

2017编程提高第7节课——面向对象设计(7)

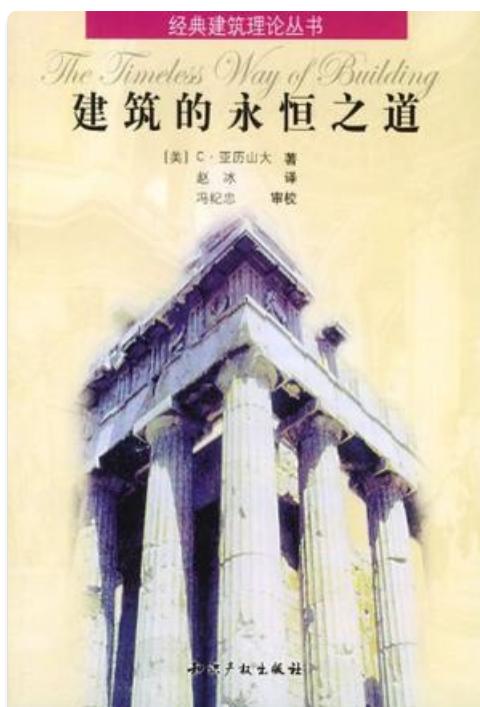
设计模式 工厂模式 建造者模式 单例模式 面向对象设计 课件

- content
{:toc}

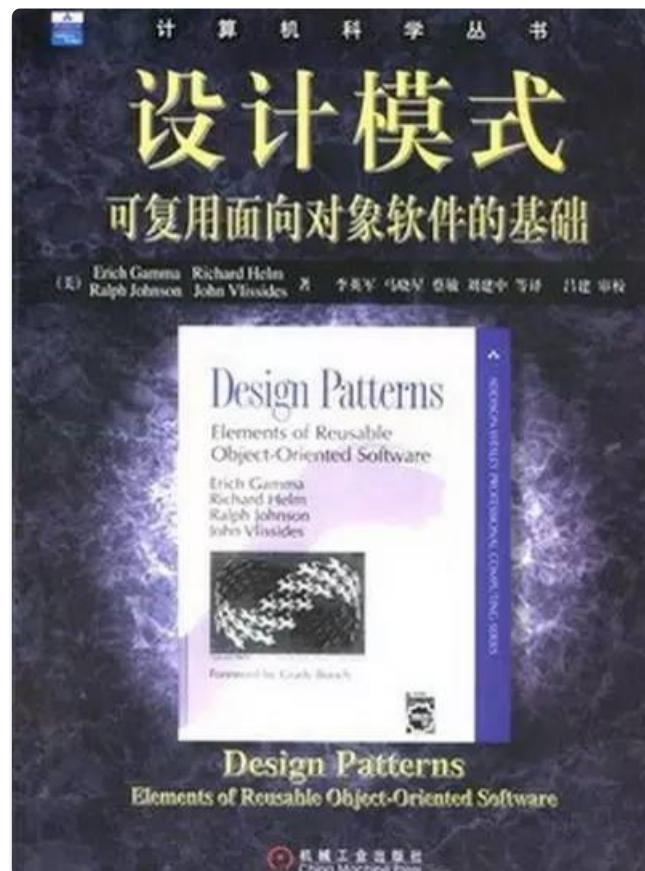
讲解设计模式的思路

- 传统思路
 - 动机、定义
 - 结构、分析
 - 案例
- 先讲问题，然后讲一般的思路，最后引导出模式
 - Why → How

神书

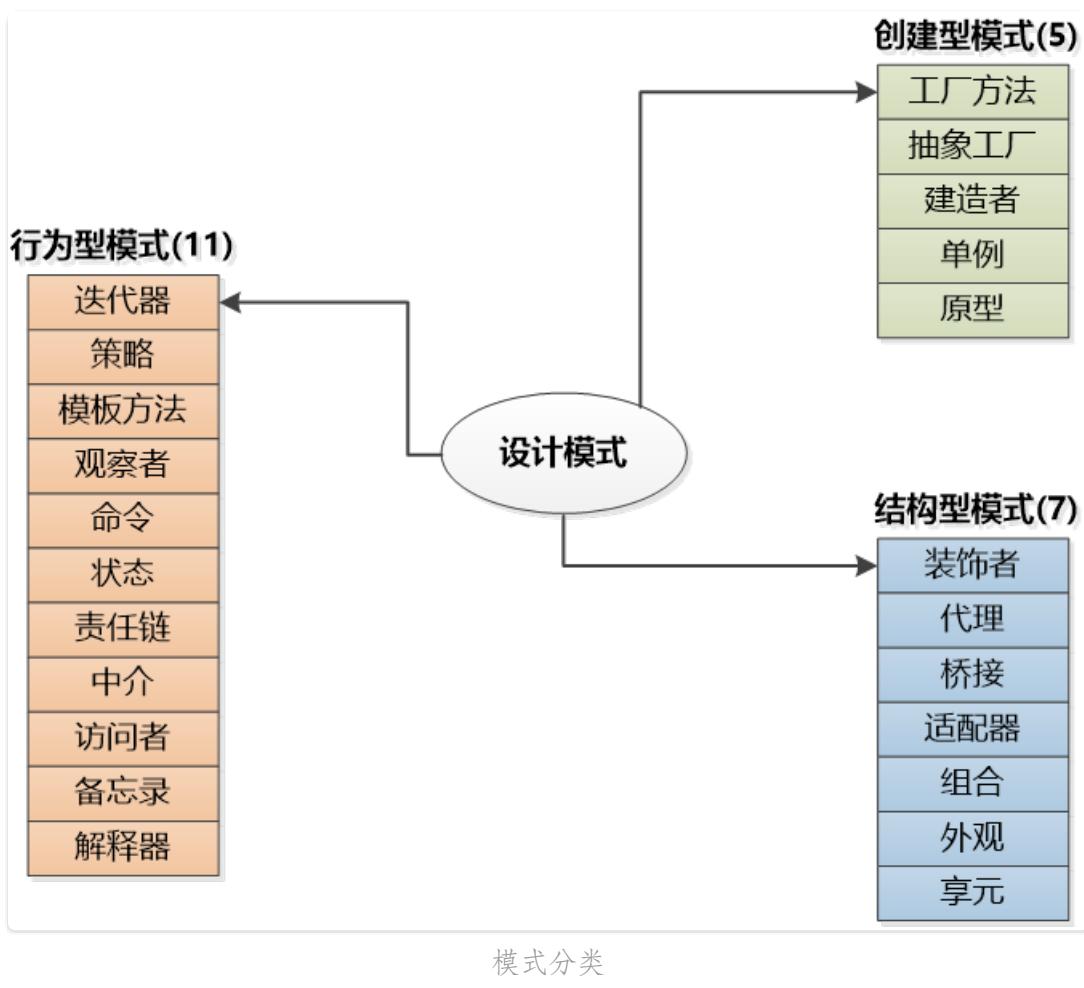


神书1



神书2

模式的分类



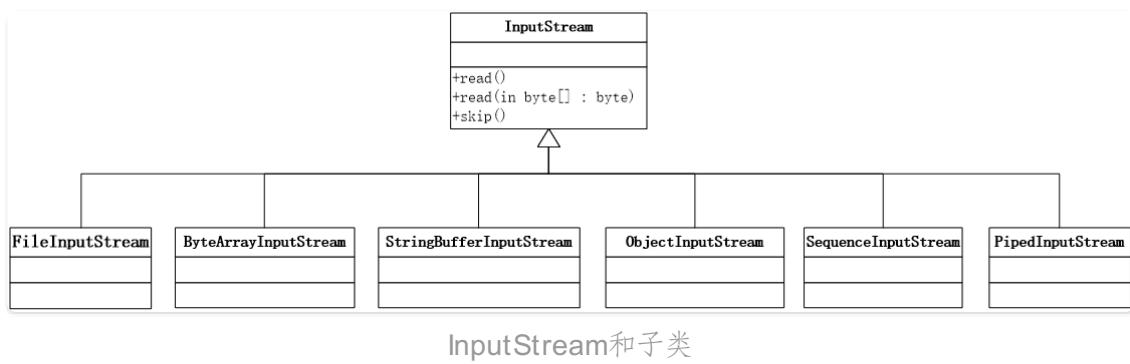
模式分类

再次重申

- 设计模式本质上是面向对象设计原则的体现
 - 针对接口编程而不是针对实现编程
 - 优先使用组合而不是继承
 - 发现变化并且封装变化
- 学会设计模式，然后忘掉设计模式
 - 用设计模式的思想去编程

装饰者模式

Java InputStream设计



InputStream和子类

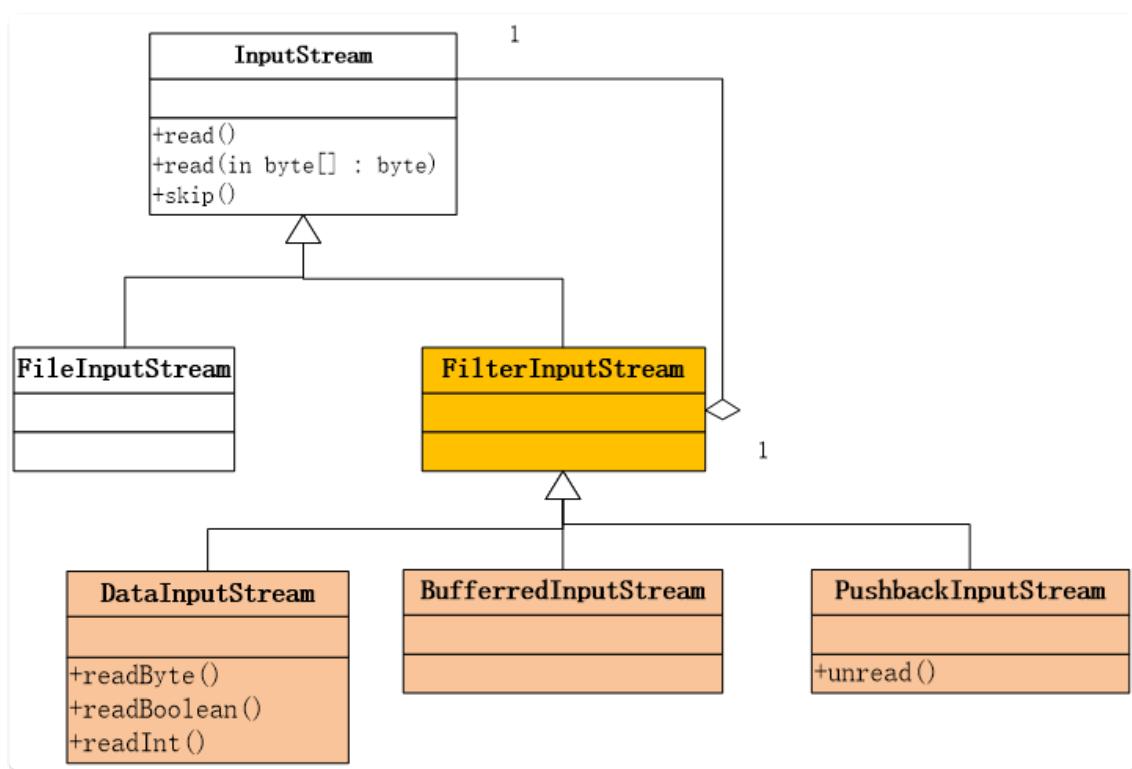
FileInputStream	从文件中读取数据，形成Stream
ByteArrayInputStream	从一个字节数组中读数据，形成Stream
StringBufferInputStream	从一个字符串中读数据，形成Stream
PipedInputStream	从另外一个线程读取数据，形成Stream
SequenceInputStream	把两个stream进行合并形成 Stream
ObjectInputStream	从序列化数据中形成Stream

子类介绍

- **Inputstream**是基于字节的
 - 需求1：直接读取Java 基本类型，而不是字节
 - 需求2：需要支持缓冲
 - 需求3：从Stream中读取的数据还可以放回Stream去
- 解决办法1：把这些功能加入到父类**InputStream**中
 - 不可行！每个子类对数据的读取方法都不同，父类无法知道每个子类的细节。
- 解决办法2：子类化
 - DataFileInputStream
 - BufferedFileInputStream
 - PushBackFileInputStream
 - 每个子类例如**FileInputStream**都需要创建3个子类

子类的爆炸问题

- 根据排列组合，仅仅**FileInputStream**就需要7个子类才能应对所有变化
 - Data + FileInputStream
 - Buffered + FileInputStream
 - PushBack + FileInputStream
 - Buffered + Data + FileInputStream
 - Buffered + PushBack + FileInputStream
 - Data + PushBack + FileInputStream
 - Data + Buffered + PushBack + FileInputStream



子类爆炸的解决

```

public class FilterInputStream extends InputStream {

    protected volatile InputStream in;

    protected FilterInputStream(InputStream in) {
        this.in = in;
    }

    public int read() throws IOException {
        return in.read();
    }

    public int read(byte b[]) throws IOException {
        return read(b, 0, b.length);
    }

    public int read(byte b[], int off, int len) throws IOException {
        return in.read(b, off, len);
    }
}
  
```

图片7

```

public class DataInputStream extends FilterInputStream implements DataInput {
    private byte bytearr[] = new byte[80];
    private char chararr[] = new char[80];

    public DataInputStream(InputStream in) {
        super(in);
    }

    public final short readShort() throws IOException {
        int ch1 = in.read();
        int ch2 = in.read();
        if ((ch1 | ch2) < 0)
            throw new EOFException();
        return (short)((ch1 << 8) + (ch2 << 0));
    }

    public final long readLong() throws IOException {
        readFully(readBuffer, 0, 8);
        return (((long)readBuffer[0] << 56) +
                ((long)(readBuffer[1] & 255) << 48) +
                ((long)(readBuffer[2] & 255) << 40) +
                ((long)(readBuffer[3] & 255) << 32) +
                ((long)(readBuffer[4] & 255) << 24) +
                ((readBuffer[5] & 255) << 16) +
                ((readBuffer[6] & 255) << 8) +
                ((readBuffer[7] & 255) << 0));
    }
}

```

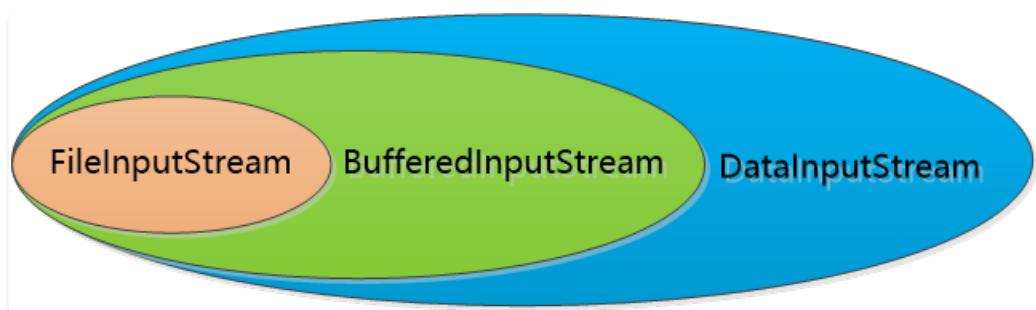
图片8

使用

```

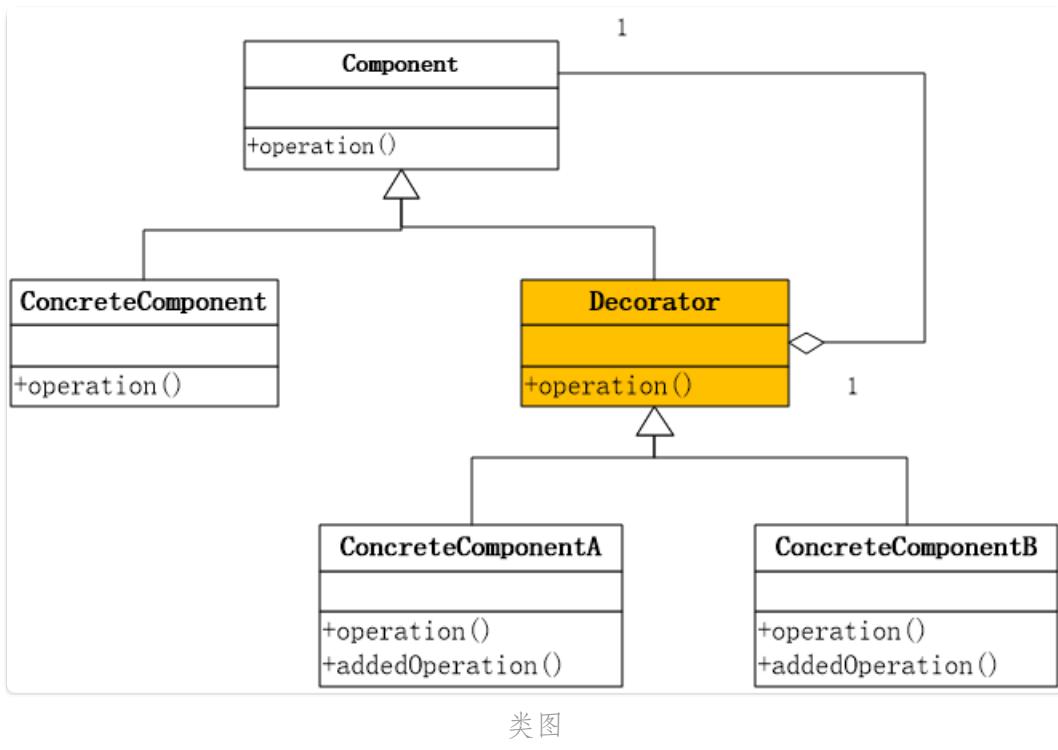
new FileInputStream(...)
new BufferedInputStream(new FileInputStream(...))
new DataInputStream(new BufferedInputStream(new FileInputStream(..)))
new BufferedInputStream(new DataInputStream(new FileInputStream(..)))

```



图片9

Decorator模式类图



类图

代理模式

ATM 和银行主机的通信

- 细节，太多的细节，ATM无法专注于自己的业务
 - 网络
 - 协议
 - 安全
 -

```

public class ATM {
    public void processTransaction(Transaction t) {
        // 和银行端发起链接，建立socket
        // 把transaction 序列化为字符串消息
        // 对消息进行加密
        // 通过socket发送消息
        // 通过socket获得银行端的响应消息
        // 处理响应消息，判断交易是否成功
    }
}

```

ATM机

把细节隐藏起来

```

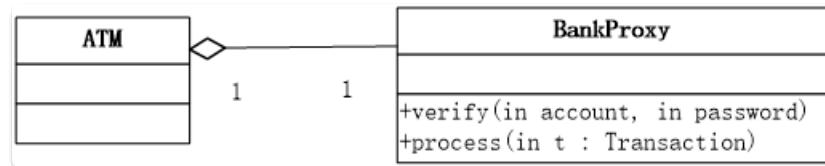
public class ATM {

    BankProxy bankProxy ;

    public void processTransaction(Transaction t) {

        Response res = bankProxy.process(t);
        if(res.isSuccessful()) {
            // ....
        } else {
            // ....
        }
    }
}

```



从数据库获得User对象

```

public interface UserManager {
    public User getUser(String id);
}

```

```

public class UserManagerImpl implements UserManager{

    @Override
    public User getUser(String id) {
        // 从数据库获得User 对象
        return null;
    }
}

```

User对象

增加缓存功能

```

public class UserManagerProxy implements UserManager {
    UserManager userManager;
    Cache<User> cache;
    @Override
    public User getUser(String id) {
        User user = cache.getUser(id);
        if(user != null) {
            return user;
        } else {
            user = userManager.getUser(id);
            cache.putUser(id, user);
            return user;
        }
    }
    public static void main(String[] args) {
        // 调用方代码，获得一个UserManagerProxy对象
        UserManager userManager = new UserManagerProxy();
        User user = userManager.getUser("U13636");
        user = userManager.getUser("U13636");
    }
}

```

缓存

增加权限检查

```

public interface ArticleService {
    public void viewArticle(Article article);
    public void deleteArticle(Article article);
}

public class ArticleServiceProxy implements ArticleService {

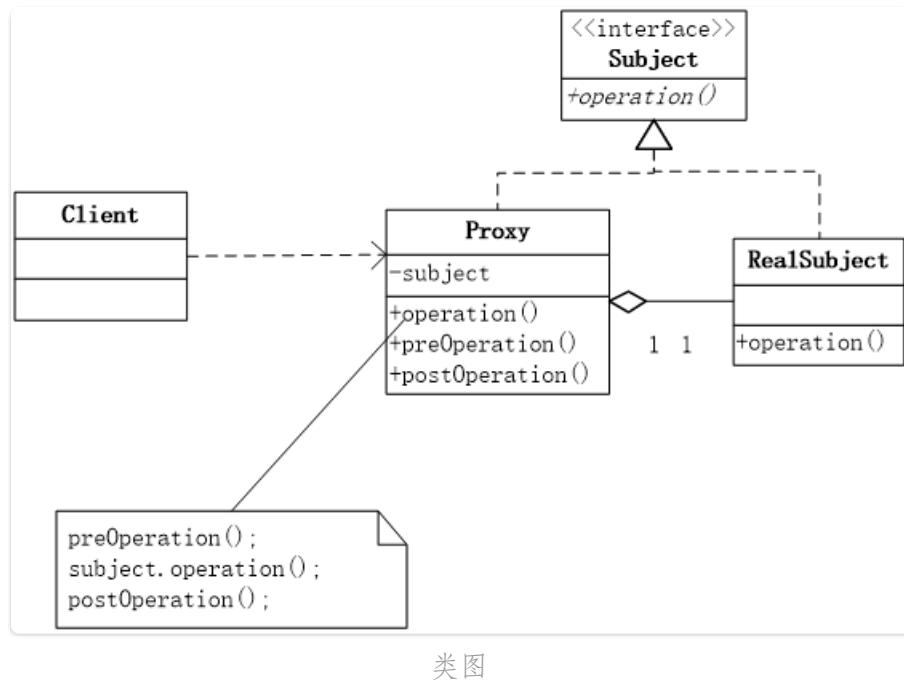
    private ArticleService articleService;
    private User user;
    @Override
    public void viewArticle(Article article) {
        if(user.hasViewPermission()){
            articleService.viewArticle(article);
        } else {
            // 抛出异常
        }
    }

    @Override
    public void deleteArticle(Article article) {
        if(user.hasDeletePermission()){
            articleService.deleteArticle(article);
        } else{
            // 抛出异常
        }
    }
}

```

权限检查

Proxy模式的类图



静态代理 vs 动态代理

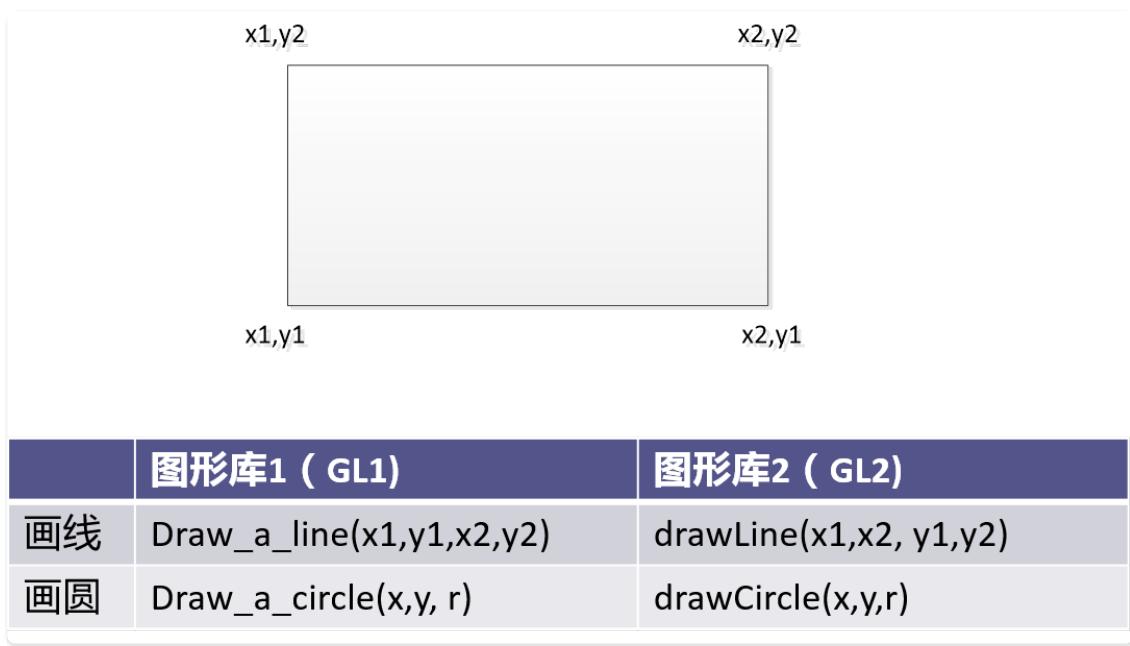
- 静态代理要求Proxy(UserManagerProxy, ArticleServiceProxy, BankProxy)需要手工编码生成
- 在很多时候需要动态给一个类增加行为
 - 例如增加事务，安全，日志等功能
- 动态代理
 - Java 动态代理
 - CGLib ASM javassist

桥接模式

画图程序

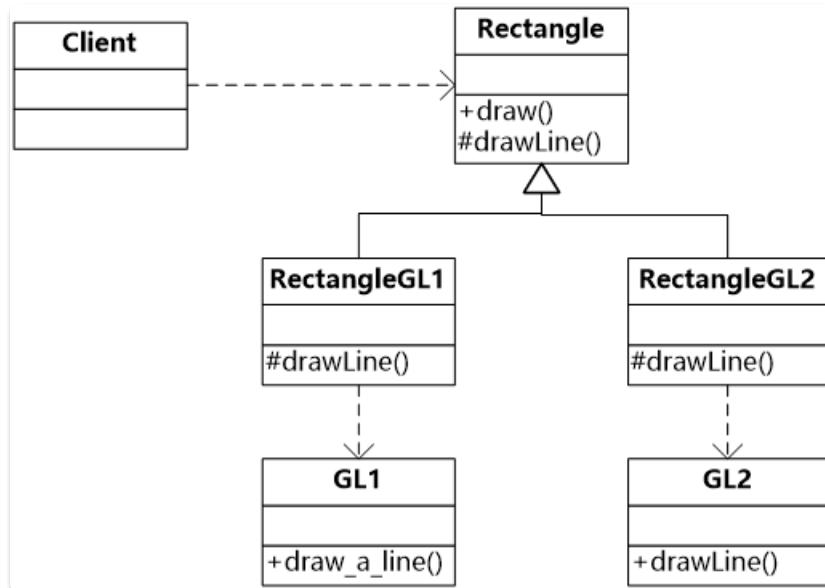
- 领导让你开发一个程序：调用公司的图形库画出一个长方形来，你搜索了一下，发现公司有两个图形库可以使用：
 - Graphic Library 1 (GL1)
 - Graphic Library 2 (GL2)
- 你的程序需要调用这两个图形库之一才能画长方形

两个图形库的差别



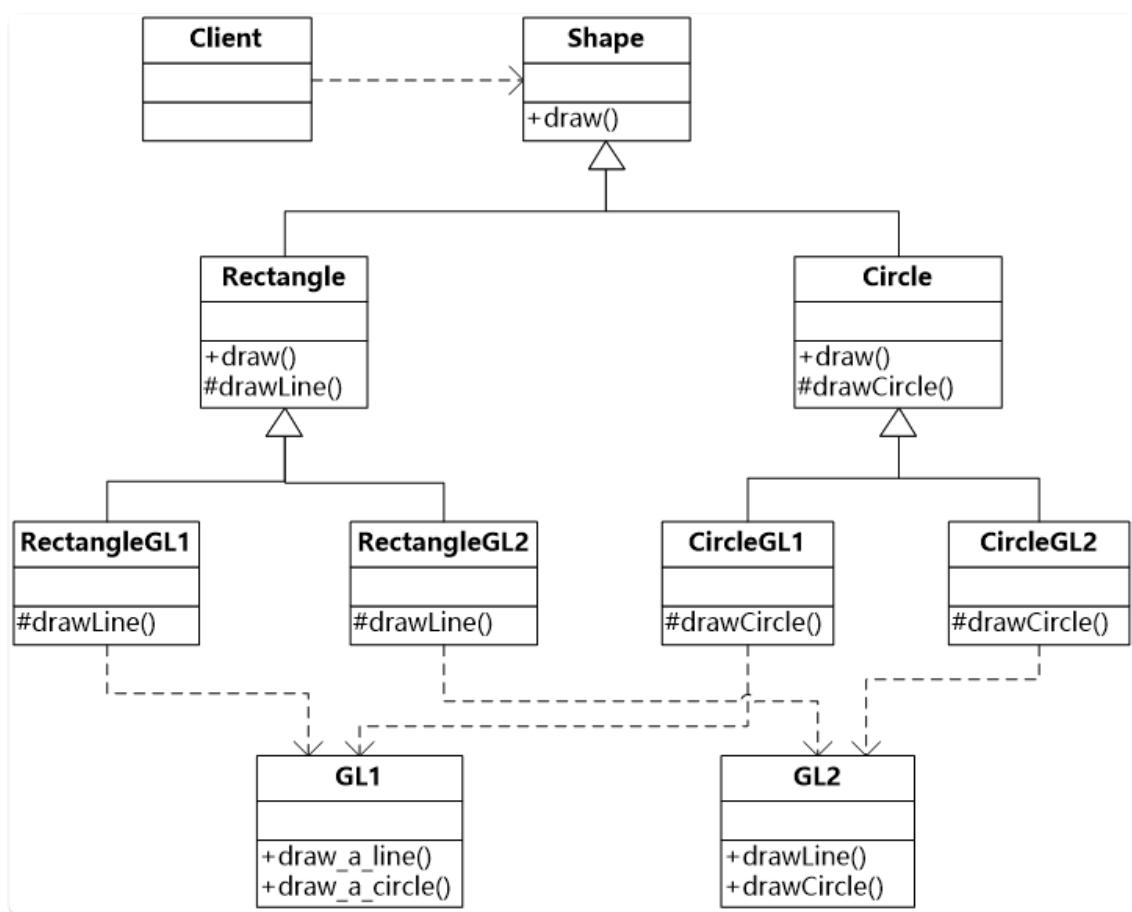
图形库差别

第一次设计



新需求

- 领导要求支持一个新的形状：圆形
- 看来需要抽象： Shape
 - 让 Rectangle 和 Circle 去继承

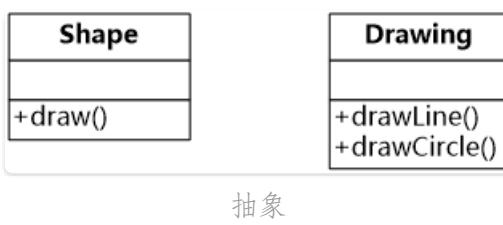


思考

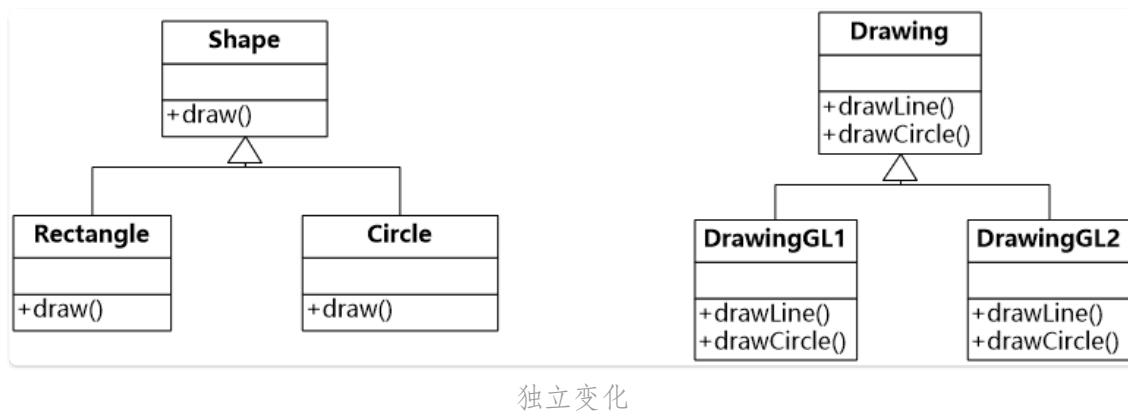
- 现在有两个图形库 GL1, GL2, 如果再来一个图形库 GL3 怎么样?
 - 需要给 Rectangle 添加一个子类 RectangleGL3
 - 需要给 Circle 添加一个子类 CircleGL3
 - 就会有6个子类 (3个图形库 * 2个形状)
- 如果再加一个新的形状, 例如三角形
 - 就会有9个子类 (3个图形库 * 3个形状)
- 子类爆炸问题出现了

解决办法：识别变化

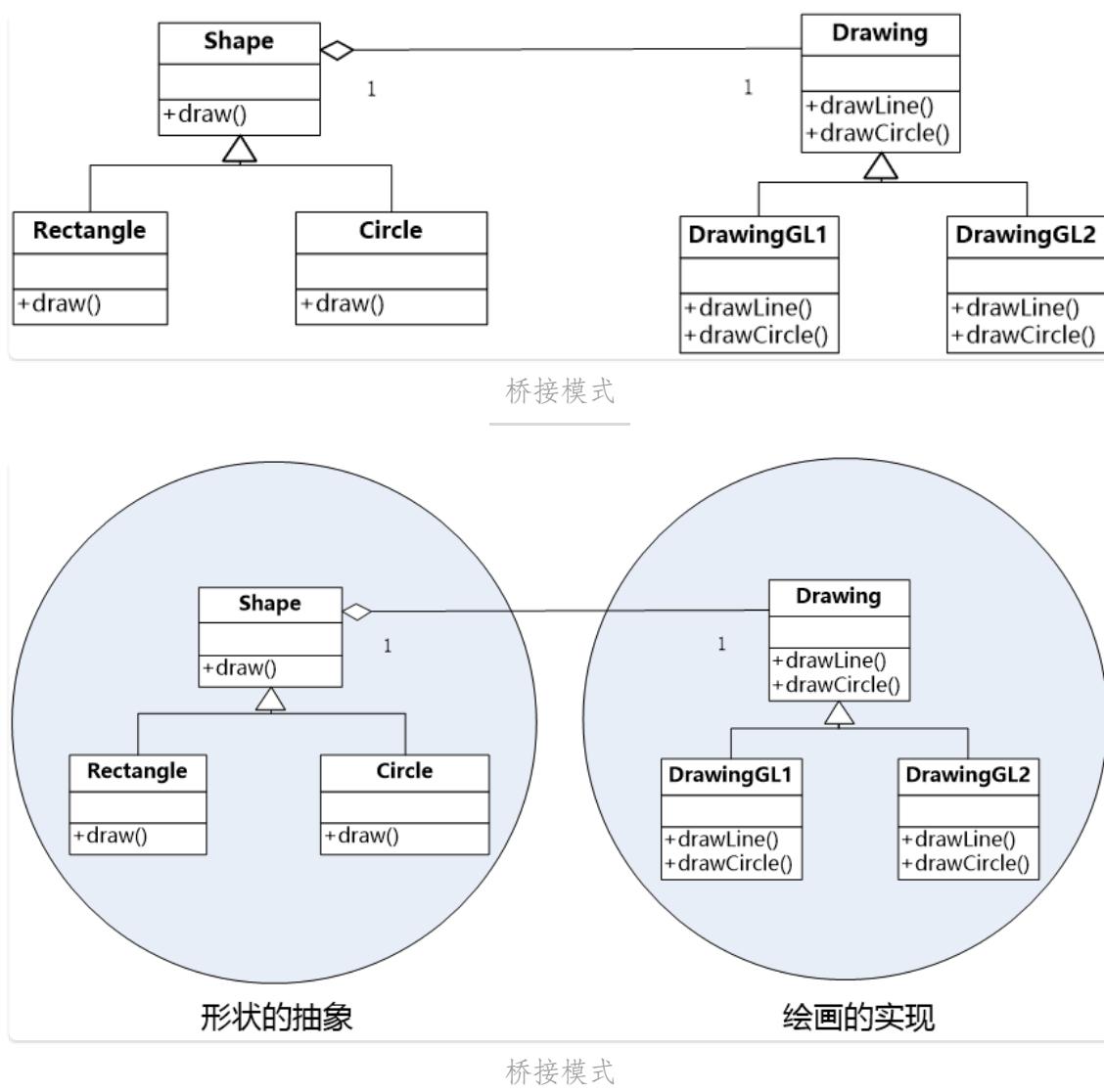
- Shape 可以有多种多样
 - 长方形, 圆形, 三角形.....
- 图形库可以有多种多样
 - 图形库1, 图形库2, 图形库3...
- 这是两个不同的维度! 应该让他们独立的变化
 - 形状已经有了抽象: Shape
 - 关键是对图形库做个抽象!



两个维度独立变化



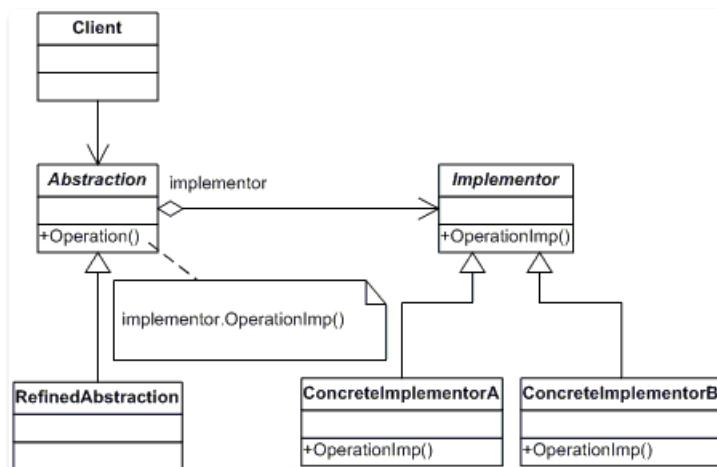
Bridge 模式



Bridge 模式的使用场景

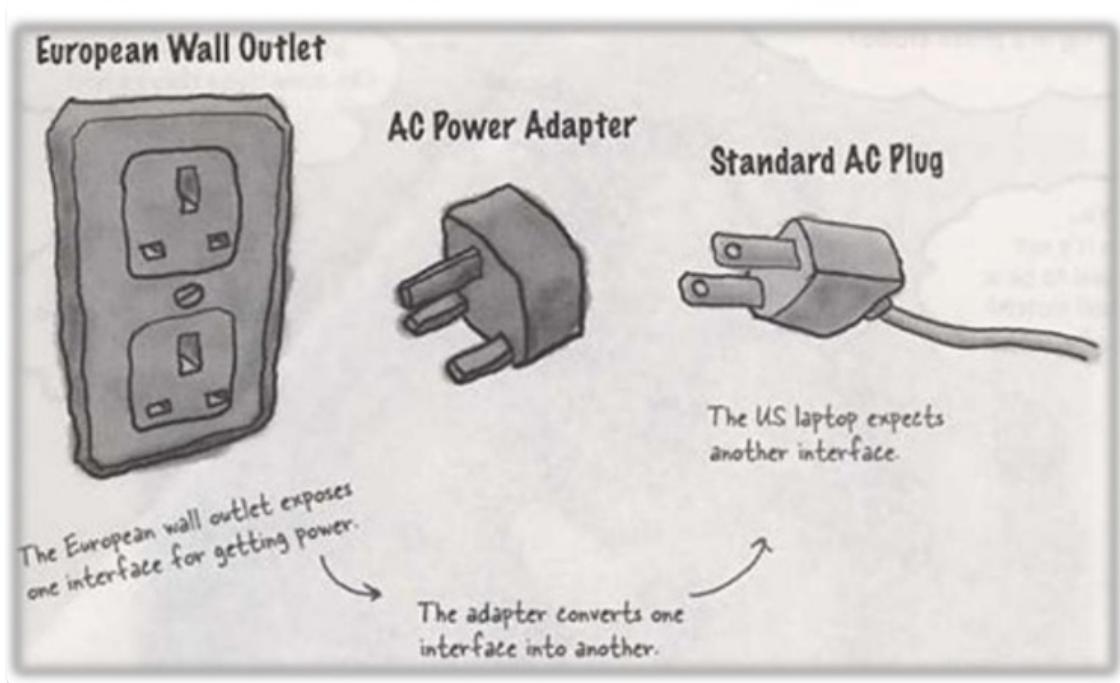
- 当有两个正交的维度，这两个维度有一定的关联，但是还想独立变化的时候.....
 - Shape VS Drawing
 - 文件格式 VS 文件序列化
 - 软件行为 VS 软件平台

Bridge 模式 类图



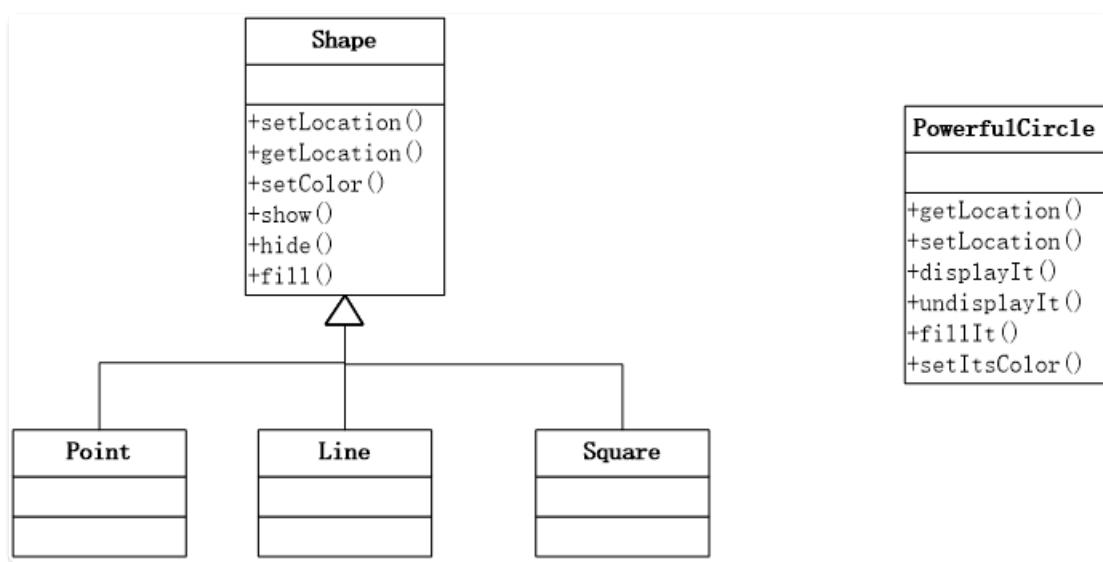
类图

适配器模式

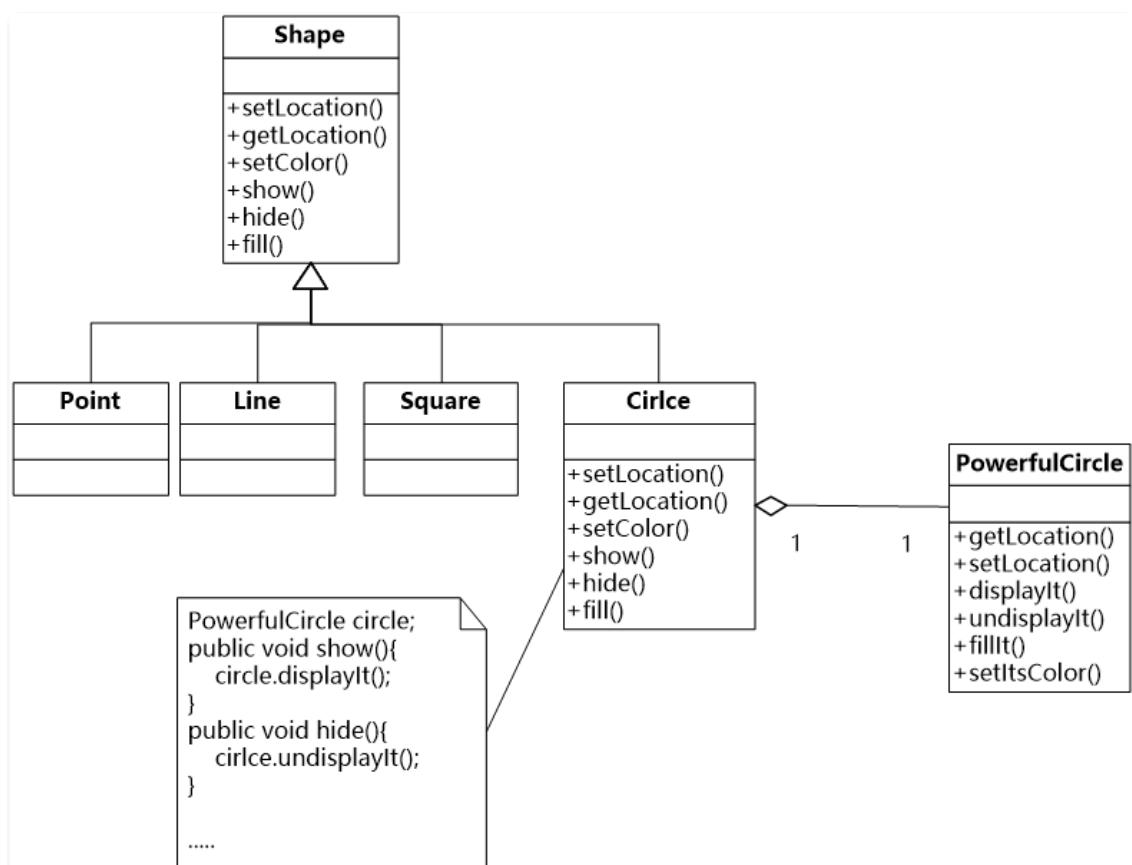


适配器

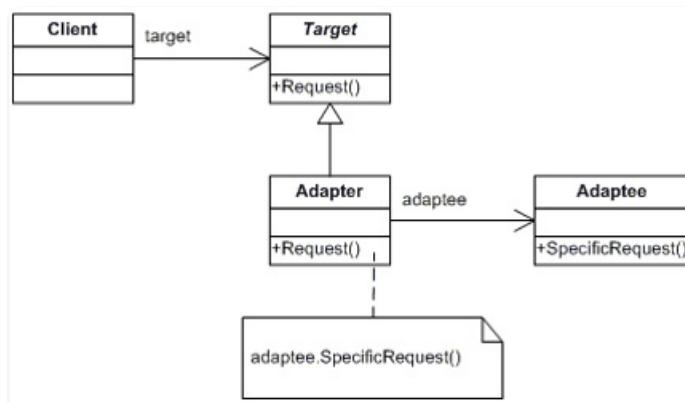
还是图形化的例子



适配一下

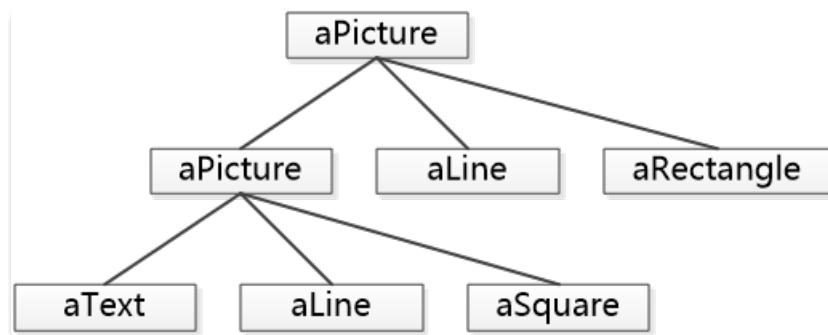


Adapter 模式类图



类图

如何描述带有组合关系的树形结构



树形结构

```

public interface Shape {
    public void draw();
}

public class Line implements Shape {
    @Override
    public void draw() {
        System.out.println("I'm a line");
    }
}

public class Rectangle implements Shape {
    @Override
    public void draw() {
        System.out.println("I'm Rectangle");
    }
}

public class Text implements Shape {
    @Override
    public void draw() {
        System.out.println("I'm a text");
    }
}
  
```

树形结构

```

public class Picture implements Shape {
    List<Shape> shapes = new ArrayList<>();
    @Override
    public void draw() {
        for(Shape shape : shapes) {
            shape.draw();
        }
    }
    public void add(Shape shape) {
        shapes.add(shape);
    }
}

Picture aPicture = new Picture();
aPicture.add(new Line());
aPicture.add(new Rectangle());

Picture p = new Picture();
p.add(new Text());
p.add(new Line());
p.add(new Square());

aPicture.add(p);

aPicture.draw();

```

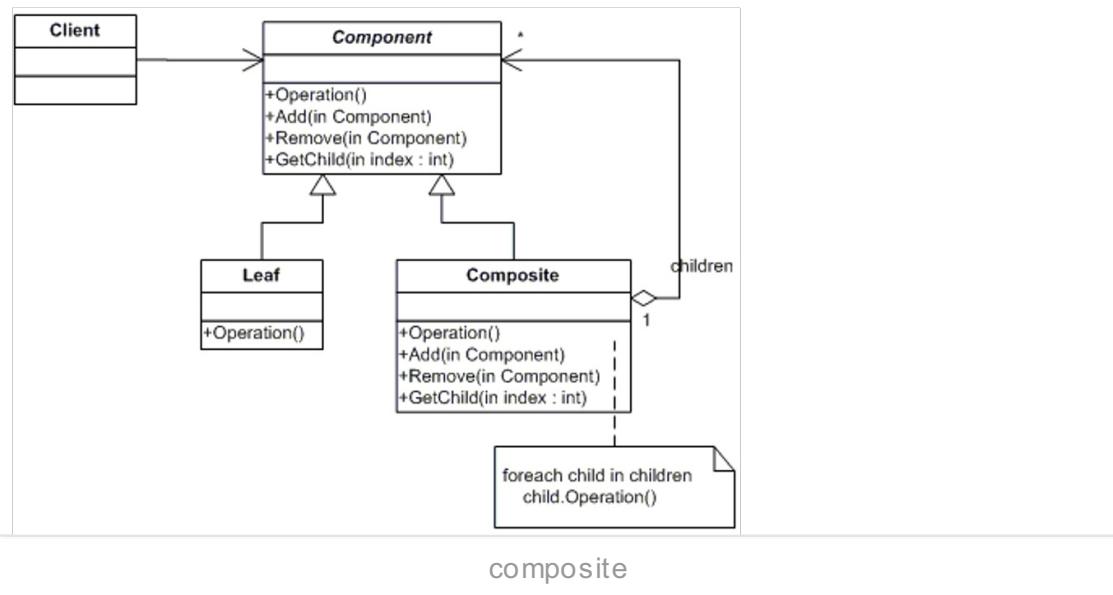
树形结构

Composite 模式

适用场景：

1. 想表达整体和部分的关系
2. 调用方想忽略整体和部分的区别

Composite 模式



Composite 变体

```

public class Employee {
    private String name;
    private String dept;
    private List<Employee> subordinates = new ArrayList<Employee>();

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

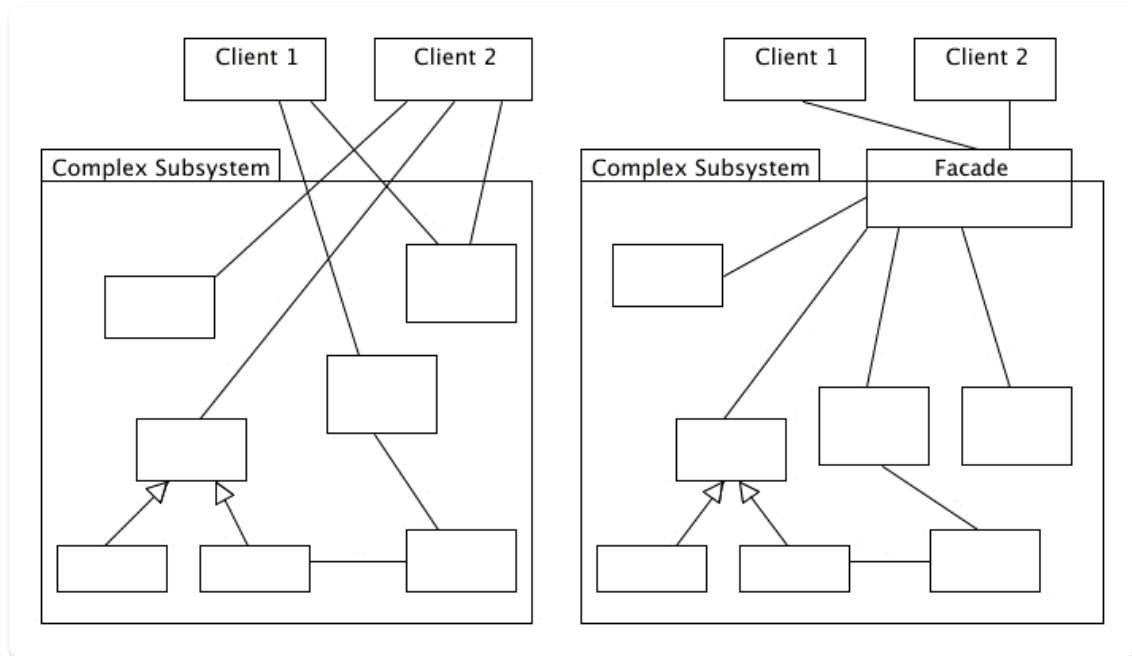
    public void add(Employee e) {
        subordinates.add(e);
    }

    public List<Employee> getSubordinates() {
        return subordinates;
    }

    public String toString() {
        return ("Employee :[ Name : " + name + ", dept : " + dept + "]");
    }
}
  
```

变体

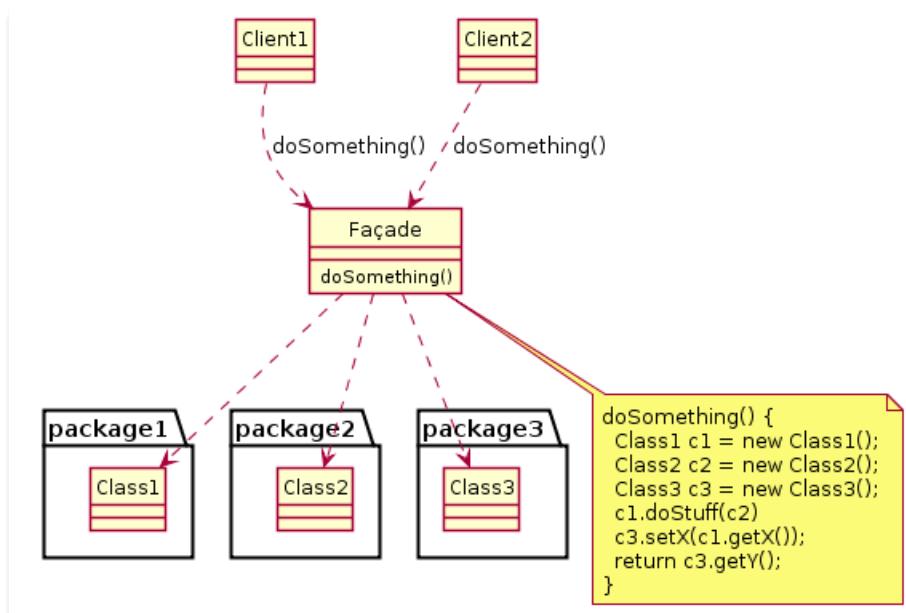
外观(Facade)模式



外观模式

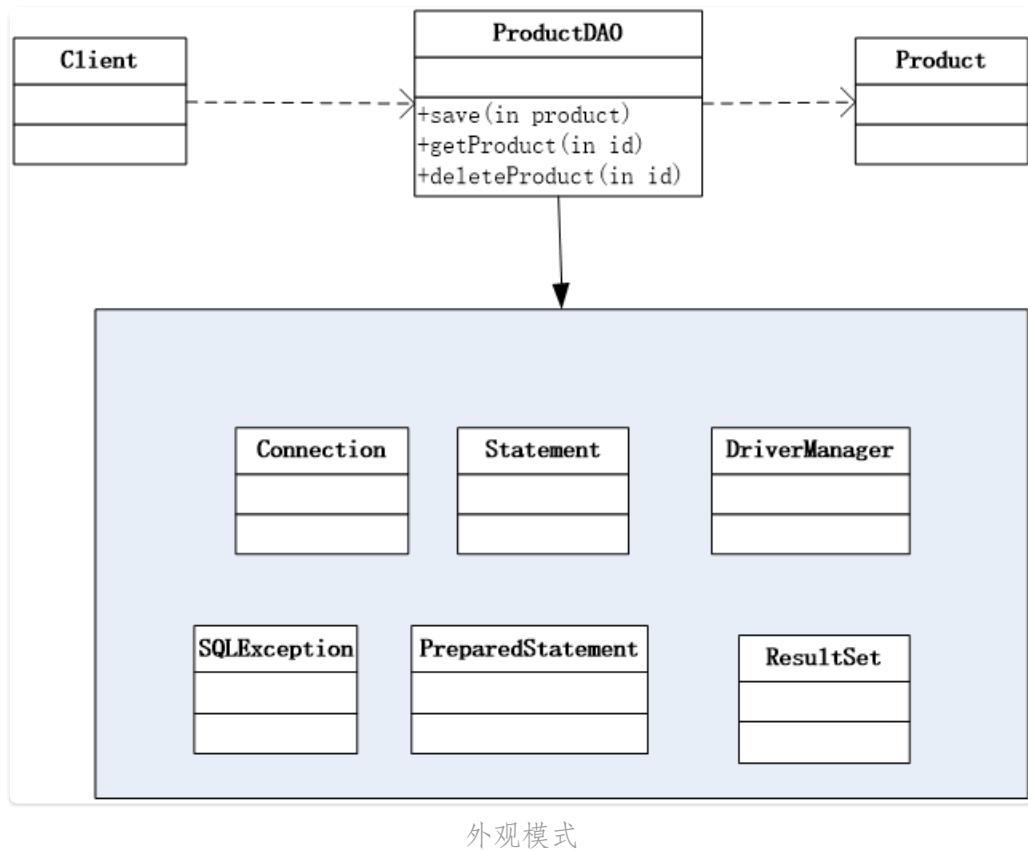
适用场景

- 需要一个简单的接口去访问一个负责的系统
- 如果一个系统非常复杂，难于理解
- 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，层与层之间不直接产生联系，而通过外观类建立联系，降低层之间的耦合度。



外观模式

Facade就在身边



外观模式

What is façade ?

A façade or façade is generally one side of the exterior of a building, especially the front, but also sometimes the sides and rear.

The word comes from the French language, literally meaning "frontage" or "face".



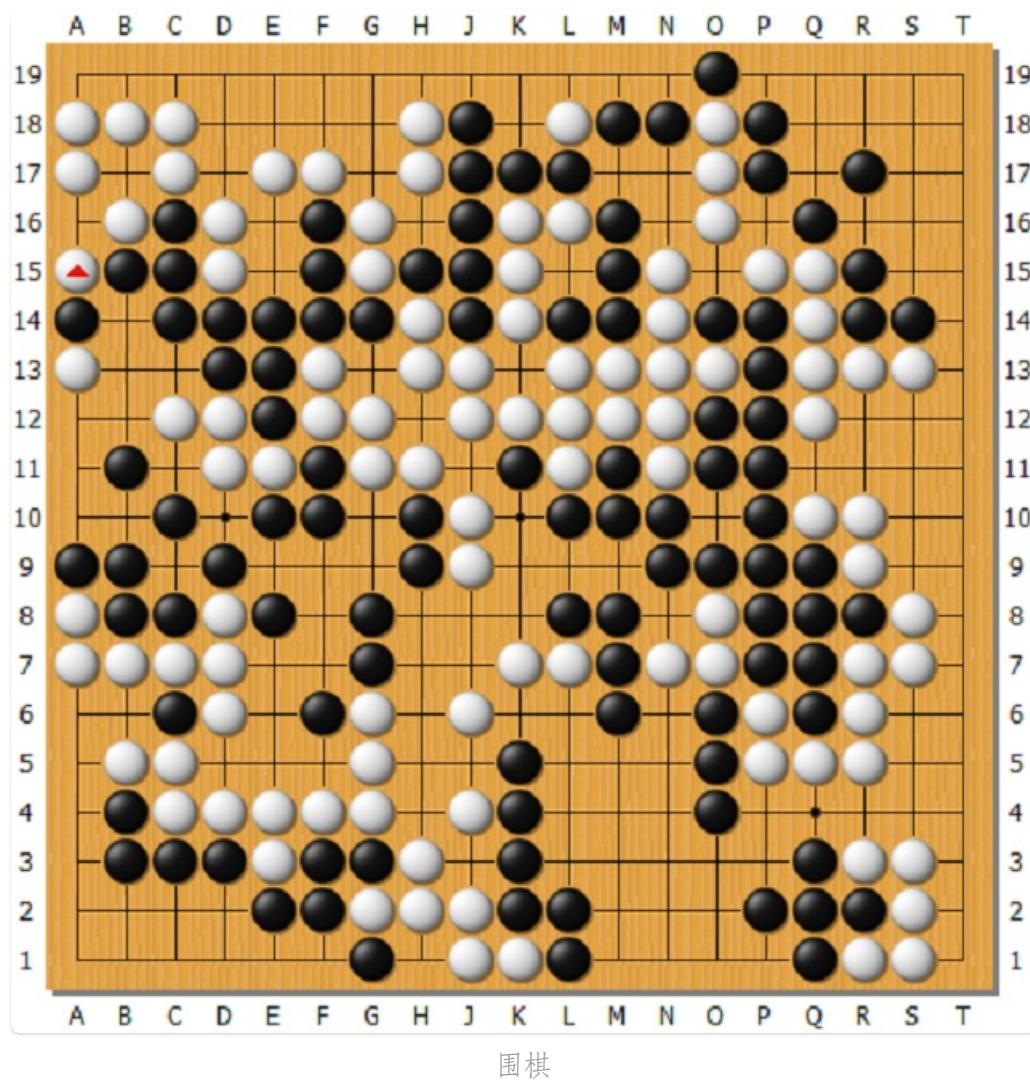
外观模式

围棋软件的例子

每个围棋棋子都是一个对象，属性有：

1. 图标
2. 位置

一个棋盘上的棋子很多，怎么设计才能尽可能的节省空间？



第一种思路

内存中的对象

```
public class GoPiece {
    private Icon icon;
    private Position position;

    public void draw() {
        //....
    }
}
```

Icon = 白色棋子图标
position = (10,20)

Icon = 黑色棋子图标
position = (122,220)

Icon = 白色棋子图标
position = (30,21)

Icon = 黑色棋子图标
position = (132,260)

Icon = 白色棋子图标
position = (40,20)

Icon = 黑色棋子图标
position = (12 , 209)

⋮

⋮

Icon = 白色棋子图标
position = (70,230)

Icon = 黑色棋子图标
position = (182,420)

第一种思路

第二种思路

内存中的GoPiece对象



```
public class GoPiece {

    private Icon icon;
    private Position pos;

    public GoPiece(Icon icon, Position pos) {
        this.icon = icon;
        this.pos = pos;
    }
}
```

第二种思路

```
public interface Icon {  
  
    public void draw(Position pos);  
}
```

```
public class IconImpl implements Icon{  
  
    public IconImpl(String iconPath) {  
        //读取icon  
    }  
  
    public void draw(Position pos) {  
        //  
    }  
}
```

第二种思路

创建共享的Icon

```
public class IconFactory {  
  
    private static Icon whiteIcon  
        = new IconImpl("path to white icon");  
  
    private static Icon blackIcon  
        = new IconImpl("path to black icon");  
  
  
    private IconFactory() {  
    }  
    public static Icon getWhiteIcon() {  
        return whiteIcon;  
    }  
    public static Icon getBlackIcon() {  
        return blackIcon;  
    }  
}
```

使用

```

GoPiece goPiece1 = new GoPiece(
    IconFactory.getBlackIcon(),
    new Position(10,20));

GoPiece goPiece2 = new GoPiece(
    IconFactory.getBlackIcon(),
    new Position(200,20));

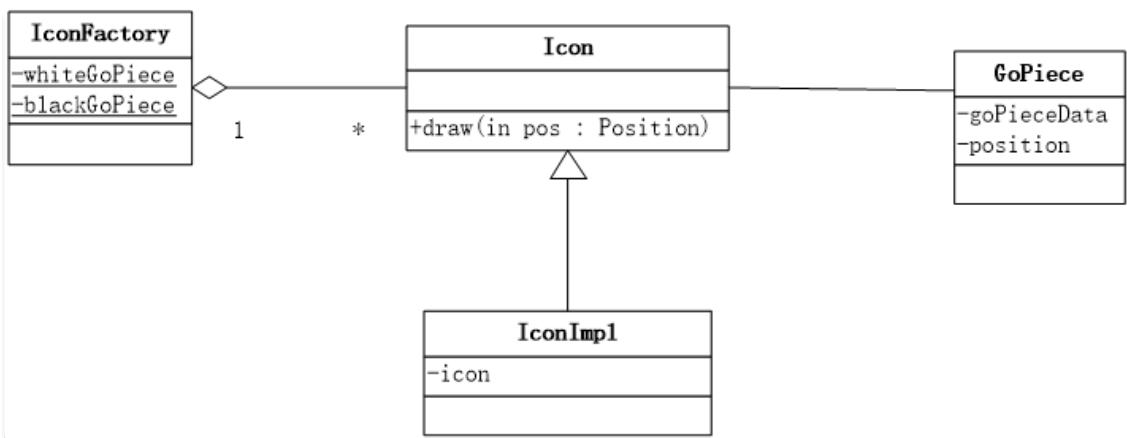
GoPiece goPiece3 = new GoPiece(
    IconFactory.getWhiteIcon(),
    new Position(300,20));

GoPiece goPiece4 = new GoPiece(
    IconFactory.getWhiteIcon(),
    new Position(200,60));

```

使用

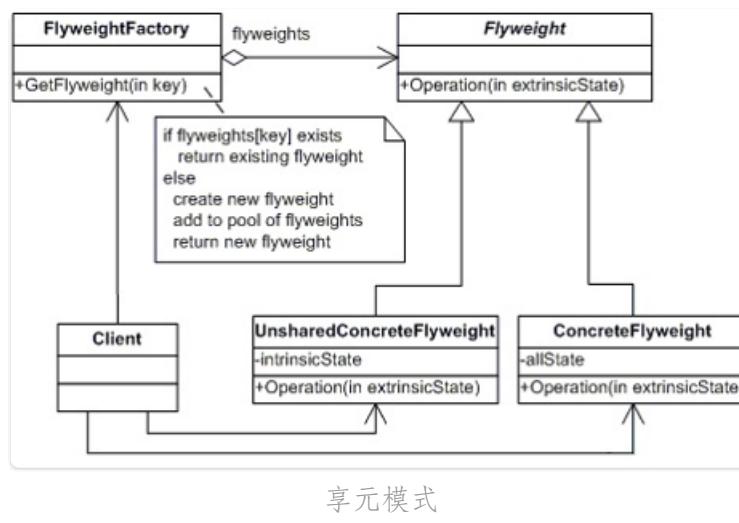
UML类图



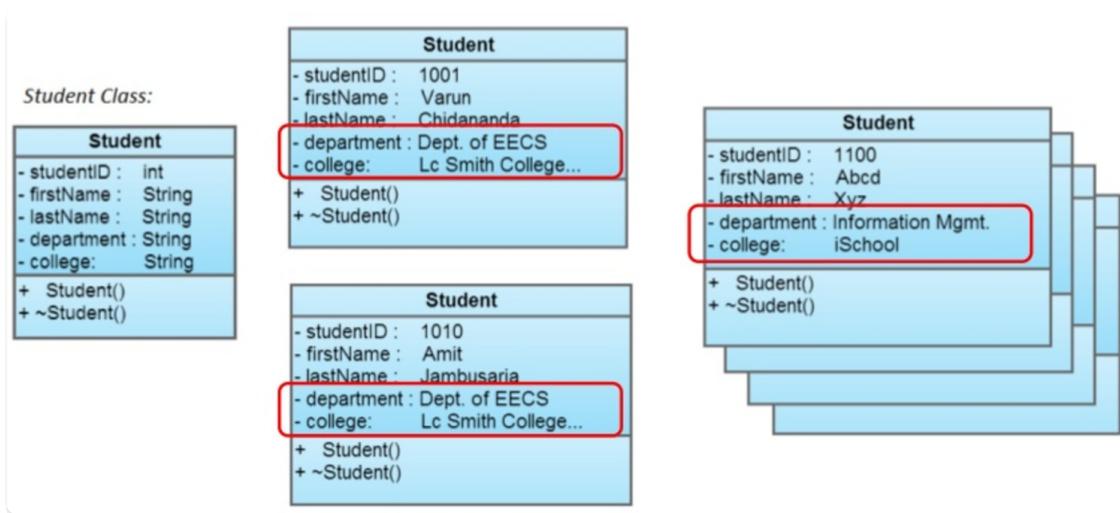
类图

Flyweight (享元) 模式

- 以共享的方式高效地支持大量细粒度对象的重用
- 共享的关键是区分内部状态(图标)和外部状态(位置)
 - 内部状态：可以共享，不随环境的变化而变化（图标）
 - 外部状态：不可以共享，随环境变化而变化（位置）



练习一下



Java String

```

String s1 = "hello";
String s2 = "hello";
String s3 = new String("hello");

System.out.println(s1 == s2) ; // true
System.out.println(s1 == s3) ; // false

String s4 = s3.intern();
System.out.println(s1 == s4) ; // true;
    
```

String

Java Boolean, Integer, Long

Boolean.valueOf :

```
public static final Boolean TRUE = new Boolean(true);
public static final Boolean FALSE = new Boolean(false);

public static Boolean valueOf(boolean b) {
    return (b ? TRUE : FALSE);
}
```

Integer.valueOf :

```
public static Integer valueOf(int i) {
    if (i >= IntegerCache.low && i <= IntegerCache.high)
        return IntegerCache.cache[i + (-IntegerCache.low)];
    return new Integer(i);
}
```

作业

作业1 自制Builder 创建xml 字符串

```
<order>
    <line-items>
        <line-item pid="P3677" qty="3" />
        <line-item pid="P9877" qty="10" />
    </line-items>
</order>
```

```
public class TagNode {
    private String tagName;
    private String tagValue;
    private List<TagNode> children = new ArrayList<>();
    private List<Attribute> attributes = new ArrayList<>();
    private static class Attribute{
        String name;
        String value;
    }
}
```

自制builder

```

TagNode orderTag = new TagNode("order");
TagNode lineItemsTag = new TagNode("line-items");
orderTag.add(lineItemsTag);
{
    TagNode lineItemTag = new TagNode("line-item");
    lineItemTag.setAttribute("pid", "P3677");
    lineItemTag.setAttribute("qty", "3");
    lineItemsTag.add(lineItemTag);
}
{
    TagNode lineItemTag = new TagNode("line-item");
    lineItemTag.setAttribute("pid", "P9877");
    lineItemTag.setAttribute("qty", "10");
    lineItemsTag.add(lineItemTag);
}

```

```

TagBuilder builder = new TagBuilder("order");

String xml = builder.addChild("line-items")
    .addChild("line-item").setAttribute("pid", "P3677").setAttribute("qty", "3")
    .addSibling("line-item").setAttribute("pid", "P9877").setAttribute("qty", "10")
    .toXML();

```

自制builder

作业2 从JDK中找出3个使用Singleton模式的类

参考链接：<https://stackoverflow.com/questions/1673841/examples-of-gof-design-patterns-in-javas-core-libraries?rq=1>

作业3 实现装饰者

```

public interface Email {
    public String getContent();
}

public class EmailImpl implements Email {
    private String content;

    public EmailImpl(String content) {
        this.content = content;
    }
    public String getContent() {
        return content;
    }
}

```

装饰者

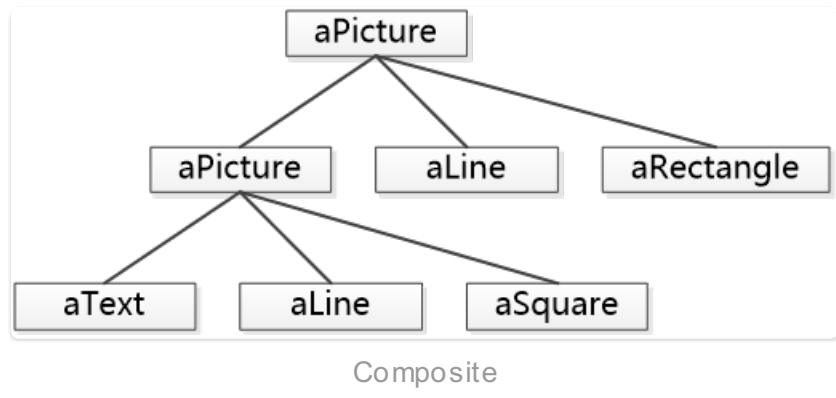
- 需求1

如果是对外发送的邮件，需要在邮件的末尾增加上公司的声明：本邮件仅为个人观点，并不代表公司立场

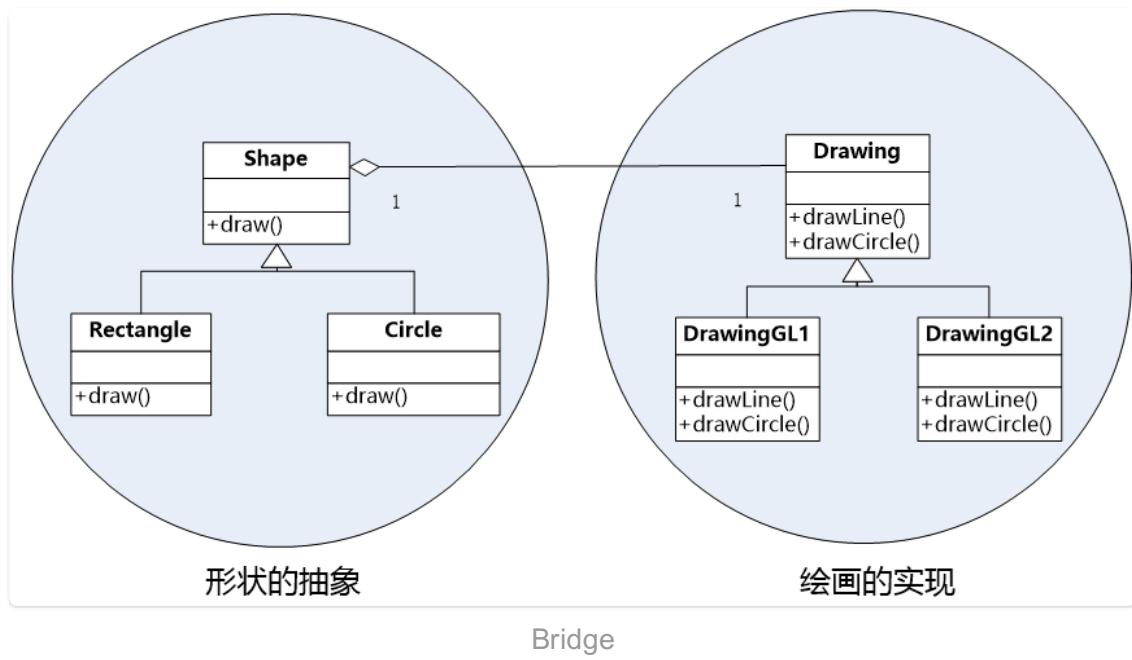
- 需求2

对邮件内容加密

作业4 实现Composite



作业5 实现Bridge



Bridge