# 类关系

关联：关联分为聚合和组合，聚合相关的东西聚在一起 has-a 语意。组合是部分整体的关系contain-a 语意。关联是一种持久的关系，依赖是暂时的。

### 总原则：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，而是要扩展原有代码，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等，后面的具体设计中我们会提到这点。

# 设计模式的六大原则

## 单一职责原则

不要存在多于一个导致类变更的原因，也就是说每个类应该实现单一的职责，如若不然，就应该把类拆分。

## 里氏替换原则（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。—— From Baidu 百科

历史替换原则中，子类对父类的方法尽量不要重写和重载。因为父类代表了定义好的结构，通过这个规范的接口与外界交互，子类不应该随便破坏它。

## 依赖倒转原则（Dependence Inversion Principle）

这个是开闭原则的基础，具体内容：面向接口编程，依赖于抽象而不依赖于具体。写代码时用到具体类时，不与具体类交互，而与具体类的上层接口交互。

## 接口隔离原则（Interface Segregation Principle）

这个原则的意思是：每个接口中不存在子类用不到却必须实现的方法，如果不然，就要将接口拆分。使用多个隔离的接口，比使用单个接口（多个接口方法集合到一个的接口）要好。

## 迪米特法则（最少知道原则）（Demeter Principle）

就是说：一个类对自己依赖的类知道的越少越好。也就是说无论被依赖的类多么复杂，都应该将逻辑封装在方法的内部，通过public方法提供给外部。这样当被依赖的类变化时，才能最小的影响该类。

最少知道原则的另一个表达方式是：只与直接的朋友通信。类之间只要有耦合关系，就叫朋友关系。耦合分为依赖、关联、聚合、组合等。我们称出现为成员变量、方法参数、方法返回值中的类为直接朋友。局部变量、临时变量则不是直接的朋友。我们要求陌生的类不要作为局部变量出现在类中。

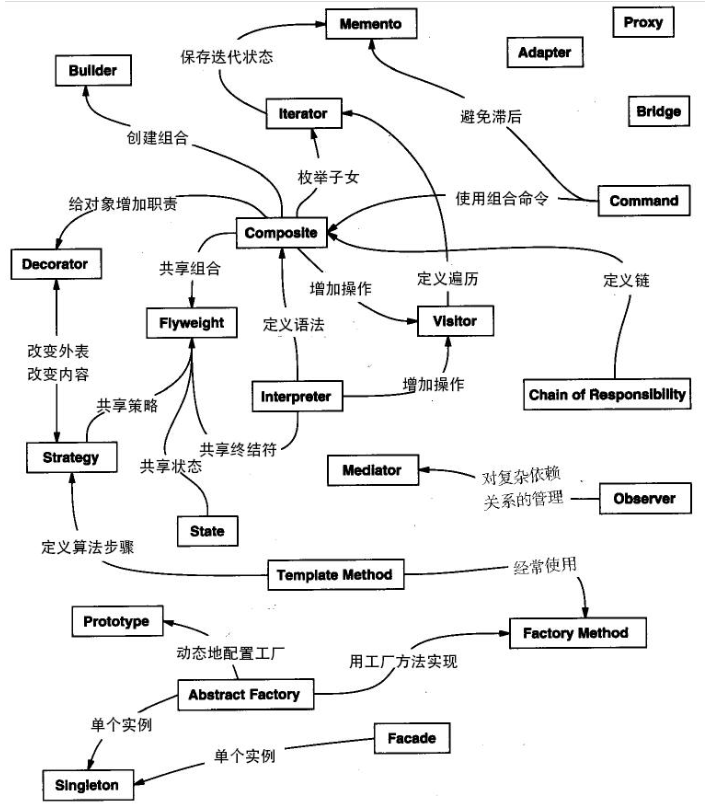
## 合成复用原则（Composite Reuse Principle）

原则是尽量首先使用合成/聚合的方式，而不是使用继承。

# 概念

**设计模式**：软件设计的惯用方案。四要素名称、问题、解决方案、效果。

* 创建型模式（5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（7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（11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。



# 创建型

## 工厂模式

### 简单工厂

简单工厂，直接 new 对象，根据参数生产对象。简单工厂的问题在于要想生产其它类型的产品就要修改工厂类，违背了开闭原则，对修改关闭，对扩展开放。

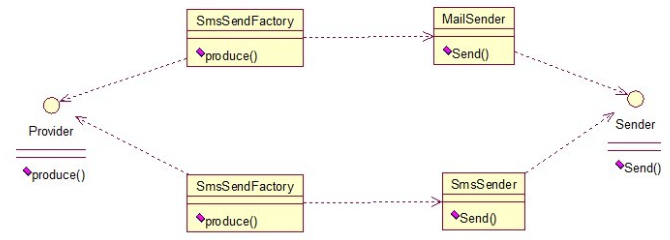
1. **public** **interface** Sender {}
2. **public** **class** MailSender **implements** Sender {}
3. **public** **class** SmsSender **implements** Sender {}
5. **public** **class** SendFactory {
6. **public** Sender produce(String type) {
7. **if** ("mail".equals(type)) {
8. **return** **new** MailSender();
9. } **else** **if** ("sms".equals(type)) {
10. **return** **new** SmsSender();
11. } **else** {
12. System.out.println("请输入正确的类型!");
13. **return** **null**;
14. }
15. }
16. }
18. **public** **class** FactoryTest {
19. **public** **static** **void** main(String[] args) {
20. SendFactory factory = **new** SendFactory();
21. Sender sender = factory.produce("sms");
22. }
23. }

不同的对象，使用不同的方法生产

1. **public** **class** **SendFactory** {
2. **public** Sender produceMail(){
3. **return** **new** MailSender();
4. }
6. **public** Sender produceSms(){
7. **return** **new** SmsSender();
8. }
9. }

## 工厂方法

有子类来生产产品，延迟到子类，缺点是每次增加商品都要添加新的工厂



1. **public** interface Sender {
2. **public** **void** Send();
3. }
4. **public** **class** MailSender implements Sender {
5. @Override
6. **public** **void** Send() {
7. System.out.println("this is mailsender!");
8. }
9. }
10. **public** **class** SmsSender implements Sender {
12. @Override
13. **public** **void** Send() {
14. System.out.println("this is sms sender!");
15. }
16. }
18. **public** interface Provider {
19. **public** Sender produce();
20. }
21. **public** **class** SendMailFactory implements Provider {
23. @Override
24. **public** Sender produce(){
25. **return** **new** MailSender();
26. }
27. }
28. **public** **class** SendSmsFactory implements Provider{
30. @Override
31. **public** Sender produce() {
32. **return** **new** SmsSender();
33. }
34. }
36. **public** **class** Test {
38. **public** **static** **void** main(String[] args) {
39. Provider provider = **new** SendMailFactory();
40. Sender sender = provider.produce();
41. sender.Send();
42. }
43. }

## 单例模式

1. **public** **class** Singleton {
2. /\* 私有构造方法，防止被实例化 \*/
3. **private** Singleton() {
4. }
5. /\* 此处使用一个内部类来维护单例 \*/
6. **private** **static** **class** SingletonFactory {
7. **private** **static** Singleton instance = **new** Singleton();
8. }
9. /\* 获取实例 \*/
10. **public** **static** Singleton getInstance() {
11. **return** SingletonFactory.instance;
12. }
13. /\* 如果该对象被用于序列化，可以保证对象在序列化前后保持一致 \*/
14. **public** Object readResolve() {
15. **return** getInstance();
16. }
17. }

## Builder建造者

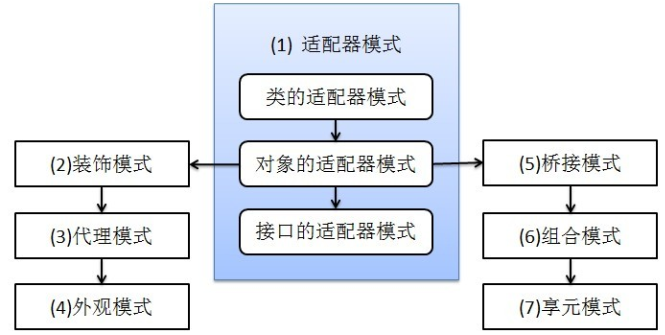
1. **public** interface Packing {
2. **public** String pack();
3. }
5. **public** **class** Wrapper implements Packing {
7. @Override
8. **public** String pack() {
9. **return** "Wrapper";
10. }
11. }
12. **public** **class** Bottle implements Packing {
14. @Override
15. **public** String pack() {
16. **return** "Bottle";
17. }
18. }
20. **public** interface Item {
21. **public** String name();
22. **public** Packing packing();
23. **public** **float** price();
24. }
26. **public** abstract **class** Burger implements Item {
28. @Override
29. **public** Packing packing() {
30. **return** **new** Wrapper();
31. }
33. @Override
34. **public** abstract **float** price();
35. }
36. **public** **class** VegBurger extends Burger {
38. @Override
39. **public** **float** price() {
40. **return** 25.0f;
41. }
43. @Override
44. **public** String name() {
45. **return** "Veg Burger";
46. }
47. }
48. **public** **class** ChickenBurger extends Burger {
50. @Override
51. **public** **float** price() {
52. **return** 50.5f;
53. }
55. @Override
56. **public** String name() {
57. **return** "Chicken Burger";
58. }
59. }
60. **public** abstract **class** ColdDrink implements Item {
62. @Override
63. **public** Packing packing() {
64. **return** **new** Bottle();
65. }
67. @Override
68. **public** abstract **float** price();
69. }
71. **public** **class** Coke extends ColdDrink {
73. @Override
74. **public** **float** price() {
75. **return** 30.0f;
76. }
78. @Override
79. **public** String name() {
80. **return** "Coke";
81. }
82. }
83. **public** **class** Pepsi extends ColdDrink {
85. @Override
86. **public** **float** price() {
87. **return** 35.0f;
88. }
90. @Override
91. **public** String name() {
92. **return** "Pepsi";
93. }
94. }
96. **public** **class** Meal {
97. **private** List<Item> items = **new** ArrayList<Item>();
99. **public** **void** addItem(Item item){
100. items.add(item);
101. }
103. **public** **float** getCost(){
104. **float** cost = 0.0f;
105. **for** (Item item : items) {
106. cost += item.price();
107. }
108. **return** cost;
109. }
111. **public** **void** showItems(){
112. **for** (Item item : items) {
113. System.out.print("Item : "+item.name());
114. System.out.print(", Packing : "+item.packing().pack());
115. System.out.println(", Price : "+item.price());
116. }
117. }
118. }
119. **public** **class** MealBuilder {
121. **public** Meal prepareVegMeal (){
122. Meal meal = **new** Meal();
123. meal.addItem(**new** VegBurger());
124. meal.addItem(**new** Coke());
125. **return** meal;
126. }
128. **public** Meal prepareNonVegMeal (){
129. Meal meal = **new** Meal();
130. meal.addItem(**new** ChickenBurger());
131. meal.addItem(**new** Pepsi());
132. **return** meal;
133. }
134. }
135. **public** **class** BuilderPatternDemo {
136. **public** **static** **void** main(String[] args) {
137. MealBuilder mealBuilder = **new** MealBuilder();
139. Meal vegMeal = mealBuilder.prepareVegMeal();
140. System.out.println("Veg Meal");
141. vegMeal.showItems();
142. System.out.println("Total Cost: " +vegMeal.getCost());
144. Meal nonVegMeal = mealBuilder.prepareNonVegMeal();
145. System.out.println("\n\nNon-Veg Meal");
146. nonVegMeal.showItems();
147. System.out.println("Total Cost: " +nonVegMeal.getCost());
148. }
149. }

## ****Prototype****原型模式

原型模式的目的是以旧对象为原型造出新对象

1. **public** **class** Prototype **implements** Cloneable, Serializable {
3. **private** **static** **final** **long** serialVersionUID = 1L;
4. **private** String string;
6. **private** SerializableObject obj;
8. /\* 浅复制 \*/
9. **public** Object clone() **throws** CloneNotSupportedException {
10. Prototype proto = (Prototype) **super**.clone();
11. **return** proto;
12. }
14. /\* 深复制 \*/
15. **public** Object deepClone() **throws** IOException, ClassNotFoundException {
17. /\* 写入当前对象的二进制流 \*/
18. ByteArrayOutputStream bos = **new** ByteArrayOutputStream();
19. ObjectOutputStream oos = **new** ObjectOutputStream(bos);
20. oos.writeObject(**this**);
22. /\* 读出二进制流产生的新对象 \*/
23. ByteArrayInputStream bis = **new** ByteArrayInputStream(bos.toByteArray());
24. ObjectInputStream ois = **new** ObjectInputStream(bis);
25. **return** ois.readObject();
26. }
28. **public** String getString() {
29. **return** string;
30. }
32. **public** **void** setString(String string) {
33. **this**.string = string;
34. }
36. **public** SerializableObject getObj() {
37. **return** obj;
38. }
40. **public** **void** setObj(SerializableObject obj) {
41. **this**.obj = obj;
42. }
44. }
46. **class** SerializableObject **implements** Serializable {
47. **private** **static** **final** **long** serialVersionUID = 1L;
48. }

# 结构型



从结构上我们可以看出，外观、代理、装饰者是一个分支。享元、组合、桥接是一个分支

## Adapter

要点：适配器的目的是利用现有资源，实现目标接口，使自己称为目标接口的一个实现类。

### 类适配器

利用现有类，通过类继承实现目标接口功能。借用**Source** 类，实现了**Targetable**  功能。

1. public class **Source** {
2. **public** **void** method1() { }
3. }
5. public interface **Targetable** {
6. public void method1();
7. public void method2();
8. }
10. public class Adapter extends Source implements Targetable {
11. @Override
12. public void method2() {
13. System.out.println("this is the targetable method!");
14. }
15. }

### 接口适配器

这是直接用一个抽象类来实现接口，根据自身需要实现相应的方法，不用全部实现。

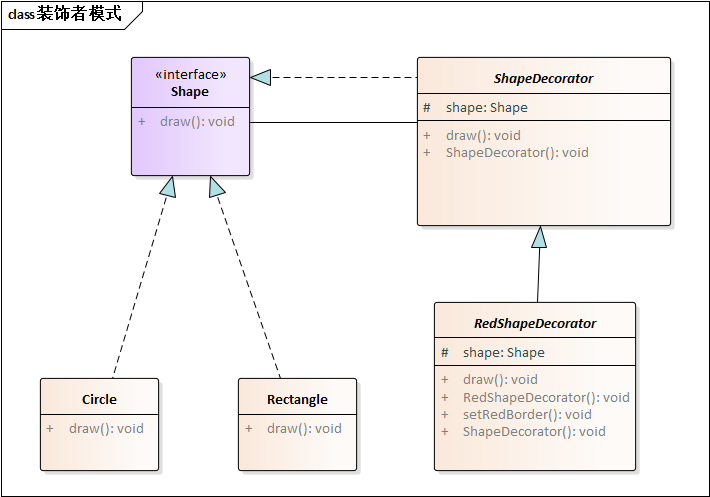
1. **public** **interface** **Sourceable** {
2. **public** **void** method1();
3. **public** **void** method2();
4. }
6. **public** **abstract** **class Wrapper2 implements** **Sourceable**{
7. **public** **void** method1(){}
8. **public** **void** method2(){}
9. }
11. **public** **class** **SourceSub1** **extends** **Wrapper2** {
12. **public** **void** method1(){
13. System.out.println("the Sub1!");
14. }
15. }
17. **public** **class** **SourceSub2** **extends** Wrapper2 {
18. **public** **void** method2(){
19. System.out.println("the Sub2!");
20. }
21. }

### 对象适配器

利用**Source** 对象，通过函数调用实现目标接口功能

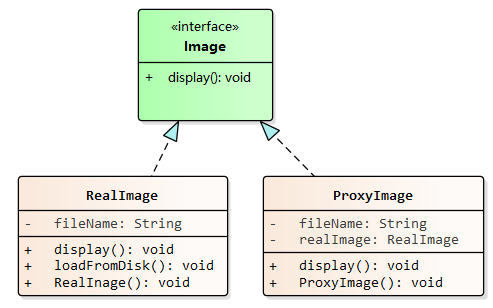
1. **public** **class** **Wrapper** **implements** **Targetable** {
2. **private** **Source** source;
3. **public** Wrapper(Source source){
4. **super**();
5. **this**.source = source;
6. }
8. @Override
9. **public** **void** method2() {
10. System.out.println("this is the targetable method!");
11. }
13. @Override
14. **public** **void** method1() {
15. source.method1();
16. }
17. }

## Decorator Pattern 装饰器模式



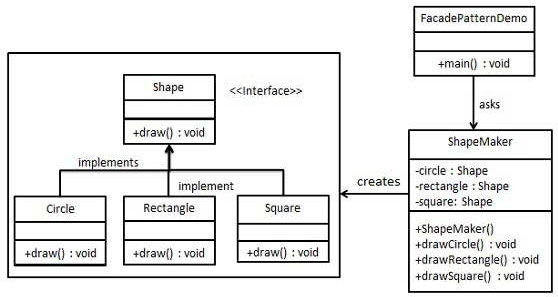
1. **public** **interface** Shape {
2. **void** draw();
3. }
5. // shape 实现类1
6. **public** **class** Rectangle **implements** Shape {
7. @Override
8. **public** **void** draw() {
9. System.out.println("Shape: Rectangle");
10. }
11. }
13. // shape 实现类2
14. **public** **class** Circle **implements** Shape {
15. @Override
16. **public** **void** draw() {
17. System.out.println("Shape: Circle");
18. }
19. }
21. //持有 Shape 实现类对象，并实现 Shape 接口
22. **public** **abstract** **class** ShapeDecorator **implements** Shape {
23. **protected** Shape decoratedShape;
25. **public** ShapeDecorator(Shape decoratedShape){
26. **this**.decoratedShape = decoratedShape;
27. }
29. **public** **void** draw(){
30. decoratedShape.draw();
31. }
32. }
34. //对 Shape 实现类的 draw() 方法进行增强
35. **public** **class** RedShapeDecorator **extends** ShapeDecorator {
36. **public** RedShapeDecorator(Shape decoratedShape) {
37. **super**(decoratedShape);
38. }
40. @Override
41. **public** **void** draw() {
42. decoratedShape.draw();
43. setRedBorder(decoratedShape);
44. }
46. **private** **void** setRedBorder(Shape decoratedShape){
47. System.out.println("Border Color: Red");
48. }
49. }
51. //测试装饰效果
52. **public** **class** DecoratorPatternDemo {
53. **public** **static** **void** main(String[] args) {
55. Shape circle = **new** Circle();
57. Shape redCircle = **new** RedShapeDecorator(**new** Circle());
59. Shape redRectangle = **new** RedShapeDecorator(**new** Rectangle());
60. System.out.println("Circle with normal border");
61. circle.draw();
63. System.out.println("\nCircle of red border");
64. redCircle.draw();
66. System.out.println("\nRectangle of red border");
67. redRectangle.draw();
68. }
69. }
71. 执行程序，输出结果：
73. Circle with normal border
74. Shape: Circle
76. Circle of red border
77. Shape: Circle
78. Border Color: Red
80. Rectangle of red border
81. Shape: Rectangle
82. Border Color: Red

## Proxy Pattern代理模式

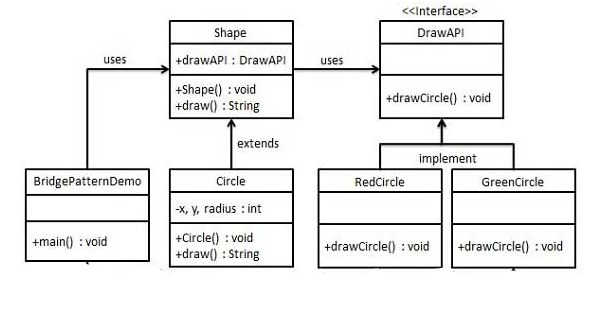


代理模式跟装饰器模式的区别在于，代理模式在编译时就能确定哪个对象要被运行，而装饰器要等到运行时才能确定。

## Facade Pattern外观模式



## Bridge桥接模式



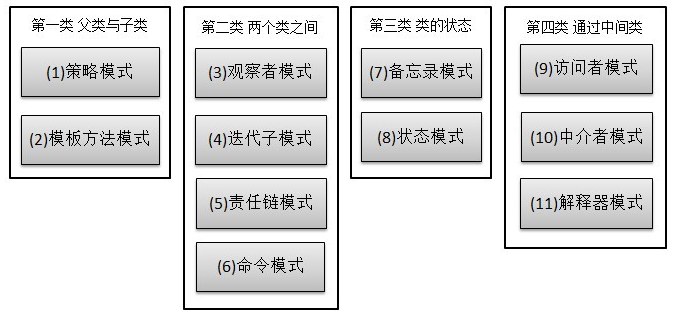
## Composite 组合模式

含有列表

## Flyweight 享元模式

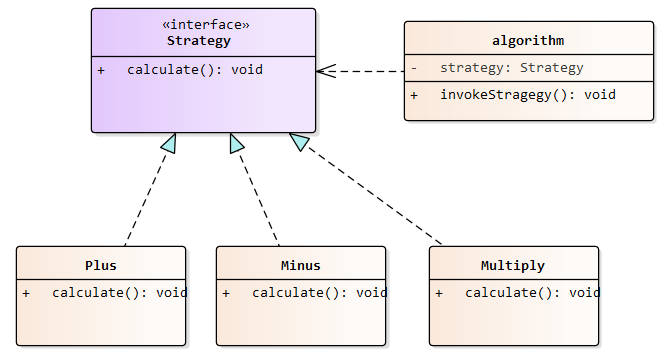
池化技术

# 行为型



## Strategy策略模式

算法根据传进来的策略，结果也不同



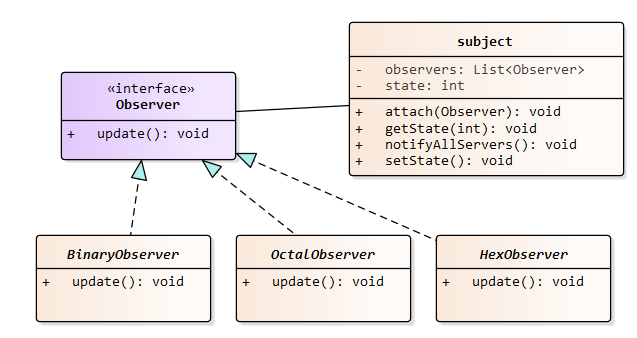
## Template method模板方法

策略模式，模板方法都差不多。

1. **public** **abstract** **class** **AbstractCalculator** {
2. /\*主方法，实现对本类其它方法的调用\*/
3. **public** **final** **void** calculate(**int** n){
4. **int** result = 0;
6. **for**(**int** j =0; j <= n; j++)
7. {
8. result += j;
9. }
11. printSum(result);
12. }
14. /\*被子类重写的方法\*/
15. **abstract** **public** **int** printSum(**int** sum);
16. }
18. **public** **class** **Plus** **extends** **AbstractCalculator** {
20. @Override
21. **public** **int** printSum(**int** sum) {
22. **return** System.out.println(sum);
23. }
24. }

模板方法跟继承的区别在于，我们要一次性实现不变的东西，把变的东西延迟实现。

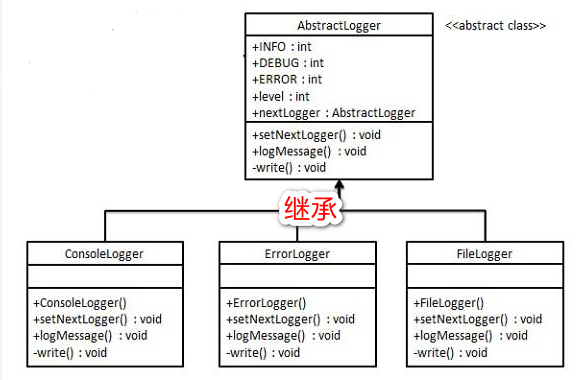
## 观察者



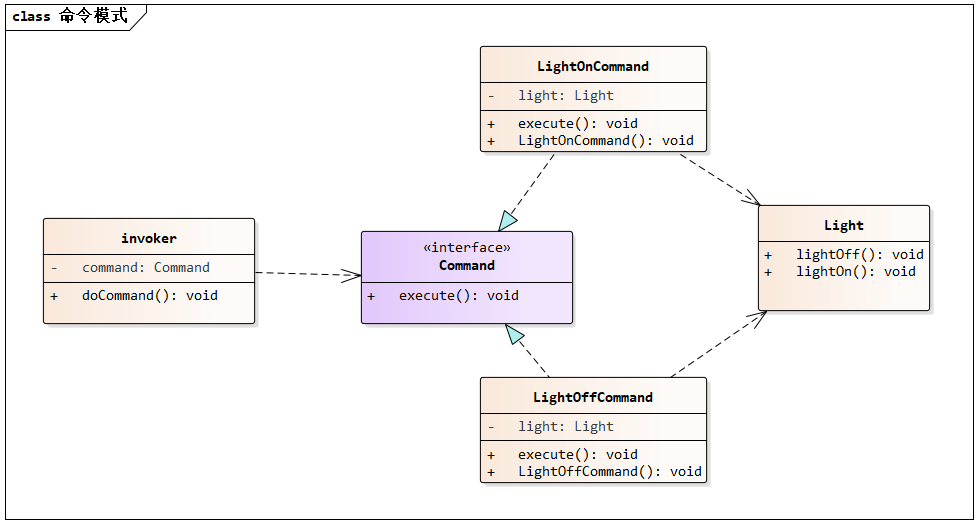
先注册到 subject 种，然后再通知

## 迭代器模式Iterator Pattern

## 责任链模式



## 命令模式



**public** **class** Light {

**public** **void** lightOn() {

        System.out.println("灯打开了！！");

    }

**public** **void** lightOff() {

        System.out.println("灯关上了！！");

    }

}

**public** **interface** Command {

**void** execute();

}

**public** **class** LightOnCommand **implements** Command{

**private** Light light;

**public** LightOnCommand(Light light) {

**this**.light = light;

    }

**public** **void** execute() {

        light.lightOn();

    }

}

**public** **class** LightOffCommand **implements** Command{

**private** Light light;

**public** LightOffCommand(Light light) {

**this**.light = light;

    }

**public** **void** execute() {

        light.lightOff();

    }

}

**public** **class** XiaoAi {

**private** Command command;

**public** **void** setCommand(Command command) {

**this**.command = command;

    }

**public** **void** doCommand() {

        command.execute();

    }

}

**public** **class** Client {

**public** **static** **void** main(String[] args) {

        XiaoAi xiaoAi = **new** XiaoAi();

        Light light = **new** Light();

        System.out.println("小爱同学帮我把灯开一下！");

        LightOnCommand lightOnCommand = **new** LightOnCommand(light);

        xiaoAi.setCommand(lightOnCommand);

        xiaoAi.doCommand();

        System.out.println("-------");

        System.out.println("小爱同学帮我关一下灯！");

        LightOffCommand lightOffCommand = **new** LightOffCommand(light);

        xiaoAi.setCommand(lightOffCommand);

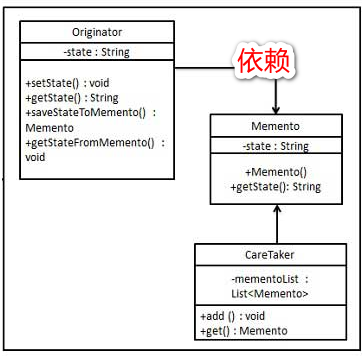
        xiaoAi.doCommand();

    }

}

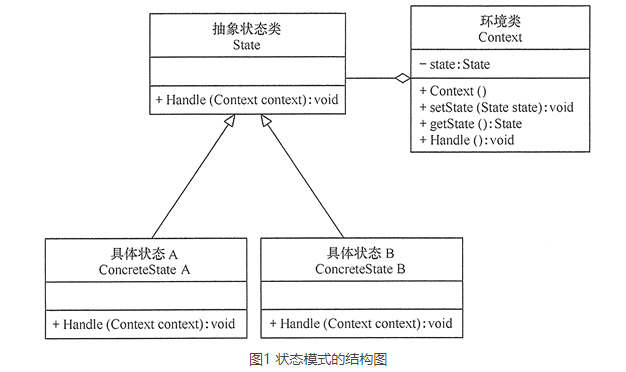
LightOnCommand、LightOffCommand是两个不同的命令对象，分别对应了 Light 两个方法的实现，这里等于是把方法转化成了对象。我觉得这里如果函数可以作为一等公民，直接传过来就行了，说明这里是语言的缺陷。

## 备忘录模式



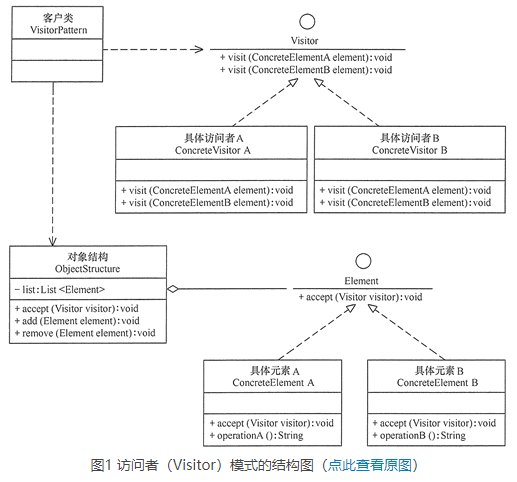
## 状态模式

**C**ontext环境对象持有状态对象，状态变化，持有的具体状态对象也会发生变化，对应有不同的动作。



## 访问者

访问者的要点在于，访问者要重载以应对不同的对象。对象接收访问的时候，直接暴漏给访问者。这个是重载的加强版。



//访问者可以处理多个对象

interface Visitor

{

void visit(ConcreteElementA element);

void visit(ConcreteElementB element);

}

class ConcreteVisitorA implements Visitor

{

public void visit(ConcreteElementA element)

public void visit(ConcreteElementB element)

}

class ConcreteVisitorB implements Visitor

{

public void visit(ConcreteElementA element)

public void visit(ConcreteElementB element)

}

interface Element

{

void accept(Visitor visitor);

}

//把自己暴露给访问者，不合适

class ConcreteElementA implements Element

{

public void accept(Visitor visitor)

{

visitor.visit(this);

}

}

//把自己暴露给访问者，不合适

class ConcreteElementB implements Element

{

public void accept(Visitor visitor)

{

visitor.visit(this);

}

}

class ObjectStructure

{

private List<Element> list=new ArrayList<Element>();

public void accept(Visitor visitor)

{

Iterator<Element> i=list.iterator();

while(i.hasNext())

{

((Element) i.next()).accept(visitor);

}

}

public void add(Element element)

{

list.add(element);

}

public void remove(Element element)

{

list.remove(element);

}

}

## 中介者

中介者，把自己暴漏给中介，交流通过中介来实现。适合规定动作的对象。比如途中收、发规定动作。把自己暴漏给中介后，中介替你收、发消息。

