1. **Factory工厂模式：**

接口：

Interface TraceFactory{

Public Trace getTrace(String type);

Void otherOperation;

}

实现：

Public class Factory implements TraceFactory{

Public Trace getTrace(String type){

If(type.equals(“file”))

Return new FileTrace();

Else if(type.equals(“system”))

Return new SystemTrace();

}

}

客户端：

Trace log=new Factory().getTrace(“system”);

**静态工厂方法**

public class TraceFactory2 {

public static Trace getTrace(String type) {

if(type.equals(“file”)

return new FileTrace();

else if (type.equals(“system”)

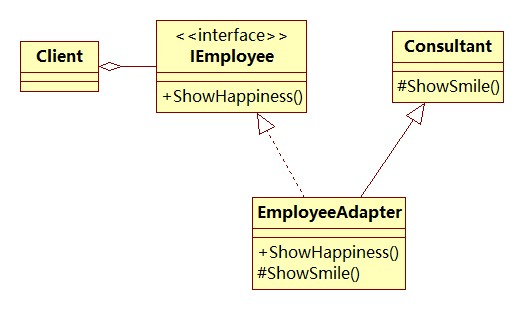
return new SystemTrace();

}

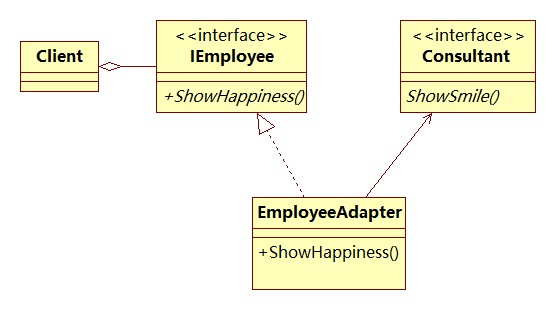
}

1. **Adapter适配器模式**

**继承：**



**委派：**



1. **Decorate装饰模式**

**模式的结构**

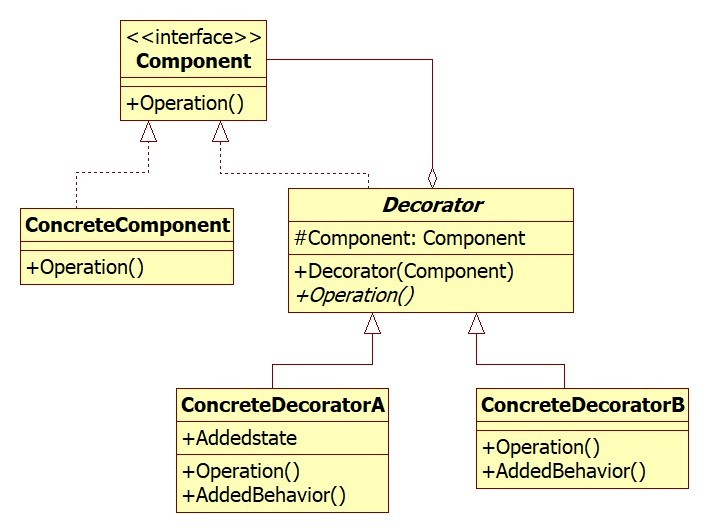
•抽象构件（Component）角色：定义一个抽象接口以规范准备接收附加责任的对象。

•具体构件（Concrete Component）角色：实现抽象构件，通过装饰角色为其添加一些职责。

•抽象装饰（Decorator）角色：继承抽象构件，并包含具体构件的实例，可以通过其子类扩展具体构件的功能。

•具体装饰（ConcreteDecorator）角色：实现抽象装饰的相关方法，并给具体构件对象添加附加的责任

实现一个通用接口作为要扩展的对象，将主要功能委托给基础对象(stack)，然后添加功能(undo,secure,..)责任

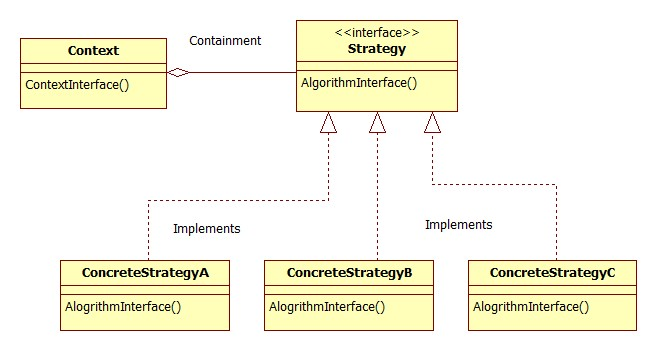


1. **Strategy策略模式**

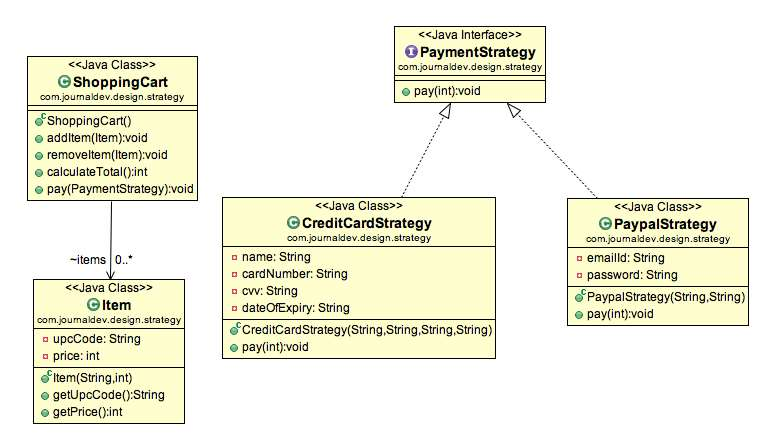
针对特定任务存在多种算法，调用者需要根据上下文环境动态的选择和切换。

定义一个算法的接口，每个算法用一个类来实现，客户端针对接口编写程序。

**UML：**



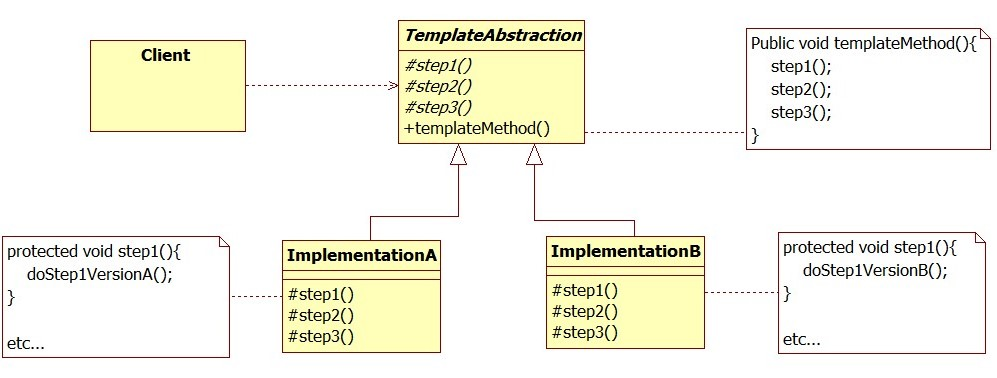
**实例：**



1. **Template模板模式**

不同的客户端具有相同的算法步骤，但是每个步骤的具体实现不同。

在父类中定义通用逻辑和各步骤的抽象方法声明，子类中进行各步骤的具体实现



**模板模式用继承+重写的方式实现算法的不同部分。**

**策略模式用委托机制实现不同完整算法的调用(接口+多态)**

1. **Iterator迭代器模式**

public class Pair<E> implements Iterable<E> {

private final E first, second;

public Pair(E f, E s) { first = f; second = s; }

public Iterator<E> iterator() {

return new PairIterator();

}

private class PairIterator implements Iterator<E> {

private boolean seenFirst = false, seenSecond = false;

public boolean hasNext() { return !seenSecond; }

public E next() {

if (!seenFirst) { seenFirst = true; return first; }

if (!seenSecond) { seenSecond = true; return second; }

throw new NoSuchElementException();

}

public void remove() {

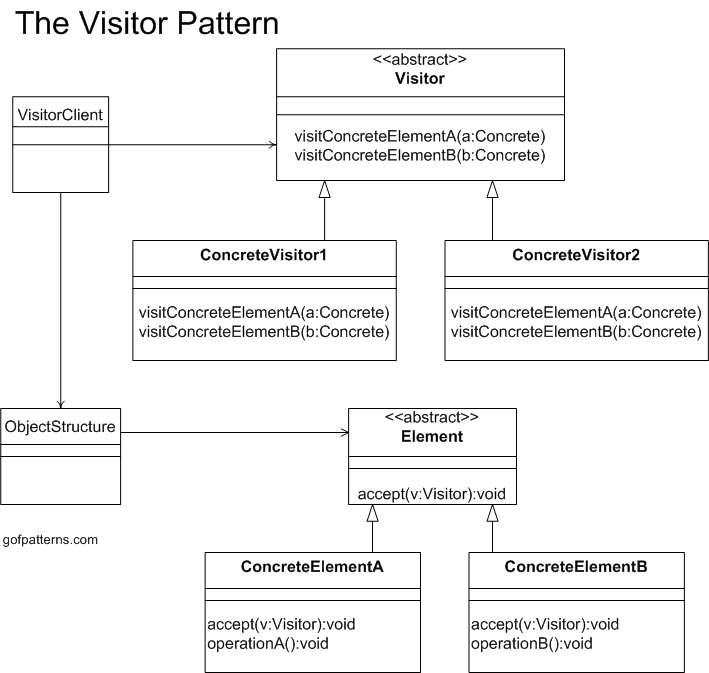
throw new UnsupportedOperationException();

}

}

}

1. **Visitor访问者模式**



public class Book implements ItemElement{

private double price;

...

int accept(ShoppingCartVisitor visitor) {

visitor.visit(this);

}

}

/\* Abstract visitor interface \*/

public interface ShoppingCartVisitor {

int visit(Book book);

int visit(Fruit fruit);

}

public class ShoppingCartVisitorImpl implements ShoppingCartVisitor {

public int visit(Book book) {

int cost=0;

if(book.getPrice() > 50){

cost = book.getPrice()-5;

}else

cost = book.getPrice();

System.out.println("Book ISBN::"+book.getIsbnNumber() + " cost ="+cost);

return cost;

}

public int visit(Fruit fruit) {

int cost = fruit.getPricePerKg()\*fruit.getWeight();

System.out.println(fruit.getName() + " cost = "+cost);

return cost;

}

}