结构型模式
适配器模式（Adapter Pattern）
意图
主要解决
何时使用
如何解决
关键代码
应用实例
优点
缺点
使用场景
注意事项
示例实现
桥接模式（Bridge）
意图
主要解决
何时使用
如何解决
关键代码
应用实例
优点
缺点
使用场景
注意事项
示例实现
过滤器模式（Filter Pattern）
示例实现
组合模式（Composite Pattern）
意图
主要解决
何时使用
如何解决
关键代码
应用实例
优点
缺点
使用场景
注意事项
示例实现

- 装饰器模式（Decorator Pattern）
  - 意图
  - 主要解决
  - 何时使用
  - 如何解决
  - 关键代码
  - 应用实例
  - 优点
  - 缺点
  - 使用场景
  - 注意事项
  - 示例实现
- 外观模式（Facade Pattern）
  - 意图
  - 主要解决
  - 何时使用
  - 如何解决
  - 关键代码
  - 应用实例
  - 优点
  - 缺点
  - 使用场景
  - 注意事项
  - 示例实现
- 享元模式（Flyweight Pattern）
  - 意图
  - 主要解决
  - 何时使用
  - 如何解决
  - 关键代码
  - 应用实例
  - 优点
  - 缺点
  - 使用场景
  - 注意事项
  - 示例实现
- 代理模式（Proxy）
  - 意图
  - 主要解决
  - 何时使用
  - 如何解决

关键代码
应用实例
优点
缺点
使用场景
注意事项
相关模式
示例实现

结构型模式

适配器模式（Adapter Pattern）

适配器模式是作为两个不兼容的接口之间的桥梁。这种类型的设计模式属于结构型模式，它结合了两个独立接口的功能。
这种模式涉及到一个单一的类，该类负责加入独立的或不兼容的接口功能。举个真实的例子，读卡器是作为内存卡和笔记本之间的适配器。您将内存卡插入读卡器，再将读卡器插入笔记本，这样就可以通过笔记本来读取内存卡。
我们通过下面的实例来演示适配器模式的使用。其中，音频播放器设备只能播放 mp3 文件，通过使用一个更高级的音频播放器来播放 vlc 和 mp4 文件。

意图

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

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。

何时使用

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。）

如何解决

继承或依赖（推荐）。

关键代码

适配器继承或依赖已有的对象，实现想要的目标接口。

应用实例

1、美国电器 110V，中国 220V，就要有一个适配器将 110V 转化为 220V。
2、JAVA JDK 1.1 提供了 Enumeration 接口，而在 1.2 中提供了 Iterator 接口，想要使用 1.2 的 JDK，则要将以前系统的 Enumeration 接口转化为 Iterator 接口，这时就需要适配器模式。
3、在 LINUX 上运行 WINDOWS 程序。
4、JAVA 中的 jdbc。

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。

使用场景

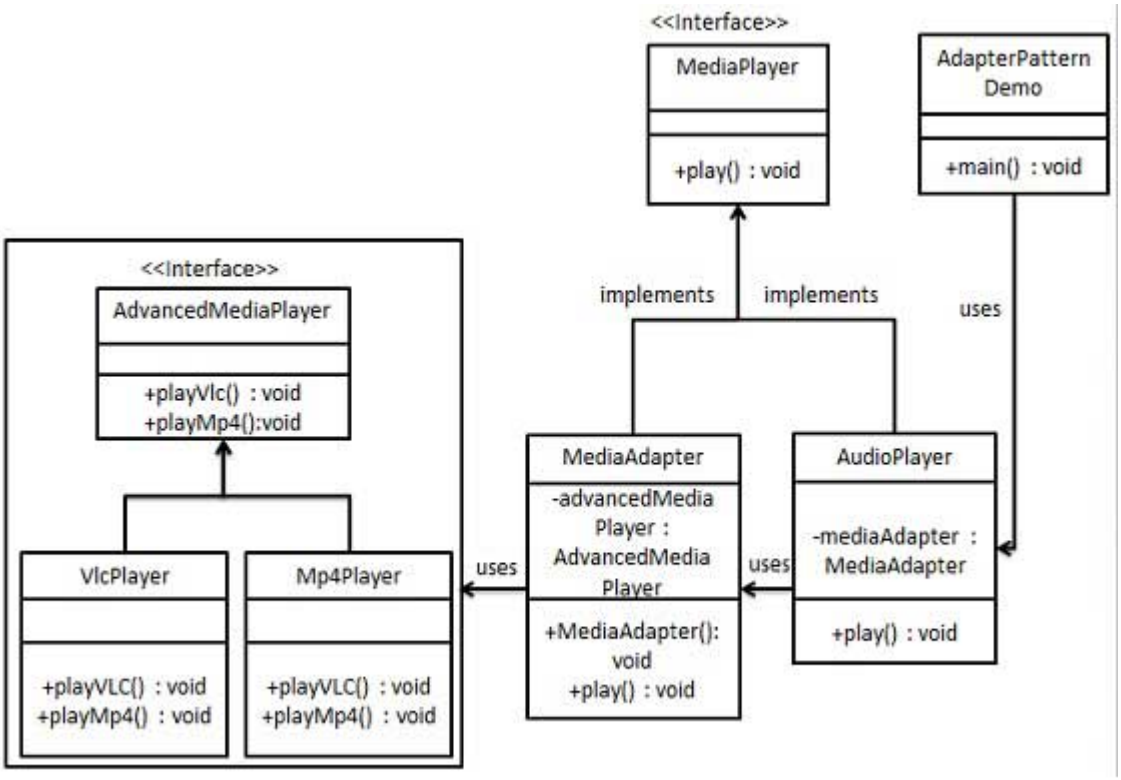
有动机地修改一个正常运行的系统的接口，这时应该考虑使用适配器模式。

注意事项

适配器不是在详细设计时添加的，而是解决正在服役的项目的问题。

示例实现

我们有一个 MediaPlayer 接口和一个实现了 MediaPlayer 接口的实体类 AudioPlayer。默认情况下，AudioPlayer 可以播放 mp3 格式的音频文件。
我们还有另一个接口 AdvancedMediaPlayer 和实现了 AdvancedMediaPlayer 接口的实体类。该类可以播放 vlc 和 mp4 格式的文件。
我们想要让 AudioPlayer 播放其他格式的音频文件。为了实现这个功能，我们需要创建一个实现了 MediaPlayer 接口的适配器类 MediaAdapter，并使用 AdvancedMediaPlayer 对象来播放所需的格式。
AudioPlayer 使用适配器类 MediaAdapter 传递所需的音频类型，不需要知道能播放所需格式音频的实际类。
AdapterPatternDemo，我们的演示类使用 AudioPlayer 类来播放各种格式。



MediaAdapter是一个适配器，AudioPlayer需要通过MediaAdapter来间接操作AdvancedMediaPlayer进行VLC或Mp4的播放，注意它们都实现了MediaPlayer接口，这意味着对客户端来说操作MediaAdapter和AudioPlayer没有什么区别。

```
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);
        }
    }
}
```

```
public class AudioPlayer implements MediaPlayer {
    MediaAdapter mediaAdapter;
    @Override
```

```
    public void play(String audioType, String fileName) {
        if(audioType.equalsIgnoreCase("mp3")) {
            System.out.println("Playing mp3 file. Name: " + fileName);
        } else if(audioType.equalsIgnoreCase("vlc")
                || audioType.equalsIgnoreCase("mp4")) {
            mediaAdapter = new MediaAdapter(audioType);
            mediaAdapter.play(audioType, fileName);
        } else {
            System.out.println("Invalid media. " + audioType + " format not
supported");
        }
    }
}
```

桥接模式（Bridge）

桥接是用于把抽象化与实现化解耦，使得二者可以独立变化。这种类型的设计模式属于结构型模式，它通过提供抽象化和实现化之间的桥接结构，来实现二者的解耦。
这种模式涉及到一个作为桥接的接口，使得实体类的功能独立于接口实现类。这两种类型的类可被结构化改变而互不影响。
我们通过下面的实例来演示桥接模式（Bridge Pattern）的用法。其中，可以使用相同的抽象类方法但是不同的桥接实现类，来画出不同颜色的圆。

意图

将抽象部分与实现部分分离，使它们都可以独立的变化。

主要解决

在有多种可能会变化的情况下，用继承会造成类爆炸问题，扩展起来不灵活。

何时使用

实现系统可能有多个角度分类，每一种角度都可能变化。

如何解决

把这种多角度分类分离出来，让它们独立变化，减少它们之间耦合。

关键代码

抽象类依赖实现类。

应用实例

1、猪八戒从天蓬元帅转世投胎到猪，转世投胎的机制将尘世划分为两个等级，即：灵魂和肉体，前者相当于抽象化，后者相当于实现化。生灵通过功能的委派，调用肉体对象的功能，使得生灵可以动态地选择。

2、墙上的开关，可以看到的开关是抽象的，不用管里面具体怎么实现的。

优点

1、抽象和实现的分离。
2、优秀的扩展能力。 3、实现细节对客户透明。

缺点

桥接模式的引入会增加系统的理解与设计难度，由于聚合关联关系建立在抽象层，要求开发者针对抽象进行设计与编程。

使用场景

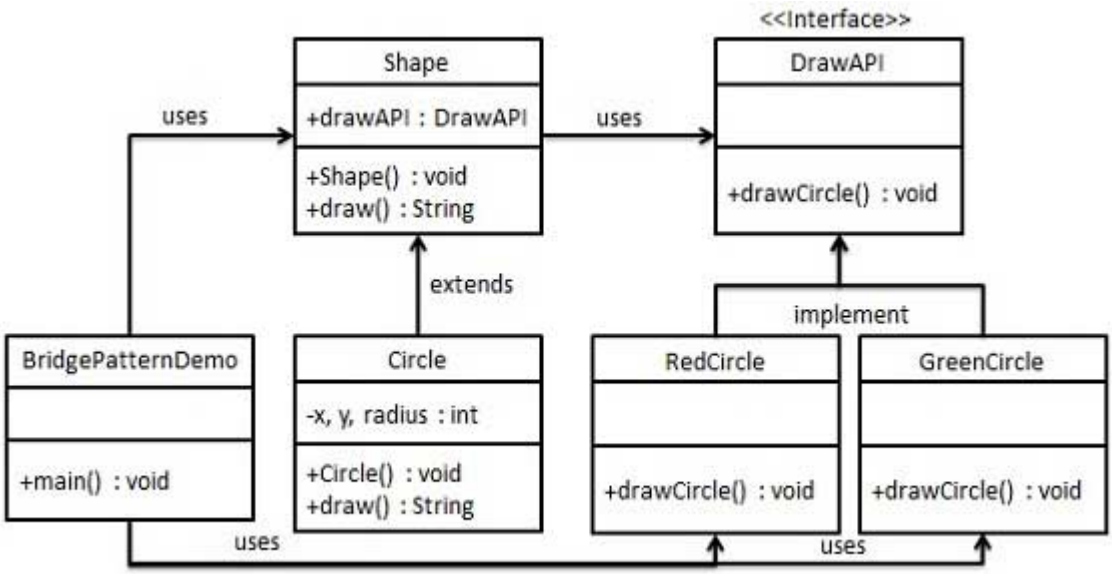
1、如果一个系统需要在构件的抽象化角色和具体化角色之间增加更多的灵活性，避免在两个层次之间建立静态的继承联系，通过桥接模式可以使它们在抽象层建立一个关联关系。
2、对于那些不希望使用继承或因为多层次继承导致系统类的个数急剧增加的系统，桥接模式尤为适用。
3、一个类存在两个独立变化的维度，且这两个维度都需要进行扩展。

注意事项

对于两个独立变化的维度，使用桥接模式再适合不过了。

示例实现

我们有一个作为桥接实现的 DrawAPI 接口和实现了 DrawAPI 接口的实体类 RedCircle、GreenCircle。
Shape 是一个抽象类，将使用 DrawAPI 的对象。
我们的演示类使用 Shape 类来画出不同颜色的圆。



Shape定义了图形的参数，需要利用DrawAPI来进行图形的绘制：

```
public abstract class Shape {
    protected DrawAPI drawAPI;
    protected Shape(DrawAPI drawAPI) {
        this.drawAPI = drawAPI;
```

```
    }
    public abstract void draw();
}
public class Circle extends Shape {
    private int x, y, radius;
    public Circle(int x, int y, int radius, DrawAPI drawAPI) {
        super(drawAPI);
        this.x = x;
        this.y = y;
        this.radius = radius;
    }
    public void draw() {
        drawAPI.drawCircle(radius, x, y);
    }
}
```

DrawAPI负责图形的实际绘制：

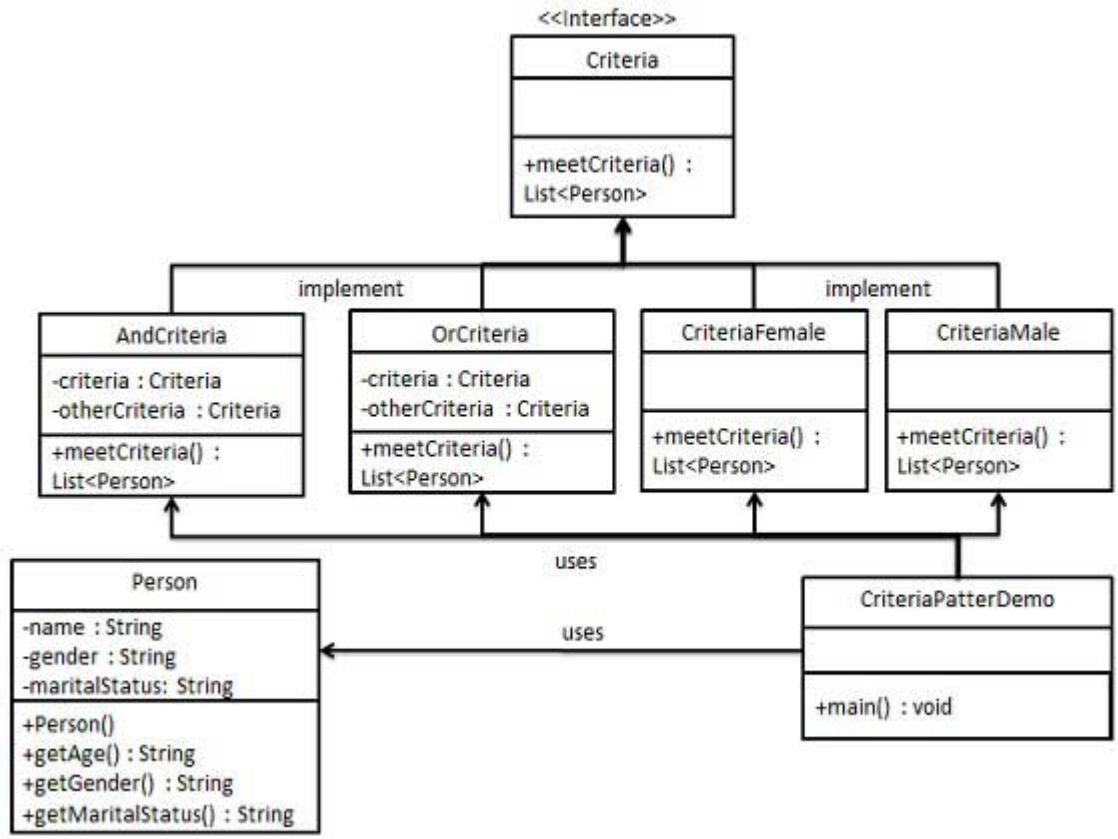
```
public interface DrawAPI {
    public void drawCircle(int radius, int x, int y);
}
public class RedCircle implements DrawAPI {
    @Override
    public void drawCircle(int radius, int x, int y) {
        // >> 画红圈代码
    }
}
```

过滤器模式（Filter Pattern）

过滤器模式或标准模式（Criteria Pattern）是一种设计模式，这种模式允许开发人员使用不同的标准来过滤一组对象，通过逻辑运算以解耦的方式把它们连接起来。
这种类型的设计模式属于 结构型模式，它结合多个标准来获得单一标准。

示例实现

我们将创建一个 Person 对象、Criteria 接口和实现了该接口的实体类，来过滤 Person 对象的列表。
我们的演示类使用 Criteria 对象，基于各种标准和它们的结合来过滤 Person 对象的列表。



下面以或逻辑的过滤实现类OrCriteria为例：

```
public interface Criteria {
    public List<Person> meetCriteria(List<Person> persons);
}
public class OrCriteria implements Criteria {
    private Criteria criteria;
    private Criteria otherCriteria;
    public OrCriteria(Criteria criteria, Criteria otherCriteria) {
        this.criteria = criteria;
        this.otherCriteria = otherCriteria;
    }
    @Override
    public List<Person> meetCriteria(List<Person> persons) {
        List<Person> firstCriteriaItems = criteria.meetCriteria(persons);
        List<Person> otherCriteriaItems = otherCriteria.meetCriteria(persons);
        for(Person person: otherCriteriaItems) {
            if(!firstCriteriaItems.contains(person)) {
                firstCriteriaItems.add(person);
            }
        }
        return firstCriteriaItems;
    }
}
public class Demo {
    public static void main(String[] args) {
        List<Person> persons = new ArrayList<Person>();
```

```
        persons.add(new Person("Robert", "Male", "Single"));
        ...添加其他人
        Criteria single = new CriterlaSingle();
        Criterla female = new CriteriaFemale();
        Criteria singleOrFemale = new OrCriteria(single, female);
        singleOrFemale.meetCriteria(persons);
    }
}
```

组合模式（Composite Pattern）

组合模式，又叫部分整体模式，是用于把一组相似的对象当作一个单一的对象。组合模式依据树形结构来组合对象，用来表示部分以及整体层次。
这种类型的设计模式属于结构型模式，它创建了对象组的树形结构。 这种模式创建了一个包含自己对象组的类。该类提供了修改相同对象组的方式。
我们通过下面的实例来演示组合模式的用法。实例演示了一个组织中员工的层次结构。

意图

将对象组合成树形结构以表示"部分-整体"的层次结构。组合模式使得用户对单个对象和组合对象的使用具有一致性。

主要解决

它在我们树型结构的问题中，模糊了简单元素和复杂元素的概念，客户程序可以向处理简单元素一样来处理复杂元素，从而使得客户程序与复杂元素的内部结构解耦。

何时使用

1、您想表示对象的部分-整体层次结构（树形结构）。
2、您希望用户忽略组合对象与单个对象的不同，用户将统一地使用组合结构中的所有对象。

如何解决

树枝和叶子实现统一接口，树枝内部组合该接口。

关键代码

树枝内部组合该接口，并且含有内部属性 List，里面放 Component。

应用实例

1、算术表达式包括操作数、操作符和另一个操作数，其中，另一个操作符也可以是操作树、操作符和另一个操作数。
2、在 JAVA AWT 和 SWING 中，对于 Button 和 Checkbox 是树叶，Container 是树枝。

优点

1、高层模块调用简单。
2、节点自由增加。

缺点

在使用组合模式时，其叶子和树枝的声明都是实现类，而不是接口，违反了依赖倒置原则。

使用场景

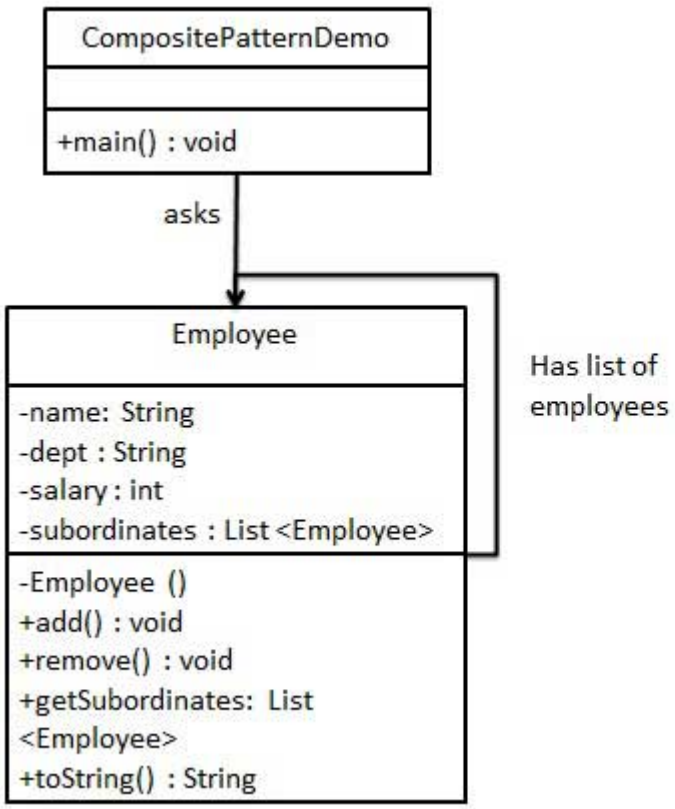
部分、整体场景，如树形菜单，文件、文件夹的管理。

注意事项

定义时为具体类。

示例实现

我们有一个类 Employee，该类被当作组合模型类。
我们的演示类使用 Employee 类来添加部门层次结构，并打印所有员工。




```
public class Employee {
    private String name;
    private String dept;
    private int salary;
    private List<Employee> subordinates;

    public Employee(String name, String dept, int sal) {
        this.name = name;
        this.dept = dept;
```

```
        this.salary = sal;
        subordinates = new ArrayList<Employee>();
    }
}
```

装饰器模式（Decorator Pattern）

装饰器模式允许向一个现有的对象添加新的功能，同时又不改变其结构。这种类型的设计模式属于结构型模式，它是作为现有的类的一个包装。
这种模式创建了一个装饰类，用来包装原有的类，并在保持类方法签名完整性的前提下，提供了额外的功能。
我们通过下面的实例来演示装饰器模式的用法。其中，我们将把一个形状装饰上不同的颜色，同时又不改变形状类。

意图

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。

主要解决

一般的，我们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。

何时使用

在不想增加很多子类的情况下扩展类。

如何解决

将具体功能职责划分，同时继承装饰者模式。

关键代码

1、Component 类充当抽象角色，不应该具体实现。
2、修饰类引用和继承 Component 类，具体扩展类重写父类方法。

应用实例

1、孙悟空有 72 变，当他变成"庙宇"后，他的根本还是一只猴子，但是他又有了庙宇的功能。
2、不论一幅画有没有画框都可以挂在墙上，但是通常都是有画框的，并且实际上是画框被挂在墙上。在挂在墙上之前，画可以被蒙上玻璃，装到框子里；这时画、玻璃和画框形成了一个物体。

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。

使用场景

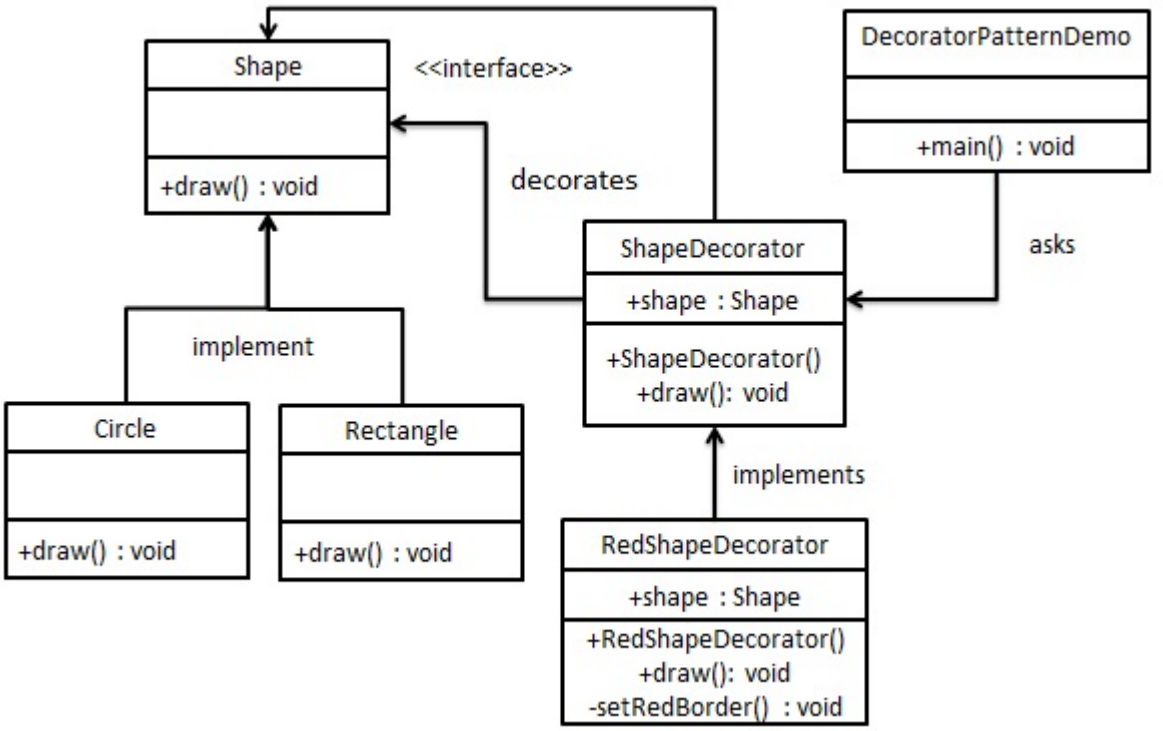
1、扩展一个类的功能。
2、动态增加功能，动态撤销。

注意事项

可代替继承。

示例实现

我们将创建一个 Shape 接口和实现了 Shape 接口的实体类。然后我们创建一个实现了 Shape 接口的抽象装饰类 ShapeDecorator，并把 Shape 对象作为它的实例变量。
RedShapeDecorator 是实现了 ShapeDecorator 的实体类。 我们的演示类使用 RedShapeDecorator 来装饰 Shape 对象。



下面的装饰器使用传入的Shape作为被装饰的目标，在原有方法的基础上还会调用装饰方法setRedBorder()：

```
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");
    }
}
```

外观模式（Facade Pattern）

外观模式隐藏系统的复杂性，并向客户端提供了一个客户端可以访问系统的接口。这种类型的设计模式属于结构型模式，它**向现有的系统添加一个接口，来隐藏系统的复杂性**。
这种模式涉及到一个单一的类，该类提供了客户端请求的简化方法和对现有系统类方法的委托调用。

意图

为子系统中的一组接口提供一个一致的界面，外观模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。

主要解决

降低访问复杂系统的内部子系统时的复杂度，简化客户端与之的接口。

何时使用

1、客户端不需要知道系统内部的复杂联系，整个系统只需提供一个"接待员"即可。
2、定义系统的入口。

如何解决

客户端不与系统耦合，外观类与系统耦合。

关键代码

在客户端和复杂系统之间再加一层，这一次将调用顺序、依赖关系等处理好。

应用实例

1、去医院看病，可能要去挂号、门诊、划价、取药，让患者或患者家属觉得很复杂，如果有提供接待人员，只让接待人员来处理，就很方便。
2、JAVA 的三层开发模式。

优点

1、减少系统相互依赖。
2、提高灵活性。
3、提高了安全性。

缺点

不符合开闭原则，如果要改东西很麻烦，继承重写都不合适。

使用场景

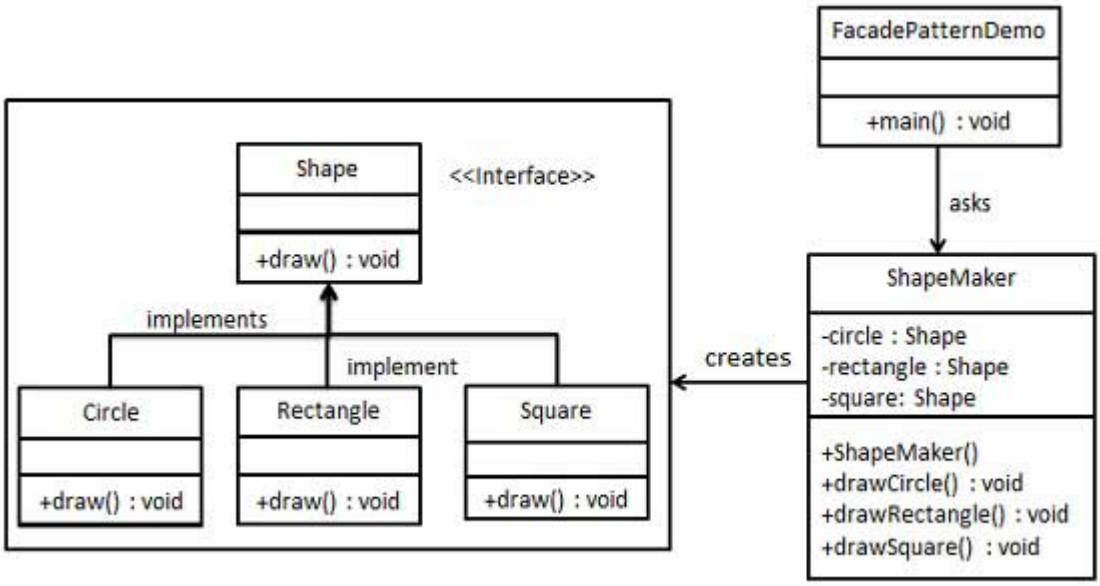
1、为复杂的模块或子系统提供外界访问的模块。
2、子系统相对独立。
3、预防低水平人员带来的风险。

注意事项

在层次化结构中，可以使用外观模式定义系统中每一层的入口。

示例实现

我们将创建一个 Shape 接口和实现了 Shape 接口的实体类。下一步是定义一个外观类 ShapeMaker。ShapeMaker 类使用实体类来代表用户对这些类的调用。我们的演示类使用 ShapeMaker 类来显示结果。



享元模式（Flyweight Pattern）

享元模式主要用于减少创建对象的数量，以减少内存占用和提高性能。这种类型的设计模式属于结构型模式，它提供了减少对象数量从而改善应用所需的对象结构的方式。
享元模式尝试重用现有的同类对象，如果未找到匹配的对象，则创建新对象。我们将通过创建 5 个对象来画出 20 个分布于不同位置的圆来演示这种模式。由于只有 5 种可用的颜色，所以 color 属性被用来检查现有的 Circle 对象。

意图

运用共享技术有效地支持大量细粒度的对象。

主要解决

在有大量对象时，有可能会造成内存溢出，我们把其中共同的部分抽象出来，如果有相同的业务请求，直接返回在内存中已有的对象，避免重新创建。

何时使用

1、系统中有大量对象
2、这些对象消耗大量内存
3、这些对象的状态大部分可以外部化
4、这些对象可以按照内蕴状态分为很多组，当把外蕴对象从对象中剔除出来时，每一组对象都可以用一个对象来代替
5、系统不依赖于这些对象身份，这些对象是不可分辨的。

如何解决

用唯一标识码判断，如果在内存中有，则返回这个唯一标识码所标识的对象。

关键代码

用 HashMap 存储这些对象。

应用实例

1、JAVA 中的 String，如果有则返回，如果没有则创建一个字符串保存在字符串缓存池里面。
2、数据库的数据池。

优点

大大减少对象的创建，降低系统的内存，使效率提高。

缺点

提高了系统的复杂度，需要分离出外部状态和内部状态，而且外部状态具有固有化的性质，不应该随着内部状态的变化而变化，否则会造成系统的混乱。

使用场景

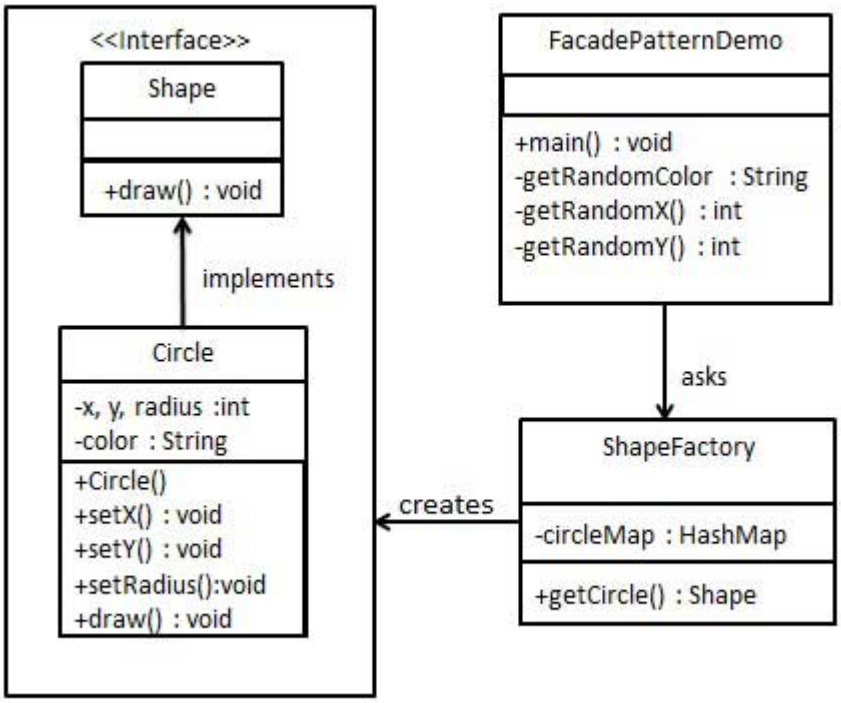
1、系统有大量相似对象。
2、需要缓冲池的场景。

注意事项

1、注意划分外部状态和内部状态，否则可能会引起线程安全问题。
2、这些类必须有一个工厂对象加以控制。

示例实现

我们将创建一个 Shape 接口和实现了 Shape 接口的实体类 Circle。下一步是定义工厂类 ShapeFactory。ShapeFactory 有一个 Circle 的 HashMap，其中键名为 Circle 对象的颜色。无论何时接收到请求，都会创建一个特定颜色的圆。ShapeFactory 检查它的 HashMap 中的 circle 对象，如果找到 Circle 对象，则返回该对象，否则将创建一个存储在 hashmap 中以备后续使用的新对象，并把该对象返回到客户端。FlyWeightPatternDemo，我们的演示类使用 ShapeFactory 来获取 Shape 对象。它将向 ShapeFactory 传递信息（red / green / blue/ black / white），以便获取它所需对象的颜色。



享元模式和工厂模式有些像，但享元模式的关键是池的构造：

```
public class ShapeFactory {
    private static final HashMap<String, Shape> circleMap = new HashMap();
    public static Shape getCircle(String color) {
        Circle circle = (Circle) circleMap.get(color);
        if(circle == null) {
            circle = new Circle(color);
            circleMap.put(color, circle);
            System.out.println("Creating circle of color: " + color);
        }
        return circle;
    }
}
```

代理模式（Proxy）

意图

为其他对象提供一种代理以控制对这个对象的访问（访问控制）。

主要解决

对一个对象进行访问控制的一个原因是为了只有在我们确实需要这个对象时才对它进行创建和初始化。我们考虑一个可以在文档中嵌入图形对象的文档编辑器。有些图形对象(如大型光栅图像)的创建开销很大。但是打开文档必须很迅速,因此我们在打开文档时应避免一次性创建所有开销很大的对象。因为并非所有这些对象在文档中都同时可见,所以也没有必要同时创建这些对象。

何时使用

1. 需要进行访问控制；
2. 需要延迟创建目标对象，因为目标对象的生命周期由代理对象管理；

如何解决

使代理类继承目标类或实现目标类的父接口，这样客户端在使用代理类时和使用目标类时是一样的。

关键代码

在客户端和目标对象之间添加一层代理，代理类可以继承目标类（类代理，CGLIB）或实现目标类的接口（接口代理，JkdProxy）。

应用实例

动机一节中 virtual proxy的例子来自于E T + +的文本构建块类。
N E X T S T E P [A d d 9 4]使用代理 (类N X P r o x y的实例)作为可分布对象的本地代表,当客户请求远程对象时,服务器为这些对象创建代理。收到消息后,代理对消息和它的参数进行编码,并将编码后的消息传递给远程实体。类似的,实体对所有的返回结果编码,并将它们返回给N X P r o x y对象。
McCullough [McC87]讨论了在S m a l l t a l k中用代理访问远程对象的问题。 Pascoe [Pas86]讨论了如何用“封装器”(E n c a p s u l a t o r s)控制方法调用的副作用以及进行访问控制。

优点

P r o x y模式在访问对象时引入了一定程度的间接性。根据代理的类型,附加的间接性有多种用途:
1) Remote Proxy可以隐藏一个对象存在于不同地址空间的事实。
2) Virtual Proxy 可以进行最优化,例如根据要求创建对象。
3) Protection Proxies和Smart Reference都允许在访问一个对象时有一些附加的内务处理(Housekeeping task)。
P r o x y模式还可以对用户隐藏另一种称之为 c o p y - o n - w r i t e的优化方式,该优化与根据需要创建对象有关。拷贝一个庞大而复杂的对象是一种开销很大的操作,如果这个拷贝根本没有被修改,那么这些开销就没有必要。用代理延迟这一拷贝过程,我们可以保证只有当这个对象被修改的时候才对它进行拷贝。
在实现 C o p y - o n - w r i t e时必须对实体进行引用计数。拷贝代理仅会增加引用计数。只有当用户请求一个修改该实体的操作时,代理才会真正的拷贝它。在这种情况下,代理还必须减少实体的引用计数。当引用的数目为零时,这个实体将被删除。
C o p y - o n - Wr i t e可以大幅度的降低拷贝庞大实体时的开销。

缺点

过多的代理类容易使得程序过于复杂。

使用场景

在需要用比较通用和复杂的对象指针代替简单的指针的时候,使用 Ｐｒｏｘｙ模式。下面是一些可以使用 Ｐｒｏｘｙ模式常见情况:

1. 远 程 代 理 (Remote Proxy) 为 一 个 对 象在 不 同 的 地 址 空间 提 供 局 部 代 表。NEXTSTEP[Add94] 使用N Ｘ Ｐｒｏｘｙ类实现了这一目的。 Coplien[Cop92] 称这种代理为“大使”(Ａｍｂａｓｓａｄｏｒ)。
2. 虚代理(Virtual Proxy)根据需要创建开销很大的对象。在动机一节描述的 ＩｍａｇｅＰｒｏｘｙ就是这样一种代理的例子。
3. 保护代理(Protection Proxy)控制对原始对象的访问。保护代理用于对象应该有不同的访问权限的时候。例如,在 Ｃｈｏｉｃｅｓ操作系统 [ＣＩＲＭ９３]中Ｋｅｍｅｌ Ｐｒｏｘｉｅｓ为操作系统对象提供
了访问保护。
4. 智能指引 (Smart Reference)取代了简单的指针,它在访问对象时执行一些附加操作。它的典型用途包括:
    - 对指向实际对象的引用计数,这样当该对象没有引用时,可以自动释放它 (也称为Ｓｍａｒｔ Ｐｏｉｎｔｅｒｓ[Ｅｄｅ９２])。
    - 当第一次引用一个持久对象时,将它装入内存。
    - 在访问一个实际对象前,检查是否已经锁定了它,以确保其他对象不能改变它。

注意事项

代理模式有多种实现方式

相关模式

Adapter: 适配器Adapter为它所适配的对象提供了一个不同的接口。相反,代理提供了与它的实体相同的接口。然而,用于访问保护的代理可能会拒绝执行实体会执行的操作,因此,它的接口实际上可能只是实体接口的一个子集。
Ｄｅｃｏｒａｔｏｒ: 尽管ｄｅｃｏｒａｔｏｒ的实现部分与代理相似,但ｄｅｃｏｒａｔｏｒ的目的不一样。Ｄｅｃｏｒａｔｏｒ为对象添加一个或多个功能,而代理则控制对对象的访问。
代理的实现与 ｄｅｃｏｒａｔｏｒ的实现类似,但是在相似的程度上有所差别。 Protection Proxy的实现可能与 ｄｅｃｏｒａｔｏｒ的实现差不多。另一方面, Remote Proxy不包含对实体的直接引用,而只是一个间接引用,如“主机 ＩＤ,主机上的局部地址。”Virtual Proxy开始的时候使用一个间接引用,例如一个文件名,但最终将获取并使用一个直接引用。

示例实现

以下以静态接口代理为例:

```
interface Inter {
    void print();
}
class Target implements Inter {
    public void print() {
```

```
        System.out.println("Target");
    }
}
class Proxy implements Inter {
    private Inter target;
    public Proxy(Inter target) {
        this.target = target;
    }
    public void print() {
        System.out.println(">> Proxy");
        target.print();
        System.out.println("<< Proxy");
    }
}
public class Main {
    public static void main(String[] args) {
        Inter target = new Target();
        Inter proxy = new Proxy(target);
        proxy.print();
    }
}
```