理解设计模式(Android)

自我介绍

- 姓名：冯建
- 日期：2016年8月

领悟之道

- 重点是思想
- 其次是经验
- 难点是本质，我们应该做一个什么样的软件?

面向对象

- 什么是对象?
- 什么是封装，继承和多态?
- 与设计模式的关系

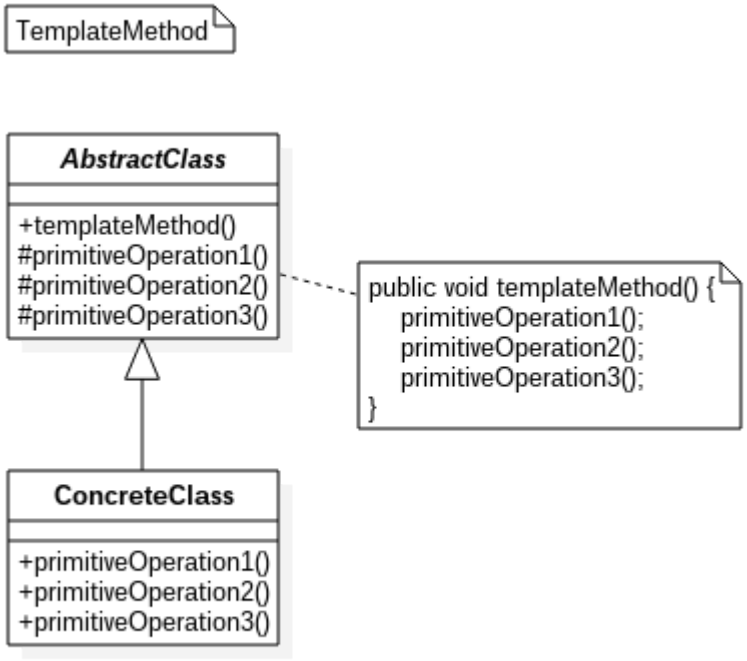
设计原则

- SRP：单一职责原则
- OCP：开闭原则
- LSP：里氏替换原则
- DIP：依赖倒置原则
- ISP：接口隔离眼则
- LKP：迪米特原则

热身模式

模板方法模式

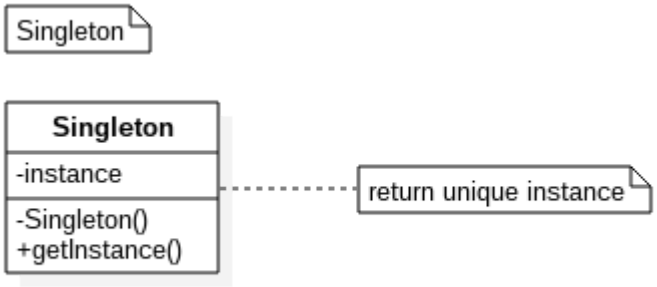
定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。TemplateMethod 使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。



注意：理解用继承去分离算法骨架以及分离的本质

单例模式

保证一个类仅有一个实例，并提供一个访问它的全局访问点。



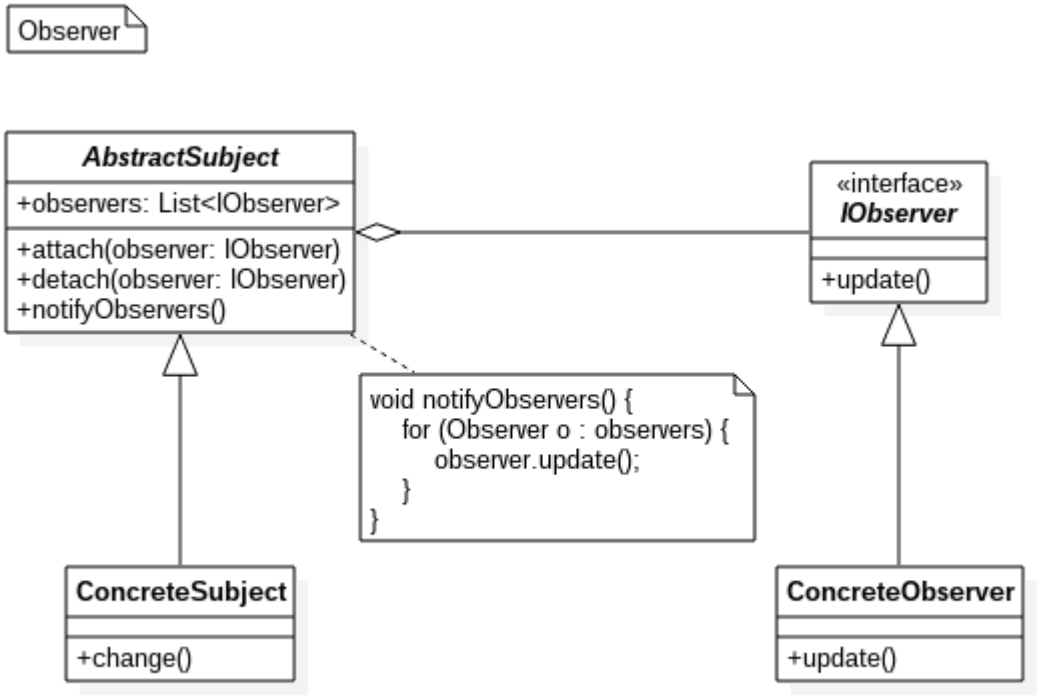
双重检查锁定与延迟初始化:

http://www.infoq.com/cn/articles/double-checked-locking-with-delay-initialization/

注意：难点在于保证二字，大家在面试的时候尽量把这个难点说清楚可以加分

观察者模式

定义对象间的一种一对多的依赖关系,当一个对象的状态发生改变时, 所有依赖于它的对象都得到通知并被自动更新。



注意：理解观察者模式的重大意义并能够自己去实现这个注册-被通知的模型

再看设计原则

- SRP：单一职责原则
- OCP：开闭原则
- LSP：里氏替换原则
- DIP：依赖倒置原则
- ISP：接口隔离眼则
- LKP：迪米特原则

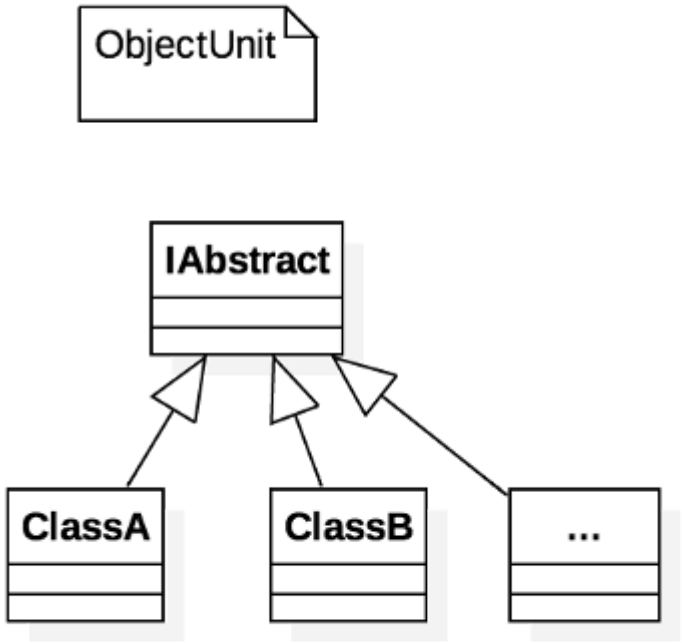
OU结构

ObjectUnit，面向对象单元

OU结构的核心在于：抽象出稳定的IAbstract，保证可复用性，稳定性；能够扩展新的实现，保证扩展性、灵活性。

使用OU结构将会自带面向对象带来的优势。

用OU结构去理解设计模式，会简化设计模式。

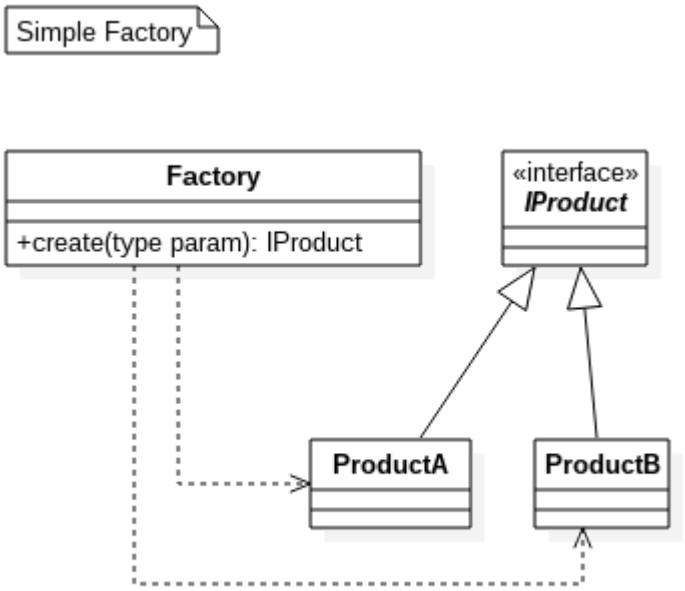


设计模式分类

- 创建型模式
  - 与对象的创建有关。
- 结构型模式
  - 处理类或对象的组合。
- 行为型模式
  - 对类或对象怎样交互和怎样分配职责进行描述。

创建型模式

简单工厂



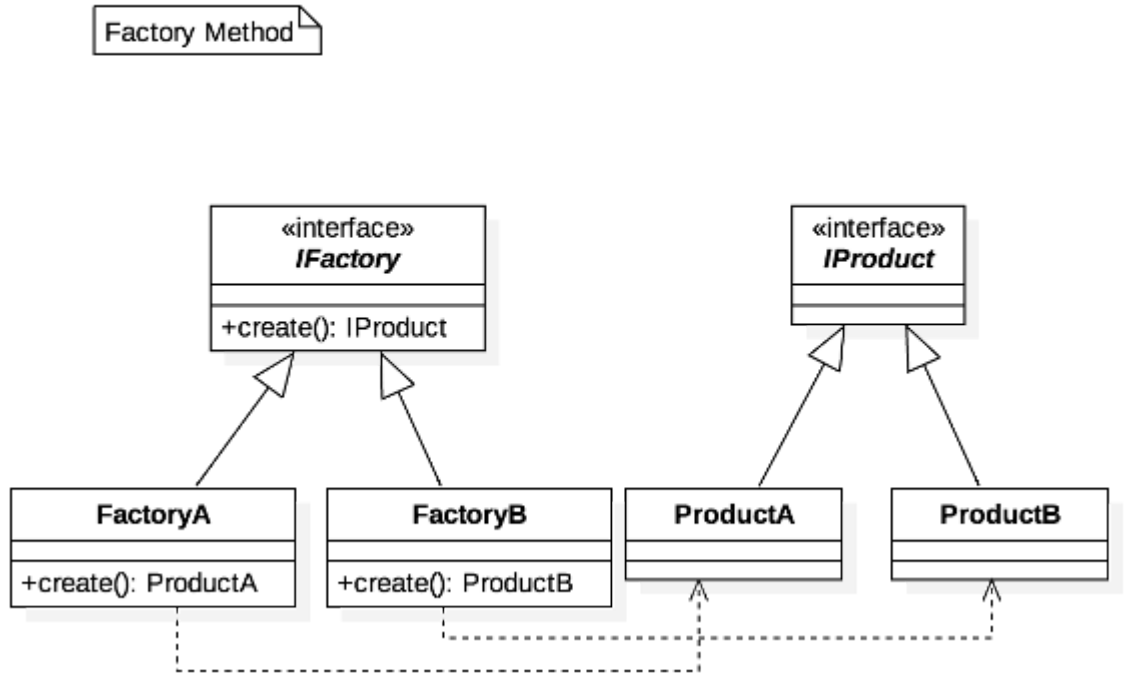
注意：理解直接创建对象和用工厂创建对象的区别所在

代码片段

```
 1 public class Factory {
 2
 3     public static final int TYPE_A = 1;
 4     public static final int TYPE_B = 2;
 5
 6     public IProduct create(int type) {
 7         if (type == TYPE_A) {
 8             return new ProductA();
 9         } else if (type == TYPE_B) {
10             return new ProductB();
11         }
12         return null;
13     }
14 }
```

工厂方法

定义一个用于创建对象的接口，让子类决定实例化哪一个类。Factory Method 使一个类的实例化延迟到其子类。



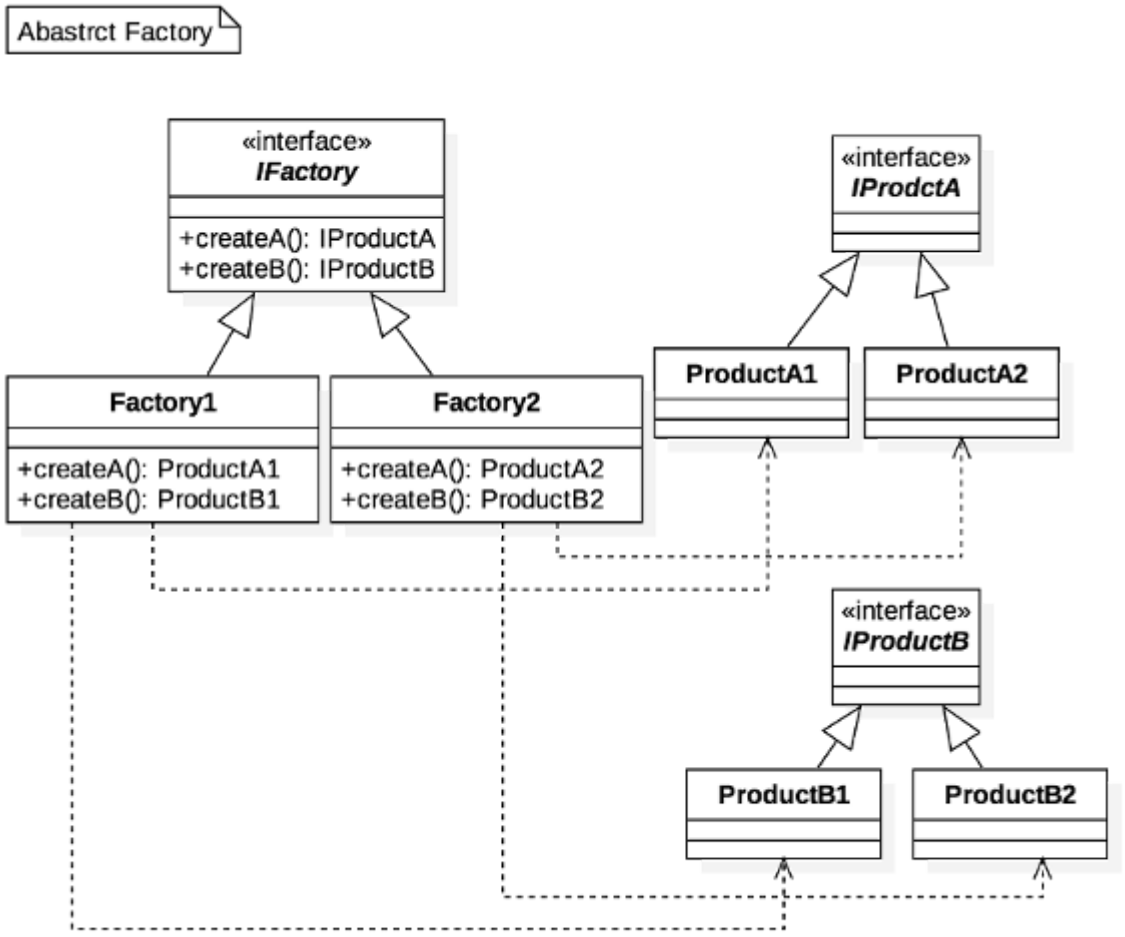
注意：学会把if-else结构转化为OU结构

代码片段

```
 1 public interface IProduct {
 2 }
 3 public class ProductA implements IProduct {
 4 }
 5 public class ProductB implements IProduct {
 6 }
 7 public interface IFactory {
 8     IProduct create();
 9 }
10 public class FactoryA implements IFactory {
11     @Override
12     public IProduct create() {
13         return new ProductA();
14     }
15 }
16 public class FactoryB implements IFactory {
17     @Override
18     public IProduct create() {
19         return new ProductB();
20     }
21 }
```

抽象工厂

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。

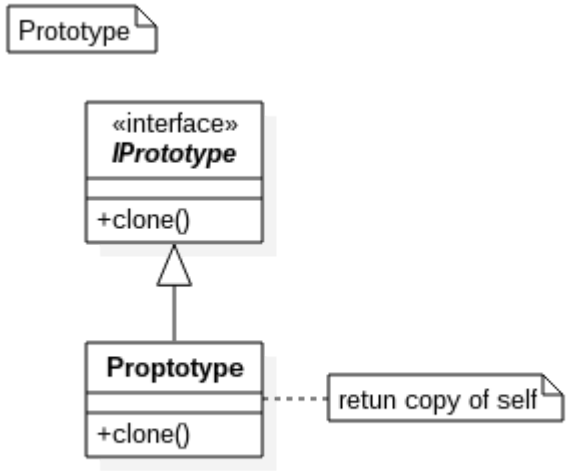


代码片段

```
 1 public interface IFactory {
 2     IProductA createA();
 3     IProductB createB();
 4 }
 5 public class Factory1 implements IFactory {
 6     @Override
 7     public IProductA createA() {
 8         return new ProductA1();
 9     }
10 
11     @Override
12     public IProductB createB() {
13         return new ProductB1();
14     }
15 }
16 public class Factory2 implements IFactory {
17     @Override
18     public IProductA createA() {
19         return new ProductA2();
20     }
21 
22     @Override
23     public IProductB createB() {
24         return new ProductB2();
25     }
26 }
```

原型模式

用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象。



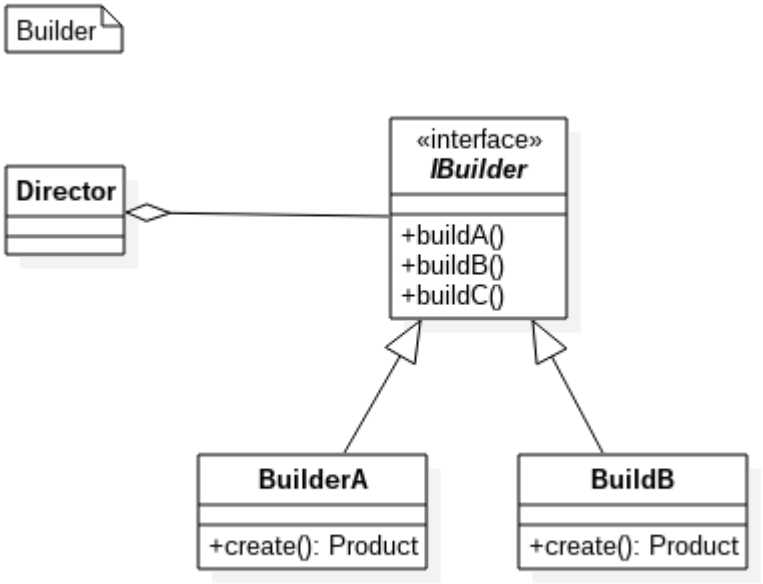
注意：理解深浅拷贝的区别以及实现一个clone的意义所在

代码片段

```
 1 public class User implements Cloneable {
 2
 3     private String name;
 4     private int age;
 5
 6     public User(String name, int age) {
 7         this.name = name;
 8         this.age = age;
 9     }
10
11     @Override
12     public Object clone() throws CloneNotSupportedException {
13         User user = new User(name, age);
14         // user.name = "cloner name";
15         return user;
16     }
17 }
```

建造者模式

将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。



注意：理解建造者模式的本质就是分离构建，用OU结构

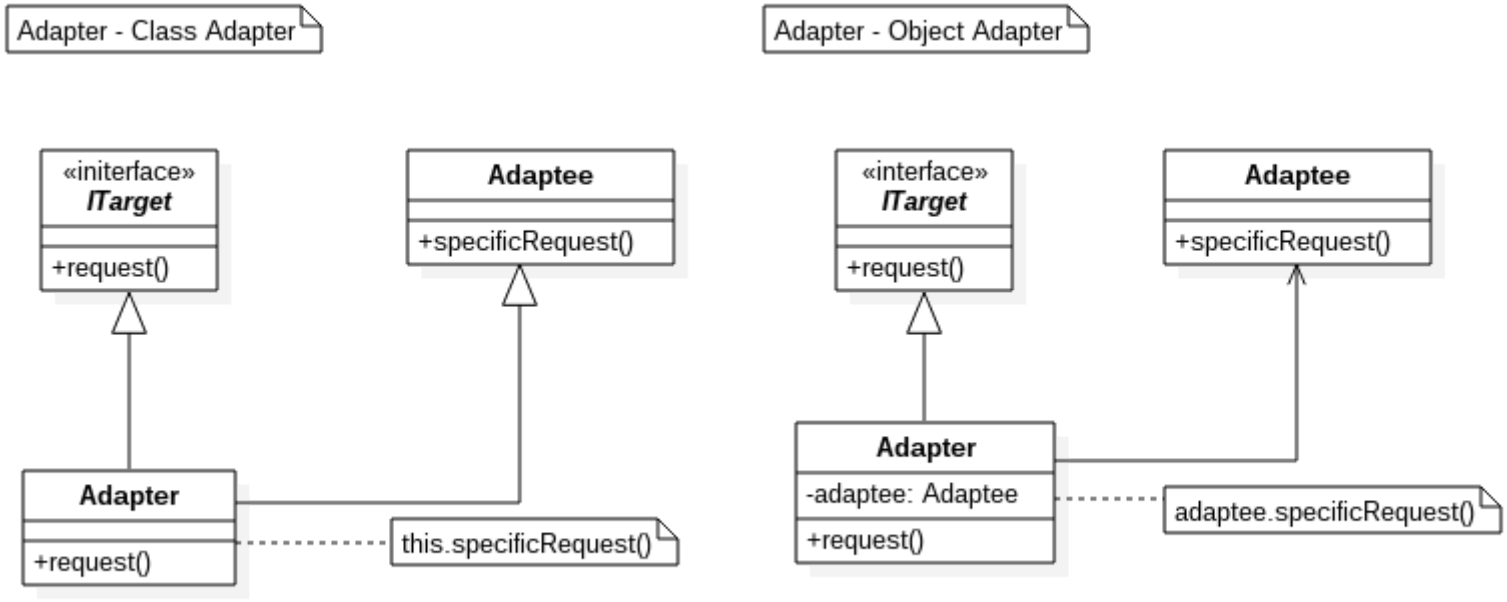
代码片段

```
 1 public class Product {
 2
 3     public void otherMethodA() {
 4     }
 5
 6     public void otherMethodB() {
 7     }
 8
 9     public void buildStepA() {
10     }
11     public void buildStepB() {
12     }
13     public void buildStepC() {
14     }
15 }
16 public interface IBuilder {
17     void buildStepA();
18     void buildStepB();
19     void buildStepC();
20 }
21 public class BuilderA implements IBuilder {
22     private Product product = new Product();
23
24     public void buildStepA() {
25         product.buildStepA();
26     }
27
28     public void buildStepB() {
29         product.buildStepB();
```

结构型模式

适配器模式

将一个类的接口转换成客户希望的另外一个接口。Adapter 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。



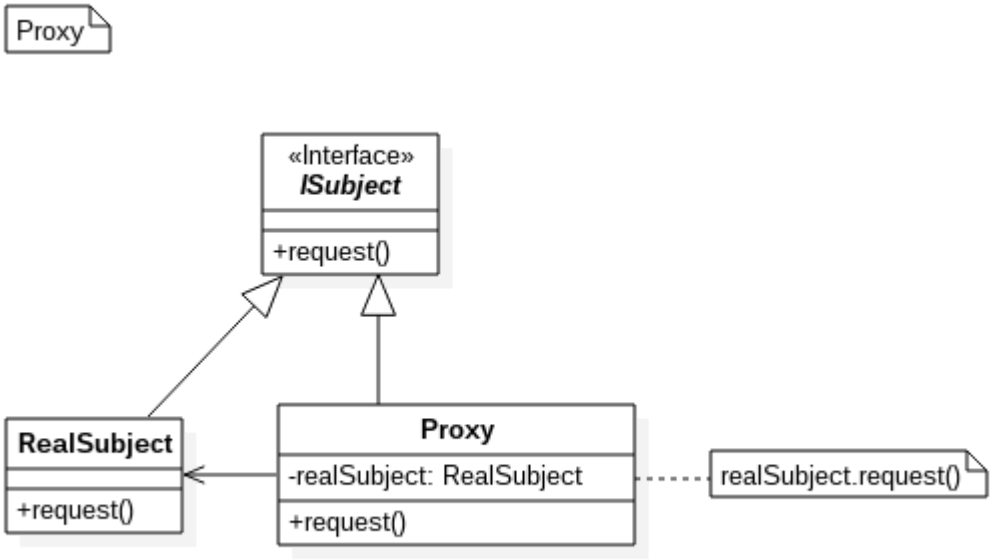
注意：比较类继承和对象组合的不同

代码片段

```
public class Adaptee {
    public void specificRequest() {
    }
}
// 类适配器
public class Adapter extends Adaptee {
    public void request() {
        super.specificRequest();
    }
}
// 对象适配器
public class Adapter {

    private Adaptee adaptee;

    public void request() {
        adaptee.specificRequest();
    }
}
```

代理模式

为其他对象提供一种代理以控制对这个对象的访问。



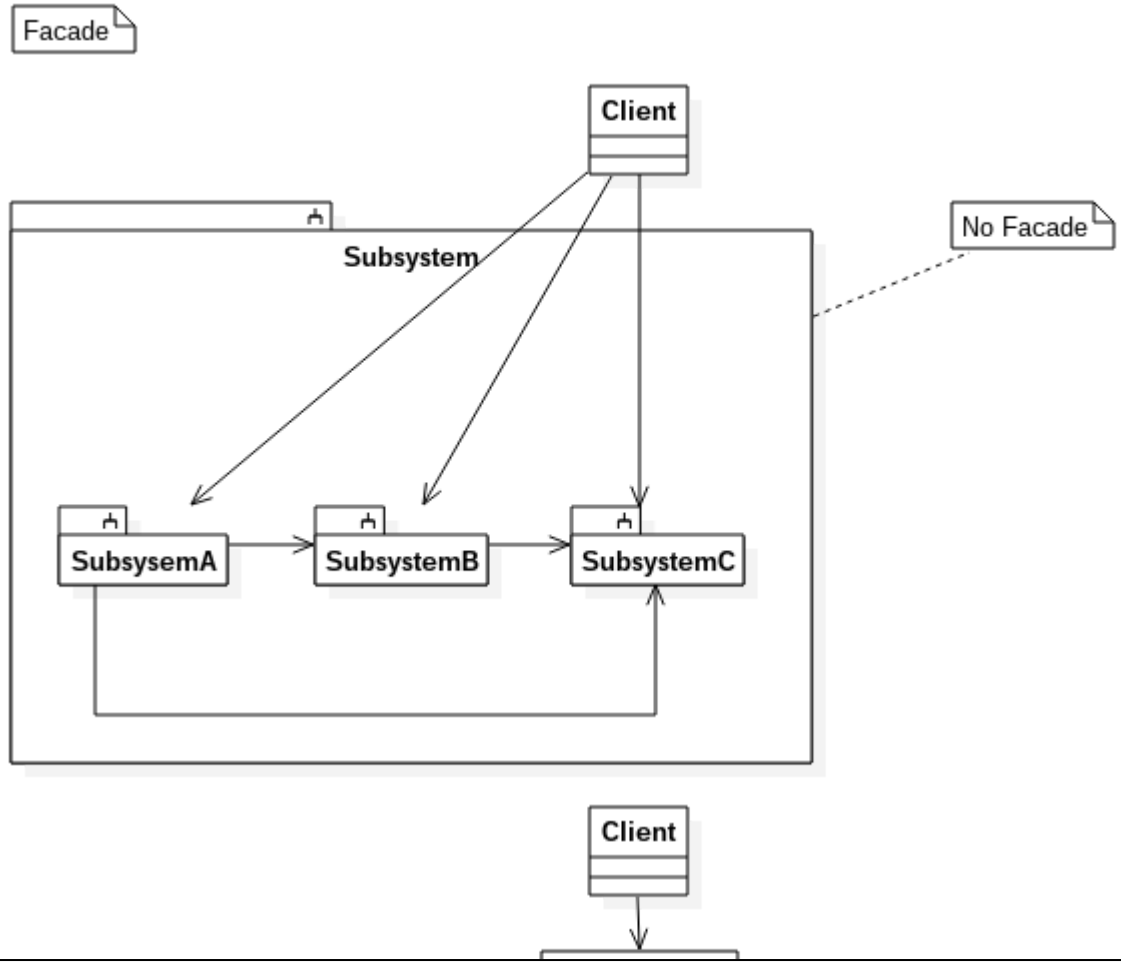
注意：理解控制访问对象的意义，以及保持代理类对外的统一性

代码片段

```
 1 public interface ISubject {
 2     void request();
 3 }
 4 public class RealSubject implements ISubject {
 5     @Override
 6         public void request() {
 7         }
 8 }
 9 public class Proxy implements ISubject {
10
11     private RealSubject realSubject;
12
13     @Override
14         public void request() {
15             if (realSubject == null) {
16                 realSubject = new RealSubject();
17             }
18             // prerequest() code
19             realSubject.request();
20             // postrequest() code
21         }
22 }
```

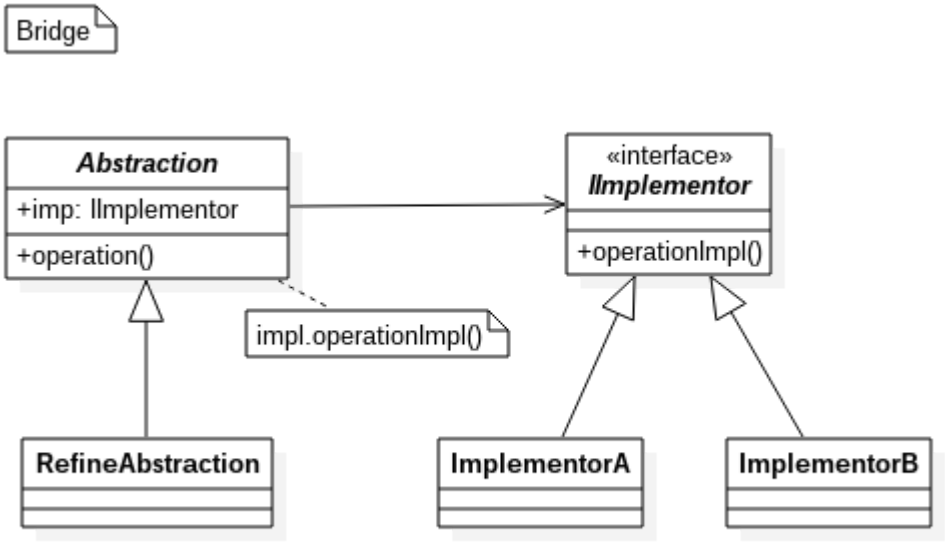
门面模式

为子系统中的一组接口提供一个一致的界面，Facade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。



桥接模式

将抽象部分与它的实现部分分离，使它们都可以独立地变化。



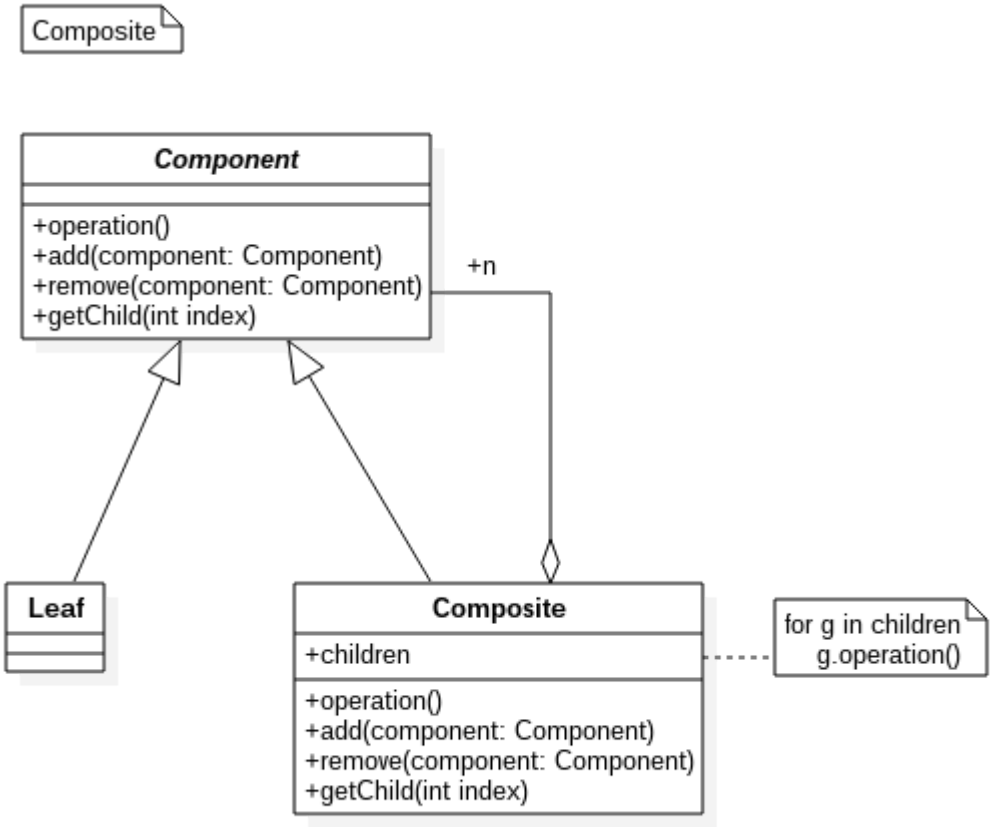
注意：理解桥接的是二维的变化，考虑多维变化的实现

代码片段

```
 1 public interface IImplementor {
 2     void operationImpl();
 3 }
 4 public class ImplementorA implements IImplementor {
 5     public void operationImpl() {
 6     }
 7 }
 8 public class ImplementorB implements IImplementor {
 9     public void operationImpl() {
10     }
11 }
12 public abstract class Abstraction {
13     public IImplementor imp;
14     public void operation() {
15         imp.operationImpl();
16     }
17 }
18 public class RefineAbstraction extends Abstraction {
19 }
```

组合模式

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



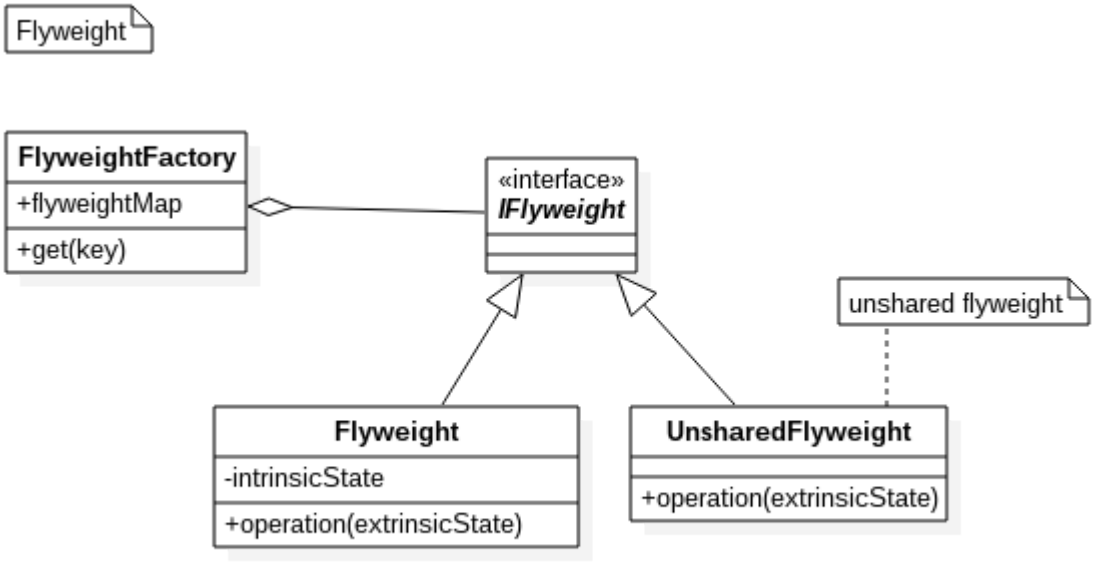
注意：理解如何实现去除部分和整体的差异性

代码片段

```
 1 public abstract class Component {
 2     public void operation() {
 3         // default impl code
 4     }
 5 }
 6 public class Composite extends Component {
 7
 8     private List<Component> components;
 9
10     @Override
11         public void operation() {
12             for (Component component : components) {
13                 component.operation();
14             }
15         }
16
17     public void add(Component component) {
18         components.add(component);
19     }
20
21     public void remove(Component component) {
22         components.remove(component);
23     }
24
25     public Component getChild(int index) {
26         return components.get(index);
27     }
28 }
29 public class Leaf extends Component {
```

享元模式

运用共享技术有效地支持大量细粒度的对象。



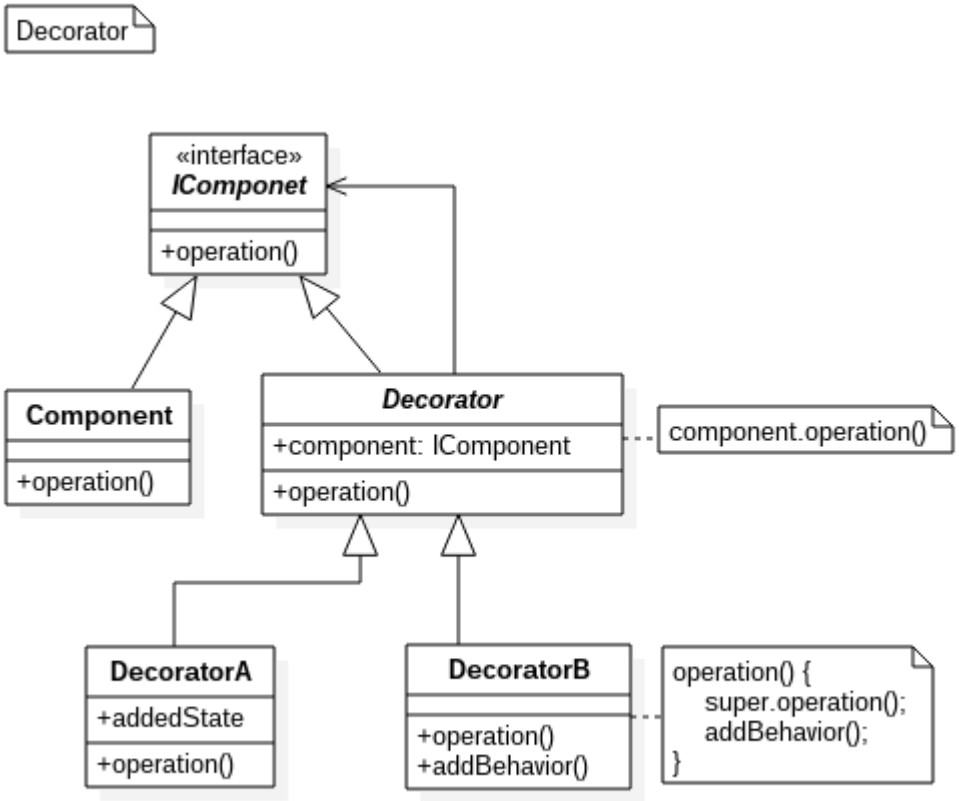
注意：能够自己写享元工厂的实现代码

代码片段

```
public interface IFlyweight {
    void operation(String extrinsicState);
}
public class Flyweight implements IFlyweight {

    private String intrinsicState1; // 内部状态，可供全局共享
    private String intrinsicState2; // 内部状态，可供全局共享
    private String extrinsicState; // 外部状态，由外部负责

    public Flyweight(String intrinsicState1, String intrinsicState2) {
        this.intrinsicState1 = intrinsicState1;
        this.intrinsicState2 = intrinsicState2;
    }

    @Override
        public void operation(String extrinsicState) {
            this.extrinsicState = extrinsicState;
        }
}
public class FlyweightFactory {
    private Map<String, IFlyweight> flyweightMap = new HashMap<>();

    // key应该根据内部状态生成
    public IFlyweight get(String intrinsicState1,
                          String intrinsicState2) {
        String key = intrinsicState1 + "-" + intrinsicState2;
        if (flyweightMap.get(key) == null) {
            IFlyweight flyweight =
                new Flyweight(intrinsicState1, intrinsicState2);
```

装饰者模式

动态地给一个对象添加一些额外的职责。就增加功能来说，Decorator 模式相比生成子类更为灵活。



注意：理解比生成子类的灵活性的体现

代码片段

```
public class Component implements IComponent {
    @Override
        public void operation() {
            System.out.println("Component");
        }
}
public abstract class Decorator implements IComponent {

    protected IComponent component;
    public Decorator(IComponent component) {
        this.component = component;
    }

    @Override
        public void operation() {
        }
}
public class DecoratorA extends Decorator {

    public DecoratorA(IComponent component) {
        super(component);
    }

    @Override
        public void operation() {
            // Feature A code here
            // ...
            component.operation();
            // Feature A code here
```

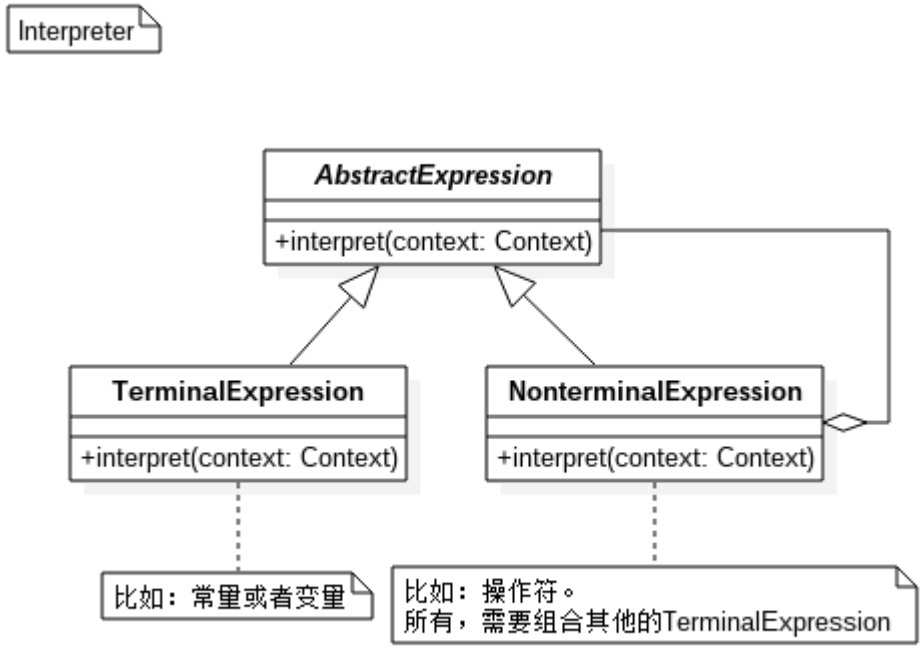
行为型模式

解释器模式

给定一个语言，定义它的文法的一种表示，并定义一个解释器，这个解释器使用该表示来解释语言中的句子。

(x+y) * 5

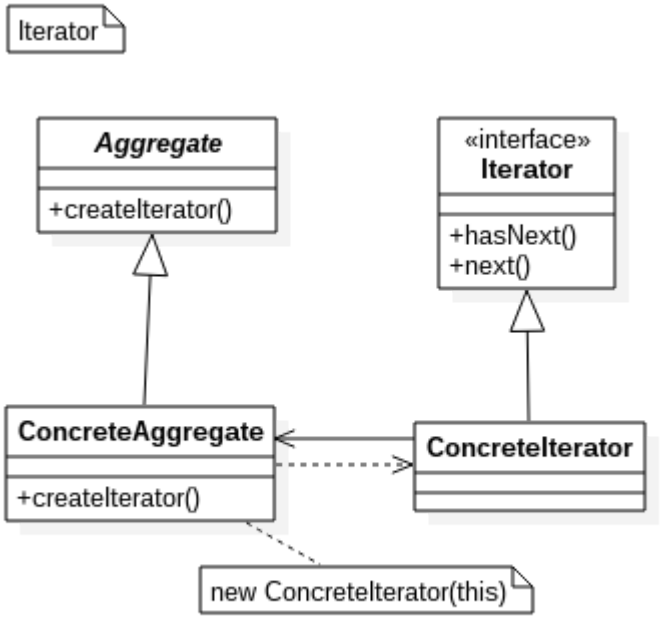
new Multi(new And(new Var(“x”),new Var(“y”)) , new Consant(“5”));



注意：会把这个表达式转化为面向对象的表达

迭代器模式

提供一种方法顺序访问一个聚合对象中各个元素，而又不需暴露该对象的内部表示。



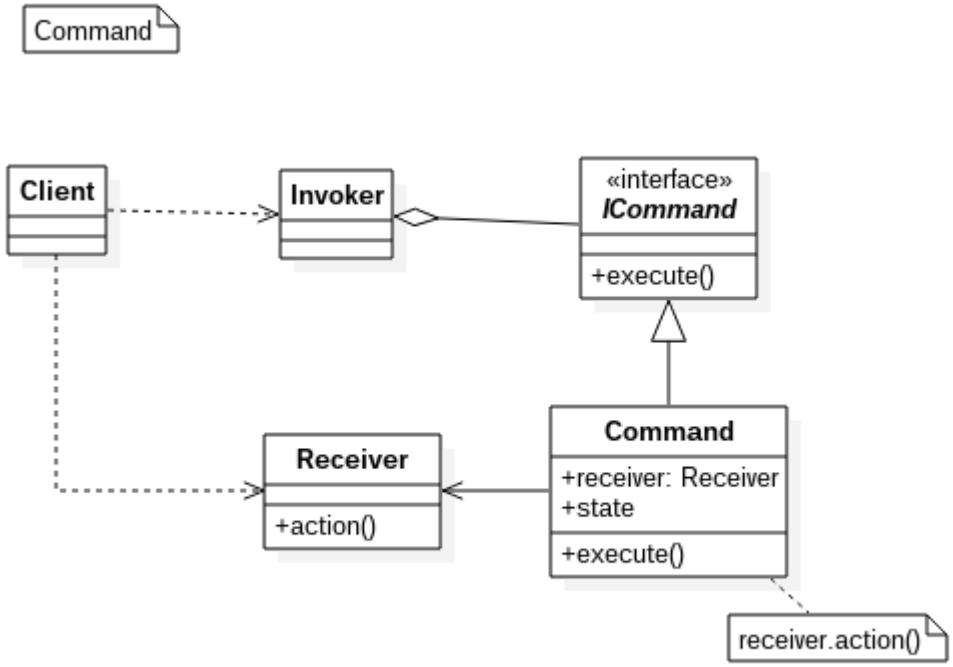
注意：实现迭代器接口

代码片段

```
1 public interface Iterator<E> {
2     boolean hasNext();
3     E next();
4 }
```

命令模式

将一个请求封装为一个对象，从而使你可用不同的请求对客户进行参数化；对请求排队或记录请求日志，以及支持可撤消的操作。



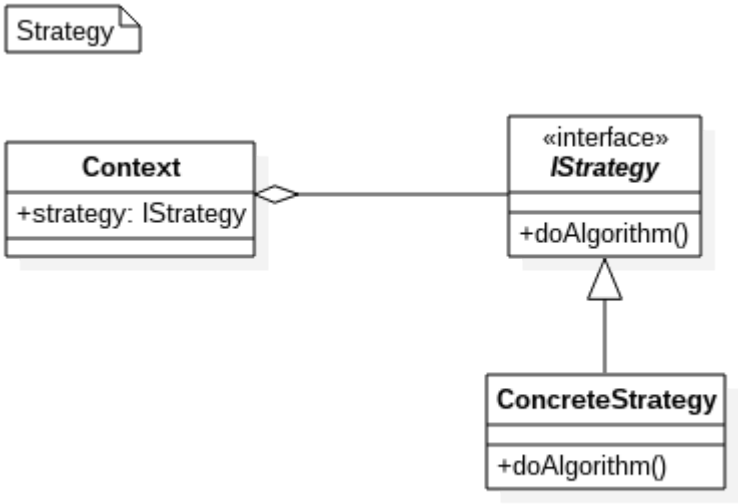
注意：理解代码块封装在面向对象中的意义

代码片段

```
 1 public interface ICommand {
 2     void execute();
 3 }
 4 public class Receiver {
 5     public void action() {
 6     }
 7 }
 8 public class Command implements ICommand {
 9     // 注入真正的行为对象
10     private Receiver receiver;
11     public void setReceiver(Receiver receiver) {
12         this.receiver = receiver;
13     }
14 
15     @Override
16     public void execute() {
17         receiver.action();
18     }
19 }
```

策略模式

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。本模式使得算法可独立于使用它的客户而变化。



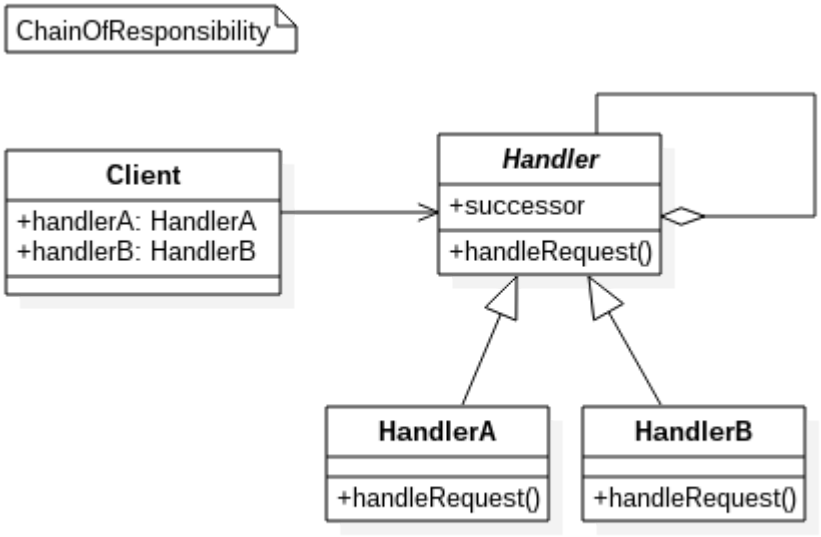
注意：理解策略模式和命令模式的区别

代码片段

```
 1 public interface IStrategy {
 2     int doAlgorithm(int... array);
 3 }
 4 public class StrategyA implements IStrategy {
 5     @Override
 6         public int doAlgorithm(int... array) {
 7             return array[0];
 8         }
 9 }
10 public class StrategyB implements IStrategy {
11     @Override
12         public int doAlgorithm(int... array) {
13             return array[array.length - 1];
14         }
15 }
16 public class Context {
17 
18     private IStrategy strategy;
19     public void setStrategy(IStrategy strategy) {
20         this.strategy = strategy;
21     }
22 
23     public int getData(int... array) {
24         return strategy.doAlgorithm(array);
25     }
26 
27     public static void main(String[] args) {
28         int[] array = {1, 2, 3, 4};
29 
```

职责链模式

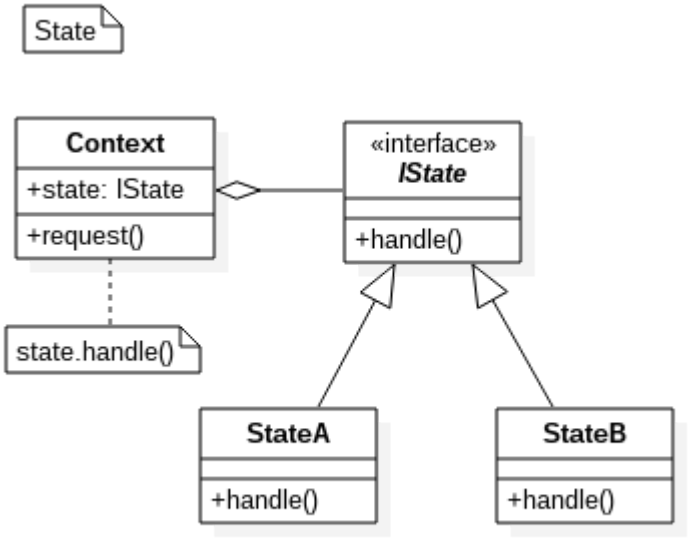
使多个对象都有机会处理请求，从而避免请求的发送者和接收者之间的耦合关系。将这些对象连成一条链，并沿着这条链传递该请求，直到有一个对象处理它为止。



注意：理解如何分离请求处理逻辑

状态模式

允许一个对象在其内部状态改变时改变它的行为。对象看起来似乎修改了它的类。



注意：理解如何分离状态和行为的自动转换

代码片段

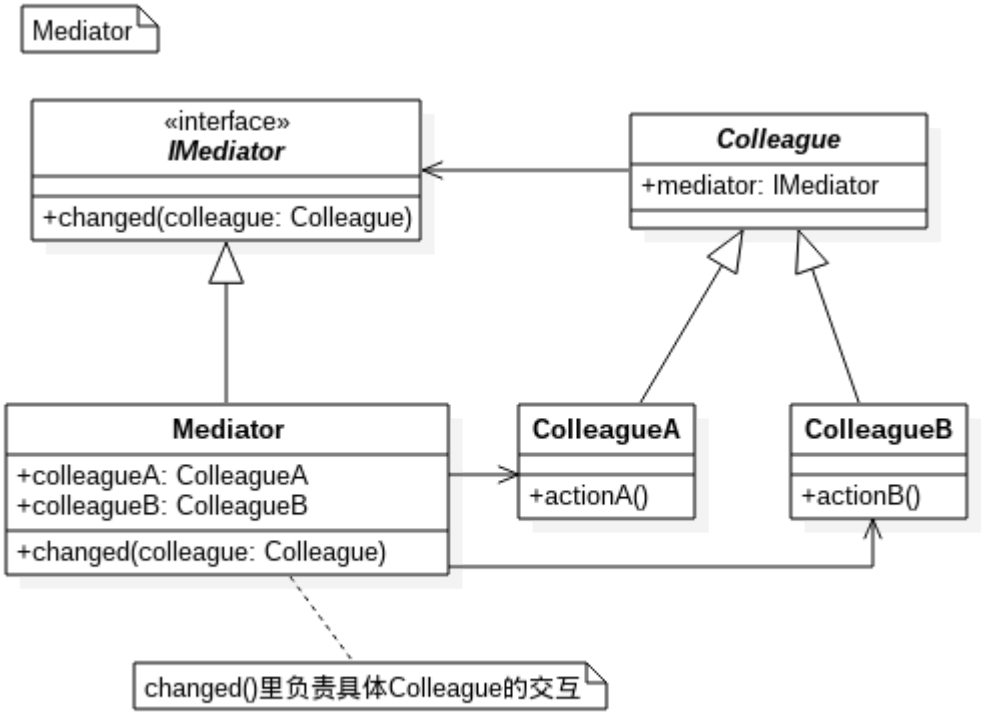
```
// 开关
public class Switch {
    public static final int ON = 0;
    public static final int OFF = 1;

    public int state = ON;

    public void change() {
        if (state == ON) {
            state = OFF;
        } else if (state == OFF) {
            state = ON;
        }
    }
}
```

代码片段

```
 1 public interface IState {
 2     void handle(Context context);
 3 }
 4 public class StateA implements IState {
 5     @Override
 6         public void handle(Context context) {
 7             context.setState(new StateB());
 8         }
 9 }
10 public class StateB implements IState {
11     @Override
12         public void handle(Context context) {
13             context.setState(new StateC());
14         }
15 }
16 public class StateC implements IState {
17     @Override
18         public void handle(Context context) {
19             // if it has StateD, change to StateD.
20             // or game over.
21         }
22 }
23 public class Context {
24     public IState state;
25 
26     public IState getState() {
27         return state;
28     }
29 
```

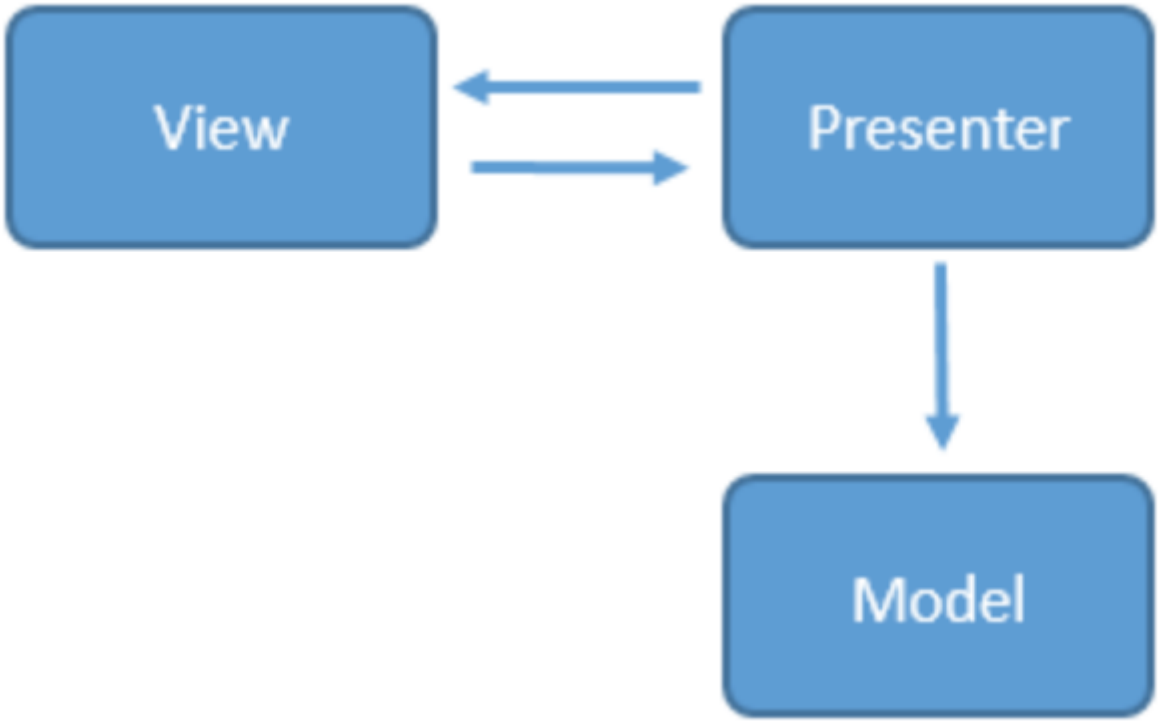
中介者模式

用一个中介对象来封装一系列的对象交互。中介者使各对象不需要显式地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互。



注意：理解中介者把网状交互转化为了星状交互以及这两者的区别

MVP图示

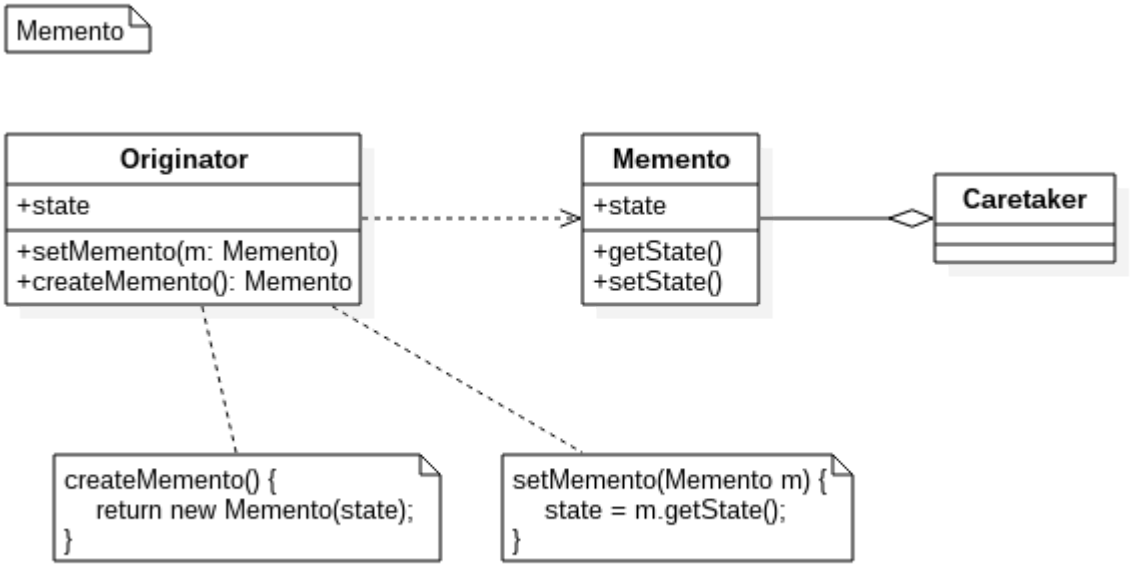


代码片段

```
public class Mediator implements IMediator {

    // 注入具体Colleague
    public ColleagueA colleagueA;
    public ColleagueB colleagueB;
    public ColleagueC colleagueC;
    public void setColleagueA(ColleagueA colleagueA) {
        this.colleagueA = colleagueA;
    }
    public void setColleagueB(ColleagueB colleagueB) {
        this.colleagueB = colleagueB;
    }
    public void setColleagueC(ColleagueC colleagueC) {
        this.colleagueC = colleagueC;
    }

    /**
     * 处理交互逻辑的枢纽就在这里
     * @param colleague
     */
    @Override
    public void changed(Colleague colleague) {
        if (colleague == colleagueB) {
            colleagueB.actionB();
        } else if (colleague == colleagueC) {
            colleagueC.actionC();
        }
    }
}
```

备忘录模式

在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态。这样以后就可将该对象恢复到原先保存的状态。



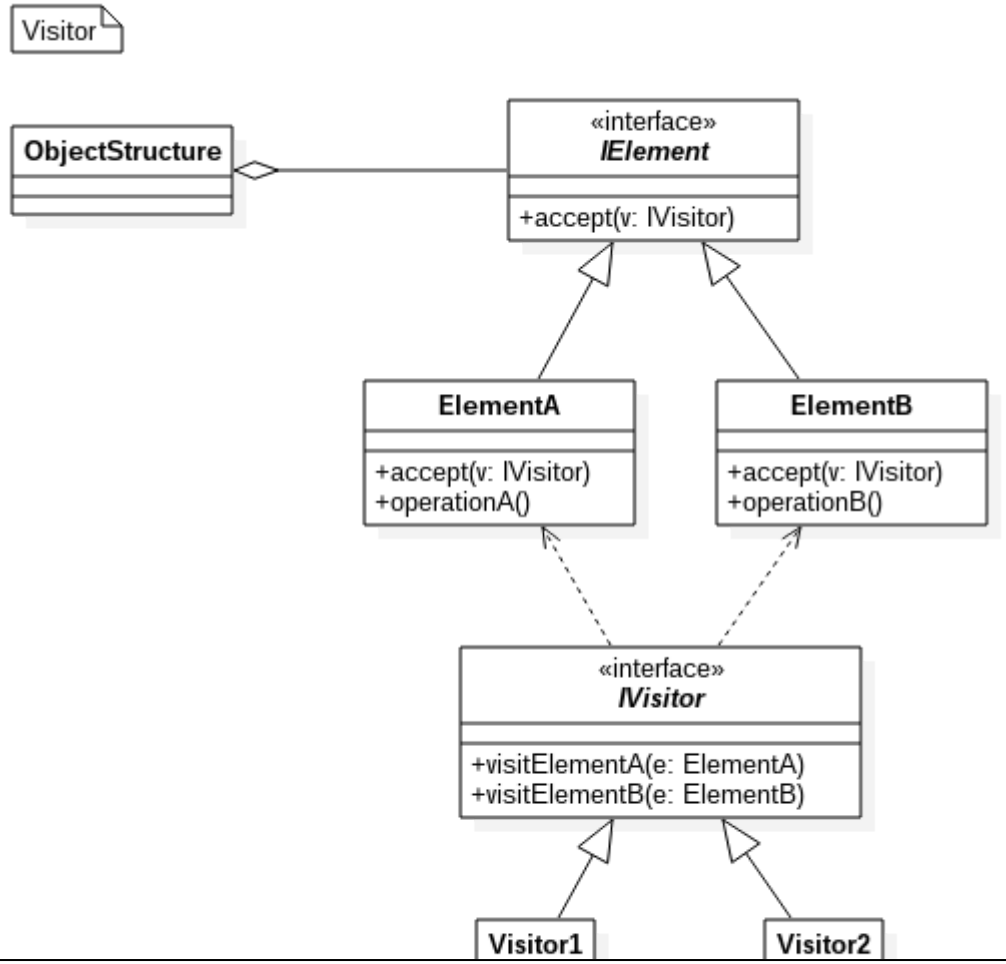
注意：学会不破坏封装性的前提下实现状态的存档实现

代码片段

```
public class Memento {
    private String state;

    public String getState() {
        return state;
    }

    public void setState(String state) {
        this.state = state;
    }
}
public class Originator {
    // state可以是任何形式，这里用String做最简化示例
    private String state;

    public Memento createMemento() {
        Memento memento = new Memento();
        memento.setState(state);
        return memento;
    }

    public void setMemento(Memento memento) {
        state = memento.getState();
    }
}
// 外部化类：分离持久化职责的作用
public class Caretaker {

    private Memento memento;
```

访问者模式

表示一个作用于某对象结构中的各元素的操作。它使你可以在不改变各元素的类的前提下定义作用于这些元素的新操作。



代码片段

```
for (IElement element : elements) {
    element.accept(new Visitor());
    if (element instanceof ElementA) {
        ((ElementA) element).a();
    } else if (element instanceof  ElementB) {
        ((ElementB) element).b();
    }
}

// 访问者
public class Visitor {
    public void visitElementA(ElementA elementA) {
        elementA.a();
    }
    public void visitElementB(ElementB elementB) {
        elementB.b();
    }
}
```

代码片段

```
 1 public interface IElement {
 2     void accept(Visitor visitor);
 3 }
 4 public class ElementA implements IElement{
 5 
 6     public void a() {
 7         System.out.println("AAAAAAAA");
 8     }
 9 
10     @Override
11         public void accept(Visitor visitor) {
12             visitor.visitElementA(this);
13         }
14 }
15 public class ElementB implements IElement {
16 
17     public void b() {
18         System.out.println("BBBBBBBBB");
19     }
20 
21     @Override
22         public void accept(Visitor visitor) {
23             visitor.visitElementB(this);
24         }
25 }
```

代码片段

```
1 // 如此的优雅
2 for (IElement element : elements) {
3     element.accept(new Visitor());
4 }
```

几点技巧

- 考虑用OU结构替换if-else结构
- 从意图的角度去理解设计模式，会事半功倍
- 有些设计模式是不同颗粒的相似形态
- 建立对象化思维，一切皆为对象
- 转移耦合到更稳定的间接层

作业

- 分析Retrofit中包含哪些设计模式
- 自行分析framework中的一个组件所包含的设计模式

最后

- 《设计模式 – 可复用面向对象软件的基础》
- 《冒号课堂 – 编程范式与OOP思想》
- 《Java与模式》
- Q我互动

谢谢大家