变量产生一个缓存，对需要再次被重用的对象保留

使用场景：jdbc 连接数据库，硬件访问，降低对象的产生和销毁

3.抽象工厂模式（Abstract Factory Pattern）

定义：Provide an interface for creating families of related or dependent objects without specifying their concrete classes.（为创建一组相关或相互依赖的对象提供一个接口，而且无须指定它们的具体类。）
抽象工厂模式通用类图：

抽象工厂模式通用源码类图：

抽象工厂类代码：

```
public abstract class AbstractCreator {
     //创建 A 产品家族
     public abstract AbstractProductA createProductA();
     //创建 B 产品家族
     public abstract AbstractProductB createProductB();
}
```

使用场景：
一个对象族（或是一组没有任何关系的对象）都有相同的约束。
涉及不同操作系统的时候，都可以考虑使用抽象工厂模式

4.模板方法模式（Template Method Pattern）

定义：Define the skeleton of an algorithm in an operation,deferring some steps to subclasses.Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.（定义一个操作中的算法的框架，而将一些步骤延迟到子类中。使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。）

AbstractClass 叫做**抽象模板**，它的方法分为两类：

- 基本方法

基本方法也叫做基本操作，是由子类实现的方法，并且在模板方法被调用。

- 模板方法

可以有一个或几个，一般是一个具体方法，也就是一个框架，实现对基本方法的调度，完成固定的逻辑。

注意：　为了防止恶意的操作，一般模板方法都加上 final 关键字，不允许被覆写。

具体模板：ConcreteClass1 和 ConcreteClass2 属于具体模板，实现父类所定义的一个或多个抽象方法，也就是父类定义的基本方法在子类中得以实现

使用场景：

- 多个子类有公有的方法，并且逻辑基本相同时。
- 重要、复杂的算法，可以把核心算法设计为模板方法，周边的相关细节功能则由各个子类实现。
- 重构时，模板方法模式是一个经常使用的模式，把相同的代码抽取到父类中，然后通过钩子函数（见“模板方法模式的扩展”）约束其行为。

5.建造者模式（Builder Pattern）

定义：Separate the construction of a complex object from its representation so that the same construction process can create different representations.（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。）

- Product 产品类

通常是实现了模板方法模式，也就是有模板方法和基本方法，例子中的 BenzModel 和 BMWModel 就属于产品类。

- Builder 抽象建造者

规范产品的组建，一般是由子类实现。例子中的 CarBuilder 就属于抽象建造者。

- ConcreteBuilder 具体建造者

实现抽象类定义的所有方法，并且返回一个组建好的对象。例子中的 BenzBuilder 和 BMWBuilder 就属于具体建造者。

● Director 导演类
负责安排已有模块的顺序，然后告诉 Builder 开始建造
使用场景：
● 相同的方法，不同的执行顺序，产生不同的事件结果时，可以采用建造者模式。
● 多个部件或零件，都可以装配到一个对象中，但是产生的运行结果又不相同时，则可以使用该模式。
● 产品类非常复杂，或者产品类中的调用顺序不同产生了不同的效能，这个时候使用建造者模式非常合适。

建造者模式与工厂模式的不同：
建造者模式最主要的功能是基本方法的调用顺序安排，这些基本方法已经实现了，顺序不同产生的对象也不同；
工厂方法则重点是创建，创建零件是它的主要职责，组装顺序则不是它关心的。

6.代理模式（Proxy Pattern）

定义：Provide a surrogate or placeholder for another object to control access to it.（为其他对象提供一种代理以控制对这个对象的访问。）

● Subject 抽象主题角色
抽象主题类可以是抽象类也可以是接口，是一个最普通的业务类型定义，无特殊要求。
● RealSubject 具体主题角色
也叫做被委托角色、被代理角色。它才是冤大头，是业务逻辑的具体执行者。
● Proxy 代理主题角色
也叫做委托类、代理类。它负责对真实角色的应用，把所有抽象主题类定义的方法限制委托给真实主题角色实现，并且在真实主题角色处理完毕前后做预处理和善后处理工作。

普通代理和强制代理：
普通代理就是我们要知道代理的存在，也就是类似的 GamePlayerProxy 这个类的存在，然后才能访问；
强制代理则是调用者直接调用真实角色，而不用关心代理是否存在，其代理的产生是由真实角色决定的。
普通代理：

在该模式下，调用者只知代理而不用知道真实的角色是谁，屏蔽了真实角色的变更对高层模块的影响，真实的主题角色想怎么修改就怎么修改，对高层次的模块没有

任何的影响，只要你实现了接口所对应的方法，该模式非常适合对扩展性要求较高的场合。

强制代理：

强制代理的概念就是要从真实角色查找到代理角色，不允许直接访问真实角色。高层模块只要调用 getProxy 就可以访问真实角色的所有方法，它根本就不需要产生一个代理出来，代理的管理已经由真实角色自己完成。

动态代理：
根据被代理的接口生成所有的方法，也就是说***给定一个接口，动态代理会宣称"我已经实现该接口下的所有方法了"。***

两条独立发展的线路。动态代理实现代理的职责，业务逻辑 Subject 实现相关的逻辑功能，两者之间没有必然的相互耦合的关系。通知 Advice 从另一个切面切入，最终在高层模块也就是 Client 进行耦合，完成逻辑的封装任务。

动态代理调用过程示意图：

动态代理的意图：横切面编程，在不改变我们已有代码结构的情况下增强或控制对象的行为。
首要条件：被代理的类必须要实现一个接口。

7.原型模式（Prototype Pattern）

定义：Specify the kinds of objects to create using a prototypical instance,and create new objects by copying this prototype.（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。）

原型模式通用代码：

```
public class PrototypeClass  implements Cloneable{
    //覆写父类 Object 方法
    @Override
    public PrototypeClass clone(){
            PrototypeClass prototypeClass = null;
            try {
                    prototypeClass = (PrototypeClass)super.clone();
            } catch (CloneNotSupportedException e) {
                    //异常处理
```

```
            }
            return prototypeClass;
    }
}
```

原型模式实际上就是实现 Cloneable 接口，重写 clone（）方法。
使用原型模式的优点：
● 性能优良
原型模式是在内存二进制流的拷贝，要比直接 new 一个对象性能好很多，特别是要在一个循环体内产生大量的对象时，原型模式可以更好地体现其优点。
● 逃避构造函数的约束
这既是它的优点也是缺点，直接在内存中拷贝，构造函数是不会执行的（参见 13.4 节）。
使用场景：
● 资源优化场景
类初始化需要消化非常多的资源，这个资源包括数据、硬件资源等。
● 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或访问权限，则可以使用原型模式。
● 一个对象多个修改者的场景
一个对象需要提供给其他对象访问，而且各个调用者可能都需要修改其值时，可以考虑使用原型模式拷贝多个对象供调用者使用。

浅拷贝和深拷贝：
浅拷贝：Object 类提供的方法 clone 只是拷贝本对象，其对象***内部的数组、引用对象***等都不拷贝，还是指向原生对象的内部元素地址，这种拷贝就叫做***浅拷贝***，其他的原始类型比如 int、long、char、string（当做是原始类型）等都会被拷贝。
注意： 使用原型模式时，引用的成员变量必须满足两个条件才不会被拷贝：一是***类的成员变量***，而不是方法内变量；二是必须是一个***可变的引用对象***，而不是一个原始类型或不可变对象。
深拷贝：对私有的类变量进行独立的拷贝
如：thing.arrayList = (ArrayList<String>)this.arrayList.clone();

8.中介者模式

定义：Define an object that encapsulates how a set of objects interact.Mediator promotes loose coupling by keeping objects from referring to each other explicitly,and it lets you vary their interaction independently.（用一个中介对象封装一系列的对象交互，中介者使各对象不需要显示地相互作用，从而使其耦合松散，而且可以独立地改变它们之间的交互。）

● Mediator 抽象中介者角色

抽象中介者角色定义统一的接口，用于各同事角色之间的通信。
● Concrete Mediator 具体中介者角色
具体中介者角色通过协调各同事角色实现协作行为，因此它必须依赖于各个同事角色。
● Colleague 同事角色
每一个同事角色都知道中介者角色，而且与其他的同事角色通信的时候，一定要通过中介者角色协作。每个同事类的行为分为两种：一种是同事本身的行为，比如改变对象本身的状态，处理自己的行为等，这种行为叫做自发行为（Self-Method），与其他的同事类或中介者没有任何的依赖；第二种是必须依赖中介者才能完成的行为，叫做依赖方法（Dep-Method）。

通用抽象中介者代码：

```
public abstract class Mediator {
     //定义同事类
     protected ConcreteColleague1 c1;
     protected ConcreteColleague2 c2;
     //通过 getter/setter 方法把同事类注入进来
     public ConcreteColleague1 getC1() {
             return c1;
     }
     public void setC1(ConcreteColleague1 c1) {
             this.c1 = c1;
     }
     public ConcreteColleague2 getC2() {
             return c2;
}
     public void setC2(ConcreteColleague2 c2) {
             this.c2 = c2;
     }
     //中介者模式的业务逻辑
     public abstract void doSomething1();
     public abstract void doSomething2();
}
```

ps：使用同事类注入而不使用抽象注入的原因是因为抽象类中不具有每个同事类必须要完成的方法。即每个同事类中的方法各不相同。

问：为什么同事类要使用构造函数注入中介者，而中介者使用 getter/setter 方式注入同事类呢？
这是因为同事类必须有中介者，而中介者却可以只有部分同事类。

使用场景：

中介者模式适用于多个对象之间紧密耦合的情况，紧密耦合的标准是：在类图中出现了蜘蛛网状结构，即每个类都与其他的类有直接的联系。

9.命令模式

定义：Encapsulate a request as an object,thereby letting you parameterize clients with different requests,queue or log requests,and support undoable operations.（将一个请求封装成一个对象，从而让你使用不同的请求把客户端参数化，对请求排队或者记录请求日志，可以提供命令的撤销和恢复功能。）

● Receive 接收者角色
该角色就是干活的角色，命令传递到这里是应该被执行的，具体到我们上面的例子中就是 Group 的三个实现类（需求组，美工组，代码组）。
● Command 命令角色
需要执行的所有命令都在这里声明。
● Invoker 调用者角色
接收到命令，并执行命令。在例子中，我（项目经理）就是这个角色。
使用场景：
认为是命令的地方就可以采用命令模式，例如，在 GUI 开发中，一个按钮的点击是一个命令，可以采用命令模式；模拟 DOS 命令的时候，当然也要采用命令模式；触发－反馈机制的处理等。

10.责任链模式

定义：Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request.Chain the receiving objects and pass the request along the chain until an object handles it.（使多个对象都有机会处理请求，从而避免了请求的发送者和接受者之间的耦合关系。将这些对象连成一条链，并沿着这条链传递该请求，直到有对象处理它为止。）

抽象处理者的代码：

```
public abstract class Handler {
     private Handler nextHandler;
     //每个处理者都必须对请求做出处理
     public final Response handleMessage(Request request){
             Response response = null;
             //判断是否是自己的处理级别

if(this.getHandlerLevel().equals(request.getRequestLevel())){
                     response = this.echo(request);
```

```
            }else{  //不属于自己的处理级别
                    //判断是否有下一个处理者
                    if(this.nextHandler != null){
                            response =
this.nextHandler.handleMessage(request);
                    }else{
                            //没有适当的处理者，业务自行处理
                    }
            }
            return response;
    }
    //设置下一个处理者是谁
    public void setNext(Handler _handler){
            this.nextHandler = _handler;
    }
    //每个处理者都有一个处理级别
    protected abstract Level getHandlerLevel();
    //每个处理者都必须实现处理任务
    protected abstract Response echo(Request request);
}
```

抽象的处理者实现三个职责：
一是定义一个请求的处理方法 handleMessage，唯一对外开放的方法；
二是定义一个链的编排方法 setNext，设置下一个处理者；
三是定义了具体的请求者必须实现的两个方法：定义自己能够处理的级别 getHandlerLevel 和具体的处理任务 echo。
注意事项：
链中节点数量需要控制，避免出现超长链的情况，一般的做法是在 Handler 中设置一个最大节点数量，在 setNext 方法中判断是否已经是超过其阈值，超过则不允许该链建立，避免无意识地破坏系统性能。

11.装饰模式（Decorator Pattern）

定义：Attach additional responsibilities to an object dynamically keeping the same interface.Decorators provide a flexible alternative to subclassing for extending functionality.（动态地给一个对象添加一些额外的职责。就增加功能来说，装饰模式相比生成子类更为灵活。）

● Component 抽象构件

Component 是一个接口或者是抽象类，就是定义我们最核心的对象，也就是最原始的对象，如上面的成绩单。
注意：在装饰模式中，必然有一个最基本、最核心、最原始的接口或抽象类充当 Component 抽象构件。
● ConcreteComponent 具体构件
ConcreteComponent 是最核心、最原始、最基本的接口或抽象类的实现，你要装饰的就是它。
● Decorator 装饰角色
一般是一个抽象类，做什么用呢？实现接口或者抽象方法，它里面可不一定有抽象的方法呀，在它的属性里必然有一个 private 变量指向 Component 抽象构件。
● 具体装饰角色
ConcreteDecoratorA 和 ConcreteDecoratorB 是两个具体的装饰类，你要把你最核心的、最原始的、最基本的东西装饰成其他东西，上面的例子就是把一个比较平庸的成绩单装饰成家长认可的成绩单。
使用场景：
● 需要扩展一个类的功能，或给一个类增加附加功能。
● 需要动态地给一个对象增加功能，这些功能可以再动态地撤销。
● 需要为一批的兄弟类进行改装或加装功能，当然是首选装饰模式。

12.策略模式（Strategy Pattern）

定义：Define a family of algorithms,encapsulate each one,and make them interchangeable.（定义一组算法，将每个算法都封装起来，并且使它们之间可以互换。）

● Context 封装角色
它也叫做上下文角色，起承上启下封装作用，屏蔽高层模块对策略、算法的直接访问，封装可能存在的变化。
● Strategy 抽象策略角色
策略、算法家族的抽象，通常为接口，定义每个策略或算法必须具有的方法和属性。各位看官可能要问了，类图中的 AlgorithmInterface 是什么意思，嘿嘿，algorithm 是“运算法则”的意思，结合起来意思就明白了吧。
● ConcreteStrategy 具体策略角色（多个）
实现抽象策略中的操作，该类含有具体的算法。
使用场景：
● 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。
注意事项：具体策略数量超过 4 个，则需要考虑使用混合模式

策略模式扩展：策略枚举

```
public enum Calculator {
	//加法运算
	ADD("+"){
		public int exec(int a,int b){
			return a+b;
		}
	},
	//减法运算
	SUB("-"){
		public int exec(int a,int b){
			return a - b;
		}
	};
	String value = "";
	//定义成员值类型
	private Calculator(String _value){
		this.value = _value;
	}
	//获得枚举成员的值
	public String getValue(){
		return this.value;
	}
	//声明一个抽象函数
	public abstract int exec(int a,int b);
}
```

定义：

- 它是一个枚举。
- 它是一个浓缩了的策略模式的枚举。

注意：

受枚举类型的限制，每个枚举项都是 public、final、static 的，扩展性受到了一定的约束，因此在系统开发中，策略枚举一般担当不经常发生变化的角色。

致命缺陷：

所有的策略都需要暴露出去，由客户端决定使用哪一个策略。

13.适配器模式（Adapter Pattern）

定义：Convert the interface of a class into another interface clients expect.Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.（将一个类的接口变换成客户端所期待的另一种接口，从而使原本因接口不匹配而无法在一起工作的两个类能够在一起工作。）

类适配器：

● Target 目标角色
该角色定义把其他类转换为何种接口，也就是我们的期望接口，例子中的 IUserInfo 接口就是目标角色。
● Adaptee 源角色
你想把谁转换成目标角色，这个“谁”就是源角色，它是已经存在的、运行良好的类或对象，经过适配器角色的包装，它会成为一个崭新、靓丽的角色。
● Adapter 适配器角色
适配器模式的核心角色，其他两个角色都是已经存在的角色，而适配器角色是需要新建立的，它的职责非常简单：把源角色转换为目标角色，怎么转换？通过继承或是类关联的方式。
使用场景：
你有动机修改一个已经投产中的接口时，适配器模式可能是最适合你的模式。比如系统扩展了，需要使用一个已有或新建立的类，但这个类又不符合系统的接口，怎么办？使用适配器模式，这也是我们例子中提到的。
注意事项：
详细设计阶段不要考虑使用适配器模式，使用主要场景为扩展应用中。

对象适配器：

对象适配器和类适配器的区别：
类适配器是类间继承，对象适配器是对象的合成关系，也可以说是类的关联关系，这是两者的根本区别。（实际项目中对象适配器使用到的场景相对比较多）。

14.迭代器模式（Iterator Pattern）

定义：Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.（它提供一种方法访问一个容器对象中各个元素，而又不需暴露该对象的内部细节。）

● Iterator 抽象迭代器
抽象迭代器负责定义访问和遍历元素的接口，而且基本上是有固定的 3 个方法：first()获得第一个元素，next()访问下一个元素，isDone()是否已经访问到底部（Java 叫做 hasNext()方法）。
● ConcreteIterator 具体迭代器
具体迭代器角色要实现迭代器接口，完成容器元素的遍历。
● Aggregate 抽象容器

容器角色负责提供创建具体迭代器角色的接口，必然提供一个类似 createIterator() 这样的方法，在 Java 中一般是 iterator()方法。

● Concrete Aggregate 具体容器

具体容器实现容器接口定义的方法，创建出容纳迭代器的对象。

ps：迭代器模式已经被淘汰，java 中已经把迭代器运用到各个聚集类（collection）中了，使用 java 自带的迭代器就已经满足我们的需求了。

15.组合模式（(Composite Pattern)）

定义：Compose objects into tree structures to represent part-whole hierarchies.Composite lets clients treat individual objects and compositions of objects uniformly.（将对象组合成树形结构以表示“部分-整体”的层次结构，使得用户对单个对象和组合对象的使用具有一致性。）

● Component 抽象构件角色

定义参加组合对象的共有方法和属性，可以定义一些默认的行为或属性，比如我们例子中的 getInfo 就封装到了抽象类中。

● Leaf 叶子构件

叶子对象，其下再也没有其他的分支，也就是遍历的最小单位。

● Composite 树枝构件

树枝对象，它的作用是组合树枝节点和叶子节点形成一个树形结构。

树枝构件的通用代码：

```
public class Composite extends Component {
     //构件容器
     private ArrayList<Component> componentArrayList = new
ArrayList<Component>();
     //增加一个叶子构件或树枝构件
     public void add(Component component){
             this.componentArrayList.add(component);
     }
     //删除一个叶子构件或树枝构件
     public void remove(Component component){
this.componentArrayList.remove(component);
     }
     //获得分支下的所有叶子构件和树枝构件
     public ArrayList<Component> getChildren(){
             return this.componentArrayList;
     }
}
```

使用场景：

- 维护和展示部分-整体关系的场景，如树形菜单、文件和文件夹管理。
- 从一个整体中能够独立出部分模块或功能的场景。

注意：

只要是树形结构，就考虑使用组合模式。

16.观察者模式（Observer Pattern）

定义：Define a one-to-many dependency between objects so that when one object changes state,all its dependents are notified and updated automatically.（定义对象间一种一对多的依赖关系，使得每当一个对象改变状态，则所有依赖于它的对象都会得到通知并被自动更新。）

- Subject 被观察者

定义被观察者必须实现的职责，它必须能够动态地增加、取消观察者。它一般是抽象类或者是实现类，仅仅完成作为被观察者必须实现的职责：管理观察者并通知观察者。

- Observer 观察者

观察者接收到消息后，即进行 update（更新方法）操作，对接收到的信息进行处理。

- ConcreteSubject 具体的被观察者

定义被观察者自己的业务逻辑，同时定义对哪些事件进行通知。

- ConcreteObserver 具体的观察者

每个观察在接收到消息后的处理反应是不同，各个观察者有自己的处理逻辑。

被观察者通用代码：

```
public abstract class Subject {
     //定义一个观察者数组
     private Vector<Observer> obsVector = new Vector<Observer>();
     //增加一个观察者
     public void addObserver(Observer o){
             this.obsVector.add(o);
     }
     //删除一个观察者
     public void delObserver(Observer o){
             this.obsVector.remove(o);
     }
     //通知所有观察者
     public void notifyObservers(){
             for(Observer o:this.obsVector){
                     o.update();
}
```

```
        }
}
```

使用场景：
- 关联行为场景。需要注意的是，关联行为是可拆分的，而不是“组合”关系。
- 事件多级触发场景。
- 跨系统的消息交换场景，如消息队列的处理机制。

注意：
- 广播链的问题

在一个观察者模式中最多出现一个对象既是观察者也是被观察者，也就是说消息最多转发一次（传递两次）。
- 异步处理问题

观察者比较多，而且处理时间比较长，采用异步处理来考虑线程安全和队列的问题。

17.门面模式（Facade Pattern）

定义：Provide a unified interface to a set of interfaces in a subsystem.Facade defines a higher-level interface that makes the subsystem easier to use.（要求一个子系统的外部与其内部的通信必须通过一个统一的对象进行。门面模式提供一个高层次的接口，使得子系统更易于使用。）

- Facade 门面角色

客户端可以调用这个角色的方法。此角色知晓子系统的所有功能和责任。一般情况下，本角色会将所有从客户端发来的请求委派到相应的子系统去，也就说该角色没有实际的业务逻辑，只是一个委托类。
- subsystem 子系统角色

可以同时有一个或者多个子系统。每一个子系统都不是一个单独的类，而是一个类的集合。子系统并不知道门面的存在。对于子系统而言，门面仅仅是另外一个客户端而已。

使用场景：
- 为一个复杂的模块或子系统提供一个供外界访问的接口
- 子系统相对独立——外界对子系统的访问只要黑箱操作即可
- 预防低水平人员带来的风险扩散

注意：
- 一个子系统可以有多个门面
- 门面不参与子系统内的业务逻辑

18.备忘录模式（Memento Pattern）

定义：Without violating encapsulation,capture and externalize an object's internal state so that the object can be restored to this state later.（在不破坏封装性的前提下，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。）

● Originator 发起人角色
记录当前时刻的内部状态，负责定义哪些属于备份范围的状态，负责创建和恢复备忘录数据。
● Memento 备忘录角色（简单的 javabean）
负责存储 Originator 发起人对象的内部状态，在需要的时候提供发起人需要的内部状态。
● Caretaker 备忘录管理员角色（简单的 javabean）
对备忘录进行管理、保存和提供备忘录。

使用场景：
● 需要保存和恢复数据的相关状态场景。
● 提供一个可回滚（rollback）的操作。
● 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：
●备忘录的生命期
●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（比如一个 for 循环中）。

clone 方式备忘录：

● 发起人角色融合了发起人角色和备忘录角色，具有双重功效

多状态的备忘录模式

● 增加了一个 BeanUtils 类，其中 backupProp 是把发起人的所有属性值转换到 HashMap 中，方便备忘录角色存储。restoreProp 方法则是把 HashMap 中的值返回到发起人角色中。

BeanUtil 工具类代码：

```
public class BeanUtils {
     //把 bean 的所有属性及数值放入到 Hashmap 中
     public static HashMap<String,Object> backupProp(Object bean){
             HashMap<String,Object> result = new
HashMap<String,Object>();
             try {
```

```
                    //获得 Bean 描述
                    BeanInfo
beanInfo=Introspector.getBeanInfo(bean.getClass());
                    //获得属性描述
                    PropertyDescriptor[]
descriptors=beanInfo.getPropertyDescriptors();
                    //遍历所有属性
                    for(PropertyDescriptor des:descriptors){
                            //属性名称
                            String fieldName = des.getName();
                            //读取属性的方法
                            Method getter = des.getReadMethod();
                            //读取属性值
                            Object fieldValue=getter.invoke(bean,new
Object[]{});
                    if(!fieldName.equalsIgnoreCase("class")){
                            result.put(fieldName, fieldValue);
                    }
               }
          } catch (Exception e) {
               //异常处理
          }
          return result;
     }
     //把 HashMap 的值返回到 bean 中
     public static void restoreProp(Object bean,HashMap<String,Object>
propMap){
try {
               //获得 Bean 描述
               BeanInfo beanInfo =
Introspector.getBeanInfo(bean.getClass());
               //获得属性描述
               PropertyDescriptor[] descriptors =
beanInfo.getPropertyDescriptors();
               //遍历所有属性
               for(PropertyDescriptor des:descriptors){
                    //属性名称
                    String fieldName = des.getName();
                    //如果有这个属性
                    if(propMap.containsKey(fieldName)){
                         //写属性的方法
                         Method setter = des.getWriteMethod();
                         setter.invoke(bean, new
Object[]{propMap.get(fieldName)});
```

```
                    }
                }
            } catch (Exception e) {
                //异常处理
                System.out.println("shit");
                e.printStackTrace();
            }
        }
}
```

多备份的备忘录：略
封装得更好一点：保证只能对发起人可读

●建立一个空接口 IMemento——什么方法属性都没有的接口，然后在发起人 Originator 类中建立一个内置类（也叫做类中类）Memento 实现 IMemento 接口，同时也实现自己的业务逻辑。

19.访问者模式（Visitor Pattern）

定义：Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates. （封装一些作用于某种数据结构中的各元素的操作，它可以在不改变数据结构的前提下定义作用于这些元素的新的操作。）

● Visitor——抽象访问者
抽象类或者接口，声明访问者可以访问哪些元素，具体到程序中就是 visit 方法的参数定义哪些对象是可以被访问的。
● ConcreteVisitor——具体访问者
它影响访问者访问到一个类后该怎么干，要做什么事情。
● Element——抽象元素
接口或者抽象类，声明接受哪一类访问者访问，程序上是通过 accept 方法中的参数来定义的。
● ConcreteElement——具体元素
实现 accept 方法，通常是 visitor.visit(this)，基本上都形成了一种模式了。
● ObjectStruture——结构对象
元素产生者，一般容纳在多个不同类、不同接口的容器，如 List、Set、Map 等，在项目中，一般很少抽象出这个角色。

使用场景：

- 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，也就说是用迭代器模式已经不能胜任的情景。
- 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。

20.状态模式（复杂）

定义：Allow an object to alter its behavior when its internal state changes.The object will appear to change its class.（当一个对象内在状态改变时允许其改变行为，这个对象看起来像改变了其类。）

- State——抽象状态角色

接口或抽象类，负责对象状态定义，并且封装环境角色以实现状态切换。

- ConcreteState——具体状态角色

每一个具体状态必须完成两个职责：本状态的行为管理以及趋向状态处理，通俗地说，就是本状态下要做的事情，以及本状态如何过渡到其他状态。

- Context——环境角色

定义客户端需要的接口，并且负责具体状态的切换。

使用场景：

- 行为随状态改变而改变的场景

这也是状态模式的根本出发点，例如权限设计，人员的状态不同即使执行相同的行为结果也会不同，在这种情况下需要考虑使用状态模式。

- 条件、分支判断语句的替代者

注意：

状态模式适用于当某个对象在它的状态发生改变时，它的行为也随着发生比较大的变化，也就是说在行为受状态约束的情况下可以使用状态模式，而且使用时对象的状态最好不要超过 5 个。

21.解释器模式（Interpreter Pattern）（少用）

定义：Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.（给定一门语言，定义它的文法的一种表示，并定义一个解释器，该解释器使用该表示来解释语言中的句子。）

● AbstractExpression——抽象解释器
具体的解释任务由各个实现类完成，具体的解释器分别由 TerminalExpression 和 Non-terminalExpression 完成。
● TerminalExpression——终结符表达式
实现与文法中的元素相关联的解释操作，通常一个解释器模式中只有一个终结符表达式，但有多个实例，对应不同的终结符。具体到我们例子就是 VarExpression 类，表达式中的每个终结符都在栈中产生了一个 VarExpression 对象。
● NonterminalExpression——非终结符表达式
文法中的每条规则对应于一个非终结表达式，具体到我们的例子就是加减法规则分别对应到 AddExpression 和 SubExpression 两个类。非终结符表达式根据逻辑的复杂程度而增加，原则上每个文法规则都对应一个非终结符表达式。
● Context——环境角色
具体到我们的例子中是采用 HashMap 代替。

使用场景：
● 重复发生的问题可以使用解释器模式
● 一个简单语法需要解释的场景

注意：
尽量不要在重要的模块中使用解释器模式，否则维护会是一个很大的问题。在项目中可以使用 shell、JRuby、Groovy 等脚本语言来代替解释器模式，弥补 Java 编译型语言的不足。

22.享元模式（Flyweight Pattern）

定义：Use sharing to support large numbers of fine-grained objects efficiently.
（使用共享对象可有效地支持大量的细粒度的对象。）
对象的信息分为两个部分：内部状态（intrinsic）与外部状态（extrinsic）。
● 内部状态
内部状态是对象可共享出来的信息，存储在享元对象内部并且不会随环境改变而改变。
● 外部状态
外部状态是对象得以依赖的一个标记，是随环境改变而改变的、不可以共享的状态。

● Flyweight——抽象享元角色
它简单地说就是一个产品的抽象类，同时定义出对象的外部状态和内部状态的接口或实现。
● ConcreteFlyweight——具体享元角色
具体的一个产品类，实现抽象角色定义的业务。该角色中需要注意的是内部状态处理应该与环境无关，不应该出现一个操作改变了内部状态，同时修改了外部状态，这是绝对不允许的。

● unsharedConcreteFlyweight——不可共享的享元角色
不存在外部状态或者安全要求（如线程安全）不能够使用共享技术的对象，该对象一般不会出现在享元工厂中。
● FlyweightFactory——享元工厂
职责非常简单，就是构造一个池容器，同时提供从池中获得对象的方法。

享元工厂的代码：

```
public class FlyweightFactory {
     //定义一个池容器
     private static  HashMap<String,Flyweight> pool= new
HashMap<String,Flyweight>();
     //享元工厂
     public static Flyweight getFlyweight(String Extrinsic){
             //需要返回的对象
             Flyweight flyweight = null;
             //在池中没有该对象
             if(pool.containsKey(Extrinsic)){
                     flyweight = pool.get(Extrinsic);
             }else{
                     //根据外部状态创建享元对象
                     flyweight = new ConcreteFlyweight1(Extrinsic);
                     //放置到池中
                     pool.put(Extrinsic, flyweight);
             }
             return flyweight;
     }
}
```

使用场景：
● 系统中存在大量的相似对象。
● 细粒度的对象都具备较接近的外部状态，而且内部状态与环境无关，也就是说对象没有特定身份。
● 需要缓冲池的场景。
注意：
● 享元模式是线程不安全的，只有依靠经验，在需要的地方考虑一下线程安全，在大部分场景下不用考虑。对象池中的享元对象尽量多，多到足够满足为止。
● 性能安全：外部状态最好以 java 的基本类型作为标志，如 String，int，可以提高效率。

23.桥梁模式（Bridge Pattern）

定义：Decouple an abstraction from its implementation so that the two can vary independently.（将抽象和实现解耦，使得两者可以独立地变化。）

● Abstraction——抽象化角色
它的主要职责是定义出该角色的行为，同时保存一个对实现化角色的引用，该角色一般是抽象类。
● Implementor——实现化角色
它是接口或者抽象类，定义角色必需的行为和属性。
● RefinedAbstraction——修正抽象化角色
它引用实现化角色对抽象化角色进行修正。
● ConcreteImplementor——具体实现化角色
它实现接口或抽象类定义的方法和属性。

使用场景：
● 不希望或不适用使用继承的场景
● 接口或抽象类不稳定的场景
● 重用性要求较高的场景

注意：
发现类的继承有 N 层时，可以考虑使用桥梁模式。桥梁模式主要考虑如何拆分抽象和实现。

设计原则：

●Single Responsibility Principle：单一职责原则

单一职责原则有什么好处：
● 类的复杂性降低，实现什么职责都有清晰明确的定义；
● 可读性提高，复杂性降低，那当然可读性提高了；
● 可维护性提高，可读性提高，那当然更容易维护了；
●变更引起的风险降低，变更是必不可少的，如果接口的单一职责做得好，一个接口修改只对相应的实现类有影响，对其他的接口无影响，这对系统的扩展性、维护性都有非常大的帮助。

ps：接口一定要做到单一职责，类的设计尽量做到只有一个原因引起变化。
单一职责原则提出了一个编写程序的标准，用“职责”或“变化原因”来衡量接口或类设计得是否优良，但是“职责”和“变化原因”都是不可度量的，因项目而异，因环境而异。

● Liskov Substitution Principle：里氏替换原则

定义：Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.
（所有引用基类的地方必须能透明地使用其子类的对象。）
通俗点讲，只要父类能出现的地方子类就可以出现，而且替换为子类也不会产生任何错误或异常，使用者可能根本就不需要知道是父类还是子类。但是，反过来就不行了，有子类出现的地方，父类未必就能适应。
定义中包含的四层含义：
1.子类必须完全实现父类的方法
2.子类可以有自己的个性
3.覆盖或实现父类的方法时输入参数可以被放大
　　如果父类的输入参数类型大于子类的输入参数类型，会出现父类存在的地方，子类未必会存在，因为一旦把子类作为参数传入，调用者很可能进入子类的方法范畴。

4. 覆写或实现父类的方法时输出结果可以被缩小
　　父类的一个方法的返回值是一个类型 T，子类的相同方法（重载或覆写）的返回值为 S，那么里氏替换原则就要求 S 必须小于等于 T，也就是说，要么 S 和 T 是同一个类型，要么 S 是 T 的子类。
● Interface Segregation Principle：接口隔离原则

接口分为两种：
实例接口（Object Interface）：Java 中的类也是一种接口
类接口（Class Interface）： Java 中经常使用 Interface 关键字定义的接口

隔离：建立单一接口，不要建立臃肿庞大的接口；即接口要尽量细化，同时接口中的方法要尽量少。
接口隔离原则与单一职责原则的不同：接口隔离原则与单一职责的审视角度是不相同的，单一职责要求的是类和接口职责单一，注重的是职责，这是业务逻辑上的划分，而接口隔离原则要求接口的方法尽量少。

● Dependence Inversion Principle：依赖倒置原则

原始定义：
①高层模块不应该依赖低层模块，两者都应该依赖其抽象；
②抽象不应该依赖细节（实现类）；
③细节应该依赖抽象。
依赖倒置原则在 java 语言中的体现：
①模块间的依赖通过抽象发生，实现类之间不发生直接的依赖关系，其依赖关系是通过接口或抽象类产生的；
②接口或抽象类不依赖于实现类；

③实现类依赖接口或抽象类。
依赖的三种写法：
①构造函数传递依赖对象（构造函数注入）
②Setter 方法传递依赖对象（setter 依赖注入）
③接口声明依赖对象（接口注入）
使用原则：
依赖倒置原则的本质就是通过抽象（接口或抽象类）使各个类或模块的实现彼此独立，不互相影响，实现模块间的松耦合，我们怎么在项目中使用这个规则呢？只要遵循以下的几个规则就可以：
①每个类尽量都有接口或抽象类，或者抽象类和接口两者都具备
②变量的表面类型尽量是接口或者是抽象类
③任何类都不应该从具体类派生（只要不超过两层的继承是可以忍受的）
④尽量不要复写基类的方法
⑤结合里氏替换原则使用

●Open Closed Principle：开闭原则

定义：软件实体应该对扩展开放，对修改关闭。
其含义是说一个软件实体应该通过扩展来实现变化，而不是通过修改已有的代码来实现变化。
软件实体：项目或软件产品中按照一定的逻辑规则划分的模块、抽象和类、方法。
变化的三种类型：
①逻辑变化
只变化一个逻辑，而不涉及其他模块，比如原有的一个算法是 a*b+c，现在需要修改为 a*b*c，可以通过修改原有类中的方法的方式来完成，前提条件是所有依赖或关联类都按照相同的逻辑处理。
②子模块变化
一个模块变化，会对其他的模块产生影响，特别是一个低层次的模块变化必然引起高层模块的变化，因此在通过扩展完成变化时，高层次的模块修改是必然的。
③可见视图变化
可见视图是提供给客户使用的界面，如 JSP 程序、Swing 界面等，该部分的变化一般会引起连锁反应（特别是在国内做项目，做欧美的外包项目一般不会影响太大）。可以通过扩展来完成变化，这要看我们原有的设计是否灵活。