参考：http://c.biancheng.net/view/8375.html

⚫软件设计名词

OOD - Object-Oriented Design(面向对象设计)

OOM - Object-Oriented Method(面向对象方法)

OOA - Object-Oriented Analysis(面向对象分析)

OOP - Object-Oriented Programming（面向对象编程）

DDD – Domain Driven Design 领域驱动设计：是一套综合软件系统分析和设计的面向对象建模方法

⚫什么是UML？

统一建模语言：用图来表达面向对象的软件设计思路

分类：用例图、类图、对象图、状态图、活动图、时序图、协作图、构件图、部署图等

⚫什么是类图？

类图是用来显示系统中类、接口他们之间静态结构和关系的一种静态模型

**接口：**



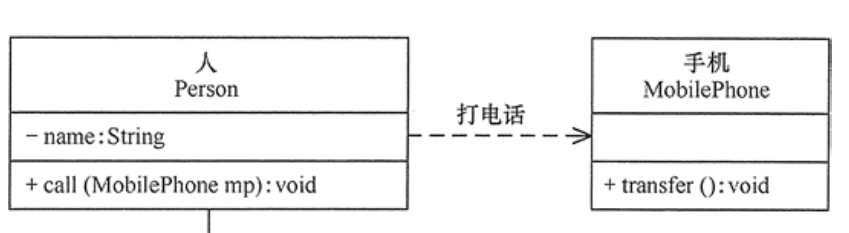
**类：**



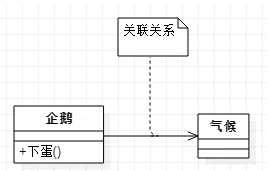
⚫UML中类之间的关系有哪些？

类与类之间的耦合度从弱到强：依赖、关联、聚合、组合、泛化和实现，泛化和实现耦合度相同。

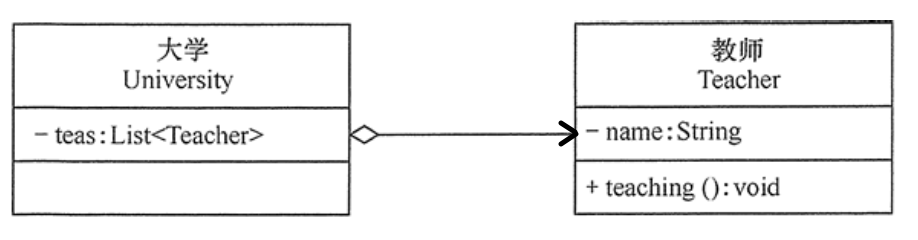
**依赖**：临时建立的关联：比如人通过手机打电话，人和手机



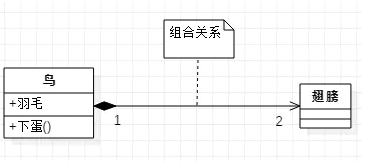
**关联**：对象之间互相引用的关系，可以是单向关联或双向关联：例如订单和商品



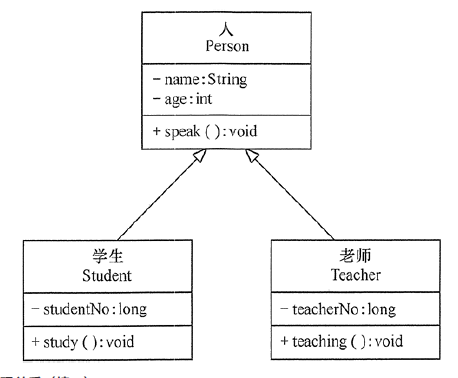
**聚合**：是一种整体和部分之间的关系，也是通过成员对象来实现的，其中成员对象是整体对象的一部分，但是成员对象可以脱离整体对象而独立存在，例如学校和老师



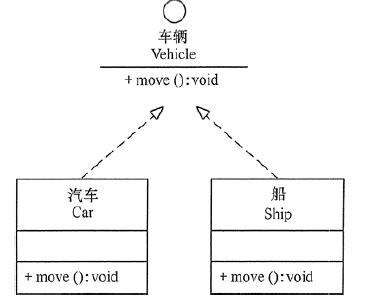
**组合**：也是一种整体和部分之间的关系，但是是一种更强的聚合关系，整体对象可以控制部分对象的生命周期，整体对象不存在，部分对象也不会存在，部分对象不能脱离整体而独立存在



**泛化（继承类）**：类的继承关系



**实现（实现接口）**：类实现接口的关系



⚫面向对象七大设计原则



**1. 开闭原则：**对扩展开放，对修改关闭

**2. 单一职责：**一个类只负责一项职责，只实现一种业务功能，也就是控制类的大小粒度，尽可能把业务细分成最小的颗粒度，这样修改一个类的功能时，不会影响另外一个类的功能

**3. 里氏替换原则：**程序中将一个父类对象替换成子类对象，程序将不会产生任何错误和异常。就是使用父类对象的地方都可以使用子类对象，**子类可以扩展父类的功能，但是不能改变父类原有的功能，关键点在于不能覆盖父类的非抽象方法，这只着重强调的。**

**4. 依赖倒置原则：**抽象不应该依赖细节，细接应该依赖抽象。就是要针对接口编程，不能针对实现编程，**核心就是面向抽象(抽象类或者接口)编程**

**5. 接口隔离原则：**每隔接口只承担一种独立的角色，接口中不要放很多方法，显得很臃肿，尽量把接口细化，每隔接口只实现一种功能**，就是当我们创建一个接口的时候，尽量进行拆分成最小粒度，避免耦合**

**与单一职责相比；单一职责更注重业务层面的职责划分，而接口隔离注重的是接口设计方法要尽可能少**

场景举例：见文档

**6. 合成复用原则：**复用时尽量使用对象组合/聚合关系，少用继承，相对继承而言，组合/聚合关系，类的耦合度较低，类之间的变化影响会降低

**7. 迪米特法则：**不要和陌生人说话，只与你的直接朋友通信，使类与类之间保持松散的耦合度

朋友包括：

1. 当前对象this

2. 以参数形式传入当前对象方法中的对象

3. 当前对象成员对象

4. 如果当前成员遍历是一个集合，那么集合中的元素对象也是朋友

5. 当前对象中所创建的对象

⚫23种设计模式

**前言：不要为了使用设计模式而使用设计模式，而是在开发中利用七大设计原则，设计出一个可复用性，可扩展性，可维护性，低耦合高内聚的框架，这才是使用设计模式的核心，要避免设计过度，同时使用多种设计模式，这会使代码变的更复杂，使用设计模式是为了更简单灵活**

1. **创建型：用于描述“怎样创建对象”，在创建对象的同时隐藏创建逻辑的方式，而不是使用new运算符直接实例化对象，它的主要特点是“将对象的创建与使用分离**

* **工厂模式**（Factory Pattern）：我们在创建对象时不会对客户端暴露创建逻辑，并且是通过使用一个共同的接口或者类（简单工厂使用普通类创建对象）来创建一个对象。

角色：

简单工厂：使用普通类，付责创建对象的内部逻辑

抽象产品：所有产品的抽象接口

具体产品：工厂创建的具体产品对象

**实践**：创建课程业务，定义一个抽象课程接口，然后根据课程类型（培训类型，管理类型，技术类型等）实现抽象课程接口，每个实现类代表一种类型的课程，每种课程类型创建内容都不一致，包括课程分类，标签，权限等，然后创建一个工厂类，根据前端选择的类型，创建不同的课程。

* 抽象工厂模式（Abstract Factory Pattern）：用一个超级工程（接口）来创建其他工厂，每隔工厂付责不同类型的产品，这种模式适用于多种产品族时，例如我生产肉类，和蔬菜类，两种产品，那么可以使用抽象工厂模式
* 单例模式（Singleton Pattern）：某个类只能生成一个实例，该类提供了一个全局访问点供外部获取该实例，可以减少内存的开销，缺点：没有接口，扩展困难，违背开闭原则

1. 饿汉式

public class Single(){

private static Single single = new Single();

private Single(){};

public static Single getInstance(){

return single;

}

}

1. 懒汉式-DCL

public class Single(){

private volatile static Single single = null;

private Single(){};

public static Single getInstance(){

if(single==null){

synchronized(Single.class){

if(single==null){

single = new Single();

}

}

}

return single;

}

}

1. 懒汉式-枚举

public enum Singleton {

INSTANCE;

public void whateverMethod() {

}

}

* 建造者模式（Builder Pattern）：将一个复杂对象分解成多个相对简单的部分，然后根据不同需要分别创建它们，最后构建成该复杂对象。

角色：

产品角色：产品实体类（Dto），这是一个复杂的对象，其中包含很多零部件，或者可以是抽象产品接口，然后子类实现具体产品

抽象建造者：付责各个零部件的组装，每个零部件是一个抽象方法，通常还包含一个返回复杂产品的方法 getResult()，

具体建造者：实现各个零件的创建方法

指挥者：调用建造者，根据零部件方法，完成复杂产品的创建，只负责调用

优点：封装性好，构建和表示分离，各个具体建造者相互独立，有利于系统解耦

缺点：产品要有共同点，组成部分必须相同，使用范围被限制

与工厂模式相比：建造者模式更关注对象的组装过程。而工厂模式关注：对象的创建的结果，客户不需要只知道是怎么创建的。

场景：创建简单对象用工厂，复杂对象用建造者模式，适用产品比较复杂，相同的方法执行顺序不同，产生的结果也不同，Stringbuilder适用的是建造者模式

* 原型模式（Prototype Pattern）：将一个对象作为原型，通过对其进行复制而克隆出多个和原型类似的新实例。用这种方式创建对象非常高效，不需要知道创建对象的细接，Java 自带的原型模式基于内存二进制流的复制，在性能上比直接 new 一个对象更加优良

**分类**：

1. 浅拷贝：对于基本数据类型和字符串，创建对象，对于引用型类型，只把引用指向了新对象，实际并没有创建新对象，实现接口Cloneable重写clone()方法，super.clone

2. 深拷贝：创建一个新对象，属性中引用的其他对象也会被克隆，不再指向原有对象地址，

实现：

1）引用类型也实现Cloneable重写clone

2）实现Serializable通过序列化反序列化实现

bos = new ByteArrayOutputStream();

oos = new ObjectOutputStream(bos);

oos.writeObject(this); //序列化

bis = new ByteArrayInputStream(bos.toByteArray());//反序列化操作

ois = new ObjectInputStream(bis);

DeepProtoType deepProtoType = (DeepProtoType) ois.readObject();

缺点：违反开闭原则，深克隆需要复杂代码

场景：例如 scope='prototype'、JSON.parseObject() 等都是原型模式的具体应用

，适用在系统中大量适用该对象，调用者需要对该对象重新赋值，或者创建对象成本比较大，需要繁琐的数据准备等场景

1. **结构型：用于描述如何将类或对象按某种布局组成更大的结构**

* 适配器模式（Adapter Pattern）：使两个原本不兼容的接口，通过中间的适配器进行适配，能够完美的在一起工作。

角色：

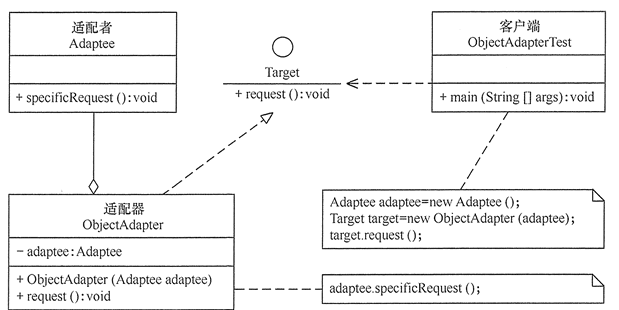
目标接口：被适配的接口（比如）

适配者：它是替代目标接口被访问，并适配目标接口的接口

适配器：他是一个转换器，通过聚合或者继承适配者，把适配者接口转换成目标接口，让客户本来是访问不了目标接口的，通过适配器转换，直接访问适配者就可以了。

双向适配器，同时实现目标接口和适配者接口

类图：



优点：不需要修改原有代码，而重用适配者就行，目标方法和适配者解耦，并解决了两者之间的差异，符合开闭原则。

注意：适配器模式分为类结构型模式和对象结构型模式两种，前者类之间的耦合度比后者高（前者用继承后者用聚合，所以使用后者），且要求程序员了解现有组件库中的相关组件的内部结构，所以应用相对较少些

//对象适配器类中的适配器，Target为目标对象，Adaptee为适配者，聚合关系降低耦合

class ObjectAdapter implements Target

{

private Adaptee adaptee;

public ObjectAdapter(Adaptee adaptee)

{

this.adaptee=adaptee;

}

@Override

public void request() //目标接口中方法

{

adaptee.specificRequest(); // 适配者的方法

}

}

场景：1.以前接口不满足新系统的需求，需要改造适配现在的新系统，2.第三方组件系统接口和我们系统接口不适配的情况

* 桥接模式（Bridge Pattern）
* 过滤器模式（Filter、Criteria Pattern）
* 组合模式（Composite Pattern）
* 装饰器模式（Decorator Pattern）：在不改变现有对象结构的情况下，动态的给该对象添加一些额外的功能。就是在原有的对象基础上，包装一些额外功能，创建一个新类，继承并聚合主体

角色：

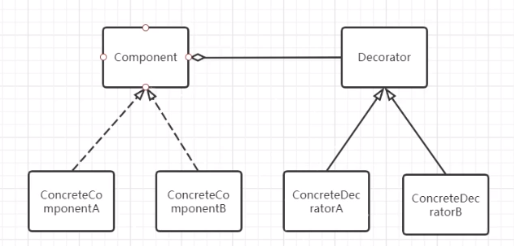
抽象主体(Component)：被装饰的抽象类

具体主体：被装饰的具体对象

装饰者(Decorator)：实际上是一个实现抽象主题的类，并用，有参构造方法聚合抽象主体类。

具体装饰者：继承装饰者，实现具体的装饰功能，调用者直接调用具体装饰者就可以实现装饰功能了

类图：



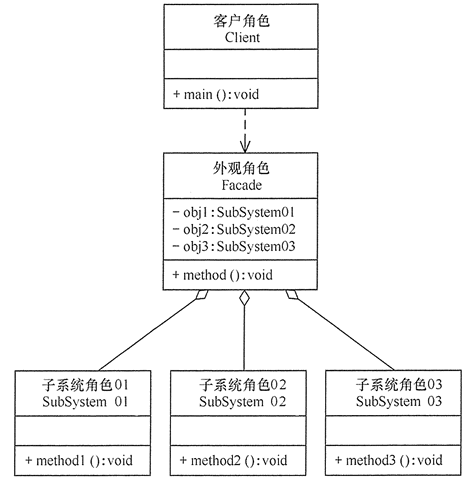
优点：装饰类和被装饰类不会互相耦合，利于扩展，符合开闭原则

缺点：多层装饰比较复杂

场景：扩展一个类的功能，动态增加功能

* 外观模式（Facade Pattern）：主要是定义了一个高层接口。它包含了对各个子系统的引用，客户端可以通过它访问各个子系统的功能

类图：



* 享元模式（Flyweight Pattern）：享元模式其实是工厂方法模式的一个改进机制，享元模式同样要求创建一个或一组对象，并且就是通过工厂方法模式生成对象的，只不过享元模式为工厂方法模式增加了缓存这一功能
* 代理模式（Proxy Pattern）：在客户端与目标对象之间起到一个中介作用和保护目标对象的作用；

分类：

静态代理：程序员自己编写代理类，跟普通类一样编译运行

动态代理：在程序运行期间，通过反射，动态的生成代理类JDK,CGLIB

角色：

抽象目标，具体目标，代理类

优点：代理对象可以扩展目标对象的功能，可以是目标对象和客户端中间解耦，增加了程序的可扩展性

缺点：增加一层代理，会使请求速度变慢，代理类的增加带来复杂性

场景：

1. **行为型：用于描述类或对象之间怎样相互协作共同完成单个对象都无法单独完成的任务，以及怎样分配职责，这些设计模式特别关注对象之间的通信。**

* 责任链模式（Chain of Responsibility Pattern）：当有一个请求者多个处理者时，请求者发送一个请求，每一个处理者持有下一个处理者的引用，由多个处理者组成一条处理链，请求就可以按照这条处理链来处理请求，直到结束。该模式为了发送请求和处理者解耦，并让请求在处理链中传递

角色：

抽象处理者、具体处理者，客户类

优点：降低请求发送者和处理者的耦合度，可根据需求增加请求处理者，利于扩展，也可以动态调整责任链中的顺序，灵活性更高

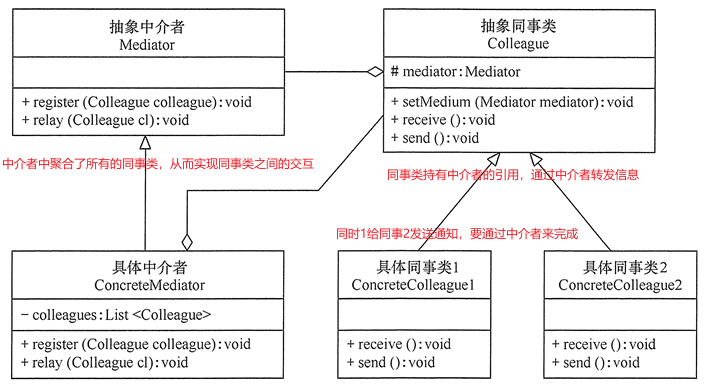
缺点：较长的责任链会带来性能问题

场景：有多个处理者

实现：在造价系统中，需要计算建筑行业的钢筋重量，前端添加计算模型触发计算后，后端有多种类型的钢筋种类，例如拉近，面筋，底筋等类型，由于每种钢筋的计算方法不一样。所以采用责任链模式，一个请求计算多个钢筋，定义了一个抽象钢筋计算类，该类还包含抽象计算方法等待子类去实现，这个抽象类实现Compareable接口，并重写compareTo方法实现自己定义排序，根据此接口可以实现有序的处理链，然后每种类型钢筋继承计算类。实现自己的算法，接收到请求后，通过applicationContext.getBeanOfType(Abstract.class)方法，获取所有子类，把子类放入集合中，排序，循环执行每个子类的计算方法。这样一个请求，多个处理者来处理，达到了解耦的目的，同时还可以增加子类，利于扩展其他计算类

* 命令模式（Command Pattern）
* 解释器模式（Interpreter Pattern）
* 迭代器模式（Iterator Pattern）
* 中介者模式（Mediator Pattern）：中介者模式将原本多个对象直接的相互依赖变成了中介者和多个同事类的依赖关系。对象之间的交互通过中介者来实现交互

类图：



* 备忘录模式（Memento Pattern）
* 观察者模式（Observer Pattern）：指多个对象间存在一对多的依赖关系，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。这种模式有时又称作发布-订阅模式

角色：

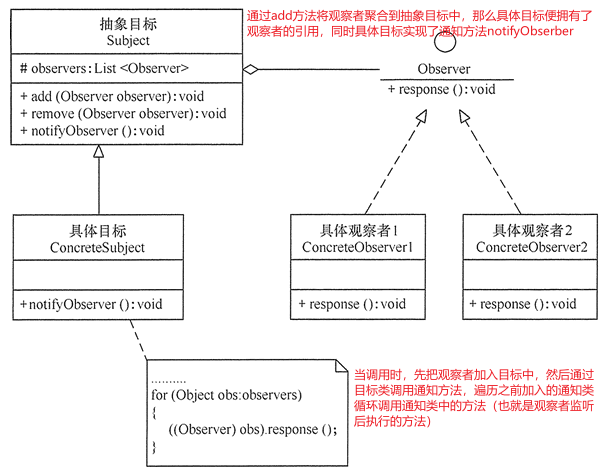
抽象目标：观察者所观察的目标，用来聚合观察者，

具体目标：根据聚合的观察者，付责通知观察者

抽象观察者：接口

具体观察者：实现接口的观察者

类图：



优点：触发机制，降低耦合

缺点：可能发送循环依赖

注意：目标对象和观察对象不能直接调用，否则将两者紧密耦合

应用：spring事件机制中有三个角色，

**事件**：ApplicationEvent就是事件，继承jdk中的EventObject，spring中所有的事件都要继承ApplicationEvent，并且通过source得到事件源，该类的实现类ApplicationContextEvent表示ApplicaitonContext的容器事件

**事件源（事件发布）**：spring底层组件ApplicationEventPublisher提供了发布事件的方法publishEvent(ApplicationEvent event);，发布事件，监听者就能监听到消息，即"通知"观察者。ApplicationContext继承了该类，所以拥有事件发布能力，ApplicationContextEvent是对应ApplicationContext的事件

**事件监听器**：ApplicationListener（抽象类，继承jdk的EventListener）就是监听器，spring中所有监听器都要继承该类，这个接口只有一个onApplicationEvent()方法，该方法接受一个ApplicationEvent或其子类对象作为参数在方法体中，可以通过对Event类型的判断来进行相应的处理（if(event instanceof MyEvent)），当事件触发时，所有的监听器就会收到消息

* 状态模式（State Pattern）
* 空对象模式（Null Object Pattern）
* 策略模式（Strategy Pattern）：定义一系列的算法，然后把每个算法封装起来，使他们可以互相替换，且算法的替换不会影响我们客户的使用，

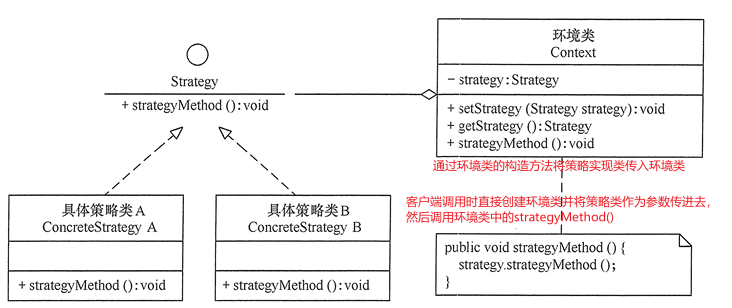
角色：

抽象策略：抽象策略父类，可以定义公共代码，和抽象算法方法。

具体策略：具体实现每个算法

环境类：持有算法类的引用，使用算法的人通过环境类调用不用的算法，

类图：



代码：

public class StrategyPattern {

public static void main(String[] args) {

Context c = new Context();

c.setStrategy(new ConcreteStrategyA());

c.strategyMethod();

}

}

//抽象策略类

interface Strategy {

public void strategyMethod(); //策略方法

}

//具体策略类A

class ConcreteStrategyA implements Strategy {

public void strategyMethod() {

System.out.println("具体策略A的策略方法被访问！");

}

}

//具体策略类B

class ConcreteStrategyB implements Strategy {

public void strategyMethod() {

System.out.println("具体策略B的策略方法被访问！");

}

}

//环境类

class Context {

private Strategy strategy;

public Strategy getStrategy() {

return strategy;

}

public void setStrategy(Strategy strategy) {

this.strategy = strategy;

}

public void strategyMethod() {

strategy.strategyMethod();

}

}

优点：省去了ifelse的判断，符合开闭原则，客户端不直接和算法打交道，而是通过持有策略类应用的环境类打交道，高内聚低耦合，利于扩展，并且算法中公共代码，可以移到父类中，避免重复代码

缺点：算法太多不利于维护，并且要理解算法之间的区别

注意：客户端管理所有策略算法将变得很复杂，如果在环境类中使用策略工厂模式来管理这些策略类将大大减少客户端的工作复杂度

场景：在系统中有多种功能类似，且互相完全独立的场景，可以把每种判断封装成一个策略类，然后动态的去替换选择

应用：

bean实例化策略模式：策略模式的类InstantiationStrategy 来创建bean。InstantiationStrategy有两个实现类

1. SimpleInstantiationStrategy：其功能就如同类名一样，只是简单的根据构造函数和参数来创建bean

2. CglibSubclassingInstantiationStrategy：在SimpleInstantiationStrategy的基础上，添加了根据Cglib来动态生成子类的功能。

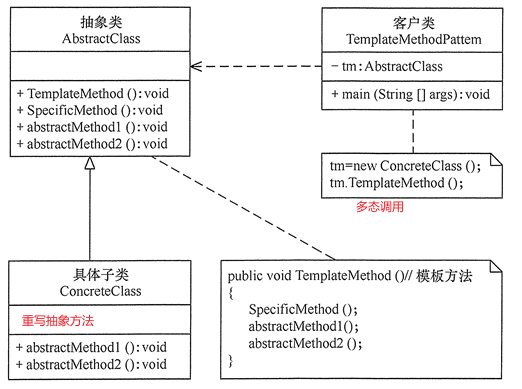
* 模板模式（Template Pattern）：定义一系列的操作步骤，把一些相同的步骤写在父类中，而一些未知的步骤的实现交给子类去实现，使得子类在不改变步骤结构的情况下，重新定义步骤中具体的内容

角色

抽象类：模板方法（包含以下三种）、抽象方法（子类实现），具体方法（模板公共方法），钩子方法（普通方法，控制模板方法中步骤的逻辑）

具体子类：实现抽象类中的模板方法、钩子方法（可不重写，控制业务）

类图



优点：把不变的部分封装在父类，代码复用，扩展的部分交给子类，便于子类继续扩展，符合开闭原则

缺点：通过钩子方法可以控制父类逻辑，增加了复杂性，父类增加抽象方法，所以子类都要改

场景：适合整体步骤固定，但其中部分易变

实践：spring中的refresh()模板方法

应用：钢筋计算步骤大概分为：解析钢筋代号（不同），获取代号对应的值（相同），代号复制（相同），获取公式，把代号代入公式计算出长度，用长度乘以每米的重量算出重量。所有钢筋的计算步骤大部分一直都写在父类中，有个别方法不一致，留给子类去实现。

* 访问者模式（Visitor Pattern）

**参考：**

[**http://c.biancheng.net/view/1354.html**](http://c.biancheng.net/view/1354.html)

[**https://www.runoob.com/design-pattern/builder-pattern.html**](https://www.runoob.com/design-pattern/builder-pattern.html)