## ****抽象工厂模式（Abstract Factory Pattern）****

提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。在一个工厂里聚合多个同类产品。

### 优点：

当一个产品族中的多个对象被设计成一起工作时，它能保证客户端始终只使用同一个产品族中的对象。

### 缺点：

产品族扩展非常困难，要增加一个系列的某一产品，既要在抽象的 Creator 里加代码，又要在具体的里面加代码。

### 示例：

public interface Shape {

void draw();

}

public class Rectangle implements Shape {

@Override

public void draw() {

System.out.println("Inside Rectangle::draw() method.");

}

}

public class Square implements Shape {

@Override

public void draw() {

System.out.println("Inside Square::draw() method.");

}

}

public class Circle implements Shape {

@Override

public void draw() {

System.out.println("Inside Circle::draw() method.");

}

}

public interface Color {

void fill();

}

public class Red implements Color {

@Override

public void fill() {

System.out.println("Inside Red::fill() method.");

}

}

public class Green implements Color {

@Override

public void fill() {

System.out.println("Inside Green::fill() method.");

}

}

public abstract class AbstractFactory {

public abstract Color getColor(String color);

public abstract Shape getShape(String shape) ;

}

public class ShapeFactory extends AbstractFactory {

@Override

public Shape getShape(String shapeType){

if(shapeType == null){

return null;

}

if(shapeType.equalsIgnoreCase("CIRCLE")){

return new Circle();

} else if(shapeType.equalsIgnoreCase("RECTANGLE")){

return new Rectangle();

} else if(shapeType.equalsIgnoreCase("SQUARE")){

return new Square();

}

return null;

}

@Override

public Color getColor(String color) {

return null;

}

}

public class ColorFactory extends AbstractFactory {

@Override

public Shape getShape(String shapeType){

return null;

}

@Override

public Color getColor(String color) {

if(color == null){

return null;

}

if(color.equalsIgnoreCase("RED")){

return new Red();

} else if(color.equalsIgnoreCase("GREEN")){

return new Green();

}

return null;

}

//创建一个工厂创造器/生成器类，通过传递形状或颜色信息来获取工厂。

public class FactoryProducer {

public static AbstractFactory getFactory(String choice){

if(choice.equalsIgnoreCase("SHAPE")){

return new ShapeFactory();

} else if(choice.equalsIgnoreCase("COLOR")){

return new ColorFactory();

}

return null;

}

}

public class AbstractFactoryPatternDemo {

public static void main(String[] args) {

//获取形状工厂

AbstractFactory shapeFactory = FactoryProducer.getFactory("SHAPE");

//获取形状为 Circle 的对象

Shape shape1 = shapeFactory.getShape("CIRCLE");

//调用 Circle 的 draw 方法

shape1.draw();

//获取形状为 Rectangle 的对象

Shape shape2 = shapeFactory.getShape("RECTANGLE");

//调用 Rectangle 的 draw 方法

shape2.draw();

//获取形状为 Square 的对象

Shape shape3 = shapeFactory.getShape("SQUARE");

//调用 Square 的 draw 方法

shape3.draw();

//获取颜色工厂

AbstractFactory colorFactory = FactoryProducer.getFactory("COLOR");

//获取颜色为 Red 的对象

Color color1 = colorFactory.getColor("RED");

//调用 Red 的 fill 方法

color1.fill();

//获取颜色为 Green 的对象

Color color2 = colorFactory.getColor("Green");

//调用 Green 的 fill 方法

color2.fill();

}

}

## ****适配器模式（Adapter Pattern）****

将一个类的接口转换成客户希望的另外一个接口。适配器模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。

主要解决在软件系统中，常常要将一些"现存的对象"放到新的环境中，而新环境要求的接口是现对象不能满足的。

1. 系统需要使用现有的类，而此类的接口不符合系统的需要。
2. 想要建立一个可以重复使用的类，用于与一些彼此之间没有太大关联的一些类，包括一些可能在将来引进的类一起工作，这些源类不一定有一致的接口。
3. 通过接口转换，将一个类插入另一个类系中。（比如老虎和飞禽，现在多了一个飞虎，在不增加实体的需求下，增加一个适配器，在里面包容一个虎对象，实现飞的接口。）

### 优点：

* 1. 可以让任何两个没有关联的类一起运行。
  2. 提高了类的复用。
  3. 增加了类的透明度。
  4. 灵活性好。

### 缺点：

1. 过多地使用适配器，会让系统非常零乱，不易整体进行把握。比如，明明看到调用的是 A 接口，其实内部被适配成了 B 接口的实现，一个系统如果太多出现这种情况，无异于一场灾难。因此如果不是很有必要，可以不使用适配器，而是直接对系统进行重构。
2. 由于 JAVA 至多继承一个类，所以至多只能适配一个适配者类，而且目标类必须是抽象类。

### 示例：

public class MediaAdapter implements MediaPlayer {

AdvancedMediaPlayer advancedMusicPlayer;

public MediaAdapter(String audioType){

if(audioType.equalsIgnoreCase("vlc") ){

advancedMusicPlayer = new VlcPlayer();

} else if (audioType.equalsIgnoreCase("mp4")){

advancedMusicPlayer = new Mp4Player();

}

}

@Override

public void play(String audioType, String fileName) {

if(audioType.equalsIgnoreCase("vlc")){

advancedMusicPlayer.playVlc(fileName);

}else if(audioType.equalsIgnoreCase("mp4")){

advancedMusicPlayer.playMp4(fileName);

}

}

}

## ****装饰器模式（Decorator Pattern）****

动态地给一个对象添加一些额外的职责。就增加功能来说，装饰器模式相比生成子类更为灵活。为了扩展一个类经常使用继承方式实现，由于继承为类引入静态特征，并且随着扩展功能的增多，子类会很膨胀。

### 优点：

装饰类和被装饰类可以独立发展，不会相互耦合，装饰模式是继承的一个替代模式，装饰模式可以动态扩展一个实现类的功能。

### 缺点：

多层装饰比较复杂。

### 作用:

1、扩展一个类的功能。

2、动态增加功能，动态撤销。

### 示例：

public interface Shape {

void draw();

}

public class Rectangle implements Shape {

@Override

public void draw() {

System.out.println("Shape: Rectangle");

}

}

public abstract class ShapeDecorator implements Shape {

protected Shape decoratedShape;

public ShapeDecorator(Shape decoratedShape){

this.decoratedShape = decoratedShape;

}

public void draw(){

decoratedShape.draw();

}

}

public class RedShapeDecorator extends ShapeDecorator {

public RedShapeDecorator(Shape decoratedShape) {

super(decoratedShape);

}

@Override

public void draw() {

decoratedShape.draw();

setRedBorder(decoratedShape);

}

private void setRedBorder(Shape decoratedShape){

System.out.println("Border Color: Red");

}

}

## 代理模式（Proxy Pattern）

为其他对象提供一种代理以控制对这个对象的访问。

在直接访问对象时带来的问题，比如说：要访问的对象在远程的机器上。在面向对象系统中，有些对象由于某些原因，直接访问会给使用者或者系统结构带来很多麻烦，我们可以在访问此对象时加上一个对此对象的访问层。

1、和适配器模式的区别：适配器模式主要改变所考虑对象的接口，而代理模式不能改变所代理类的接口。

2、和装饰器模式的区别：装饰器模式为了增强功能，而代理模式是为了加以控制。

### 示例：

public class RealImage implements Image {

private String fileName;

public RealImage(String fileName){

this.fileName = fileName;

loadFromDisk(fileName);

}

@Override

public void display() {

System.out.println("Displaying " + fileName);

}

private void loadFromDisk(String fileName){

System.out.println("Loading " + fileName);

}

}

public class ProxyImage implements Image{

private RealImage realImage;

private String fileName;

public ProxyImage(String fileName){

this.fileName = fileName;

}

@Override

public void display() {

if(realImage == null){

realImage = new RealImage(fileName);

}

realImage.display();

}

}

## 迭代器模式（Iterator Pattern）

提供一种方法顺序访问一个聚合对象中各个元素, 而又无须暴露该对象的内部表示。

* + 1. 它支持以不同的方式遍历一个聚合对象。
    2. 迭代器简化了聚合类。
    3. 在同一个聚合上可以有多个遍历。
    4. 在迭代器模式中，增加新的聚合类和迭代器类都很方便，无须修改原有代码。
    5. 访问一个聚合对象的内容而无须暴露它的内部表示。
    6. 需要为聚合对象提供多种遍历方式。
    7. 为遍历不同的聚合结构提供一个统一的接口。

### 示例：

public interface Iterator {

public boolean hasNext();

public Object next();

}

public interface Container {

public Iterator getIterator();

}

public class NameRepository implements Container {

public String names[] = {"Robert" , "John" ,"Julie" , "Lora"};

@Override

public Iterator getIterator() {

return new NameIterator();

}

private class NameIterator implements Iterator {

int index;

@Override

public boolean hasNext() {

if(index < names.length){

return true;

}

return false;

}

@Override

public Object next() {

if(this.hasNext()){

return names[index++];

}

return null;

}

}

}

## 策略模式（Strategy Pattern）

一个类的行为或其算法可以在运行时更改。这种类型的设计模式属于行为型模式。

定义一系列的算法,把它们一个个封装起来, 并且使它们可相互替换。一个系统有许多许多类，而区分它们的只是他们直接的行为。将这些算法封装成一个一个的类，任意地替换。

### 优点：

1、算法可以自由切换。

2、避免使用多重条件判断。

3、扩展性良好。

### 示例：

public interface Strategy {

public int doOperation(int num1, int num2);

}

public class OperationAdd implements Strategy{

@Override

public int doOperation(int num1, int num2) {

return num1 + num2;

}

}

public class OperationSubtract implements Strategy{

@Override

public int doOperation(int num1, int num2) {

return num1 - num2;

}

}

public class Context {

private Strategy strategy;

public Context(Strategy strategy){

this.strategy = strategy;

}

public int executeStrategy(int num1, int num2){

return strategy.doOperation(num1, num2);

}

}

## 访问者模式（Visitor Pattern）

将数据结构与数据操作分离。需要对一个对象结构中的对象进行很多不同的并且不相关的操作，而需要避免让这些操作"污染"这些对象的类，使用访问者模式将这些封装到类中。

### 优点：

1、符合单一职责原则。

2、优秀的扩展性。

3、灵活性。

### 缺点：

1、对象结构中对象对应的类很少改变，但经常需要在此对象结构上定义新的操作。

2、需要对一个对象结构中的对象进行很多不同的并且不相关的操作，而需要避免让这些操作"污染"这些对象的类，也不希望在增加新操作时修改这些类。

注意事项：访问者可以对功能进行统一，可以做报表、UI、拦截器与过滤器。

### 示例：

public interface ComputerPart {

public void accept(ComputerPartVisitor computerPartVisitor);

}

public class Keyboard implements ComputerPart {

@Override

public void accept(ComputerPartVisitor computerPartVisitor) {

computerPartVisitor.visit(this);

}

}

public class Monitor implements ComputerPart {

@Override

public void accept(ComputerPartVisitor computerPartVisitor) {

computerPartVisitor.visit(this);

}

}

public class Computer implements ComputerPart {

ComputerPart[] parts;

public Computer(){

parts = new ComputerPart[] {new Mouse(), new Keyboard(), new Monitor()};

}

public class Mouse implements ComputerPart {

@Override

public void accept(ComputerPartVisitor computerPartVisitor) {

computerPartVisitor.visit(this);

}

}

@Override

public void accept(ComputerPartVisitor computerPartVisitor) {

for (int i = 0; i < parts.length; i++) {

parts[i].accept(computerPartVisitor);

}

computerPartVisitor.visit(this);

}

}

public interface ComputerPartVisitor {

public void visit(Computer computer);

public void visit(Mouse mouse);

public void visit(Keyboard keyboard);

public void visit(Monitor monitor);

}

public class ComputerPartDisplayVisitor implements ComputerPartVisitor {

@Override

public void visit(Computer computer) {

System.out.println("Displaying Computer.");

}

@Override

public void visit(Mouse mouse) {

System.out.println("Displaying Mouse.");

}

@Override

public void visit(Keyboard keyboard) {

System.out.println("Displaying Keyboard.");

}

@Override

public void visit(Monitor monitor) {

System.out.println("Displaying Monitor.");

}

}

public class VisitorPatternDemo {

public static void main(String[] args) {

ComputerPart computer = new Computer();

computer.accept(new ComputerPartDisplayVisitor());

}

}