S.O.L.I.D 设计原则&设计模式

xieqiaoyun

S.O.L.I.D 设计原则

- 单一职责：分离关注点、指责专一，降低耦合度
- 开闭原则：对内保护修改，对外方便扩展；抽象可变部分以提升系统扩展
- 继承原则：遵循继承的语意和意图
- 接口隔离：减少不必要的继承，降低耦合度，使系统易于理解和使用
- 依赖反转：依赖于运行容器环境，减少与具体实现类的相互制约

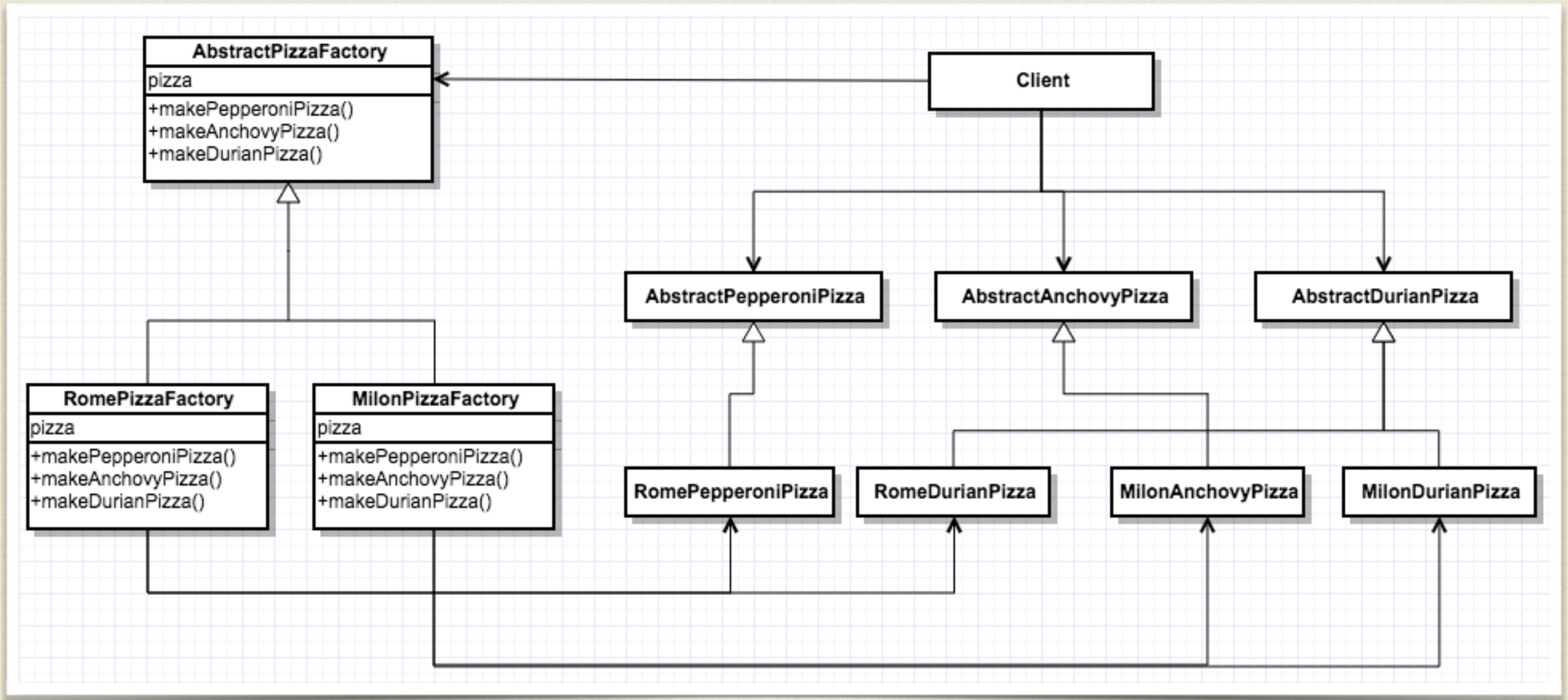
宗旨：实现低耦合、高凝聚力、易组合、依赖环境的可靠系统

设计模式全罗列

设计模式全罗列创建类设计模式关注抽象实例化过程

ABSTRACT FACTORY

- 提供接口运行时创建相关联产品族



代码示例：三个具体产品类

```
abstract class AbstractPepperoniPizza{}
class RomePepperoniPizza extends AbstractPepperoniPizza{
    RomePepperoniPizza(){
        System.out.println("create Rome Pepperoni pizza");
    }
}
class MilonPepperoniPizza extends AbstractPepperoniPizza{
    MilonPepperoniPizza(){
        System.out.println("create Milon Pepperoni pizza'
    }
}
```

自己负责实例化本类产品

```
abstract class AbstractAchrovyPizza{}
class RomeAchrovyPizza extends AbstractAchrovyPizza{
    RomeAchrovyPizza(){
        System.out.println("create Rome Achrovy pizza");
    }
}
class MilonAchrovyPizza extends AbstractAchrovyPizza{
    MilonAchrovyPizza(){
        System.out.println("create Milon Achrovy pizza");
    }
}
```

```
abstract class AbstractDurianPizza {}
class RomeDurianPizza extends AbstractDurianPizza{
    RomeDurianPizza(){
        System.out.println("create Rome Durian pizza");
    }
}
class MilonDurianPizza extends AbstractDurianPizza{
    MilonDurianPizza(){
        System.out.println("create Milon Durian pizza");
    }
}
```

代码示例：两个工厂

```
class RomePizzaFactory extends
AbstractPizzaFactory{
    @Override
    public AbstractPepperoniPizza
makePeppenoriPizza(){
        return new RomePepperoniPizza();
    }

    @Override
    public AbstractAchrovyPizza makeAchrovyPizza()
{
        return new RomeAchrovyPizza();
    }

    @Override
    public AbstractDurianPizza makeDurianPizza(){
        return new RomeDurianPizza();
    }
}
```

```
class MilonPizzaFactory extends
AbstractPizzaFactory{
    @Override
    public AbstractPepperoniPizza
makePeppenoriPizza(){
        return new MilonPepperoniPizza();
    }

    @Override
    public AbstractAchrovyPizza makeAchrovyPizza()
{
        return new MilonAchrovyPizza();
    }

    @Override
    public AbstractDurianPizza makeDurianPizza(){
        return new MilonDurianPizza();
    }
}
```

每个具体工厂类负责实例化子类

代码示例：维度扩大一倍，产品家族更宽泛

```
abstract class AbstractPizzaFactory{
    private static final RomePizzaFactory ROME_FACTORY = new RomePizzaFactory();
    private static final MilonPizzaFactory MILON_FACTORY = new MilonPizzaFactory();

    static AbstractPizzaFactory getFactory(String factoryName){
        AbstractPizzaFactory factory = null;
        if(factoryName.equals("Rome")){
            factory = ROME_FACTORY;
        } else if (factoryName.equals("Milon")){
            factory = MILON_FACTORY;
        }
        return factory;
    }

    public abstract AbstractPepperoniPizza makePeppenoriPizza();
    public abstract AbstractAchrovyPizza makeAchrovyPizza();
    public abstract AbstractDurianPizza makeDurianPizza();
}

public class AbstractFactoryDemo {
    public static void main(String[] args){
        AbstractPizzaFactory romeFactory = AbstractPizzaFactory.getFactory("Rome");
        romeFactory.makeDurianPizza();

        AbstractPizzaFactory milonFactory = AbstractPizzaFactory.getFactory("Milon");
        milonFactory.makePeppenoriPizza();
    }
}
```

每个具体工厂抽象类是单例

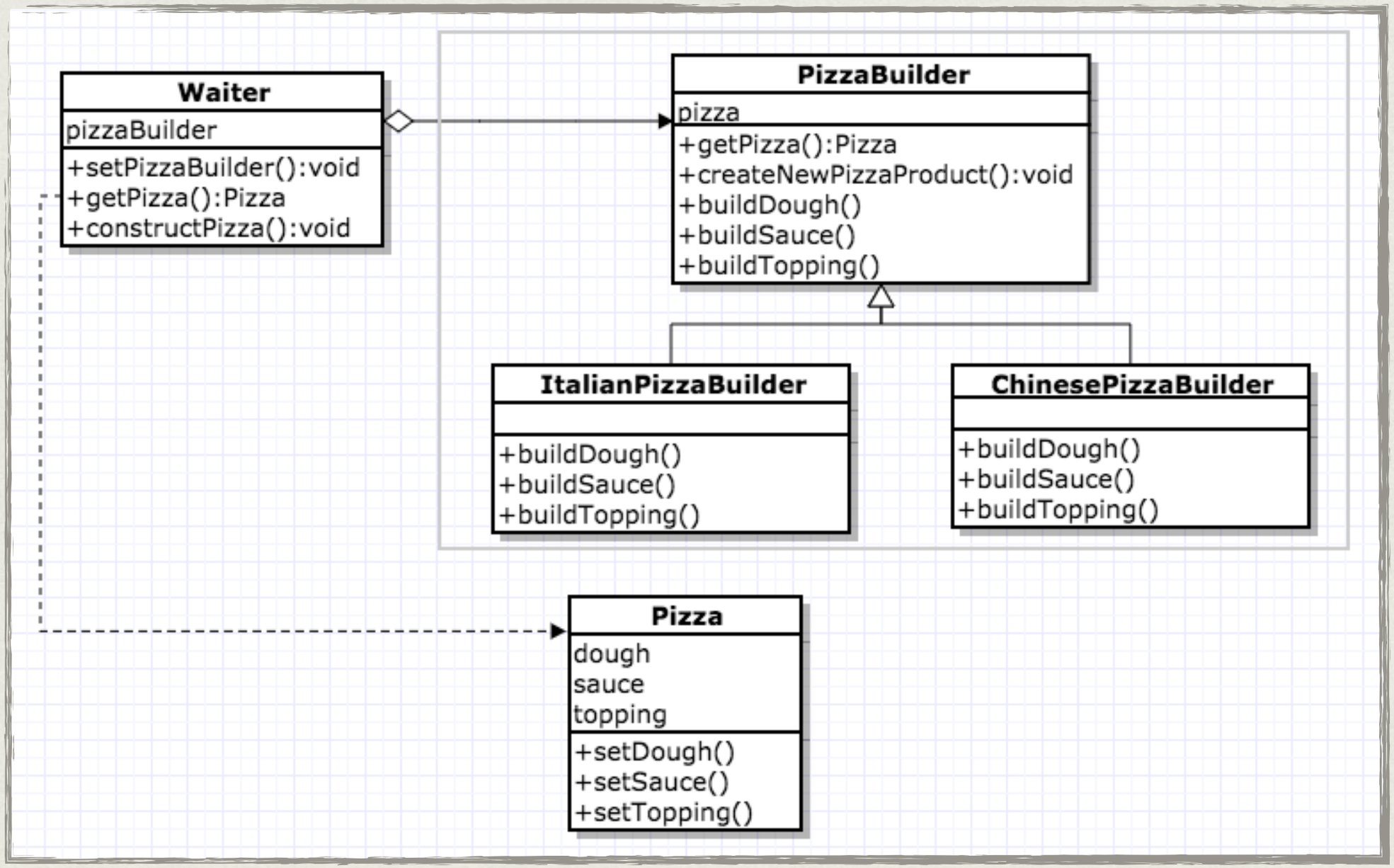
根据参数确定用哪个工厂实现，参数多了不好弄

ABSTRACT FACTORY

- 抽象工厂常通过FactoryMethod、Singleton实现
- 每个具体工厂常用单例实现(满足依赖反转原则)
- 可代替Facade对客户端隐藏平台相关特性
- 作用于对象上，通过使用FactoryMethod更灵活定制子类或扩展子类(满足开闭原则)
- 常用于平台无关的场景(多个系统平台、多数据源…)

BUILDER

- 将复杂对象的构造和表示分离，逐步构建复杂对象
- 解释一个复杂对象的表示



BUILDER 示例代码

复杂对象(就假设有很多实例化时需要的参数)

```
class Pizza{
    private String dough = "";
    private String sauce = "";
    private String topping = "";

    public void setDough(String dough){
        this.dough = dough;
    }
    public void setSauce(String sauce){
        this.sauce = sauce;
    }
    public void setTopping(String topping){
        this.topping = topping;
    }

    @Override
    public String toString(){
        return "pizza of dough:" + this.dough + " sauce:" + this.sauce + " topping:" +
this.topping;
    }
}
```

BUILDER 示例代码

Abstract Builder

```
abstract class PizzaBuilder {
    protected Pizza pizza;
   public Pizza getPizza(){
        return pizza;
    }
    public void createNewPizzaProduct(){
        pizza = new Pizza();
    }
    public abstract void buildDough();
    public abstract void buildSauce();
    public abstract void buildTopping();
}
```

Concrete Builder

复制代码？不科学。。。

```
class ItalianPizzaBuilder extends PizzaBuilder{
    public void buildDough(){
        pizza.setDough("cross");
    }
   public void buildSauce(){
        pizza.setSauce("mild");
    }
    public void buildTopping(){
        pizza.setTopping("ham+pineapple");
    }
}
```

```
class ChinesePizzaBuilder extends PizzaBuilder{
    public void buildDough(){
        pizza.setDough("pan baked");
    }
    public void buildSauce(){
        pizza.setSauce("hot");
    }
    public void buildTopping(){
        pizza.setTopping("pepperoni+salami");
    }
}
```

BUILDER 示例代码

```
class Waiter {
    private PizzaBuilder pizzaBuilder;
   public void setPizzaBuilder(PizzaBuilder pb){
        this.pizzaBuilder = pb;
    }
   public Pizza getPizza(){
        return pizzaBuilder.getPizza();
    }
   public void constructPizza(){
        pizzaBuilder.createNewPizzaProduct();
        pizzaBuilder.buildDough();
        pizzaBuilder.buildSauce();
        pizzaBuilder.buildTopping();
    }
}
```

逐步构建复杂对象

给derector传入构建工厂，
具体细节自己搞定吧

```
public static void main(String[] args) {
    Waiter waiter = new Waiter();
    PizzaBuilder italianPizzaBuilder = new ItalianPizzaBuilder();
    PizzaBuilder chinesePizzaBuilder = new ChinesePizzaBuilder();

    waiter.setPizzaBuilder(italianPizzaBuilder);
    waiter.constructPizza();
    Pizza pizza1 = waiter.getPizza();
    System.out.println(pizza1.toString());
    waiter.setPizzaBuilder(chinesePizzaBuilder);
    waiter.constructPizza();
    Pizza pizza2 = waiter.getPizza();
    System.out.println(pizza2.toString());
}
```

构造完成才返回具体对象

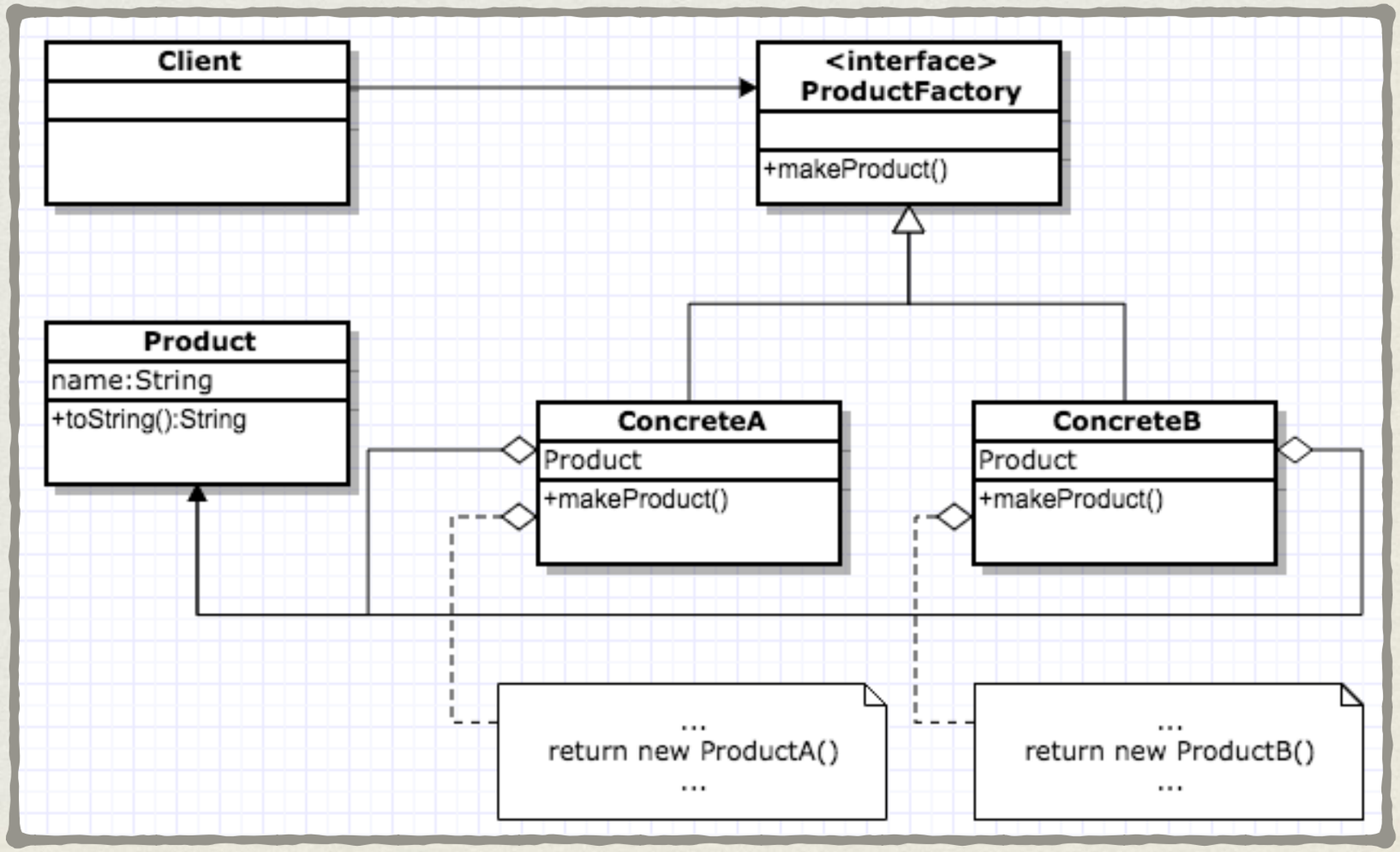
BUILDER：SUMMARY

- 提供抽象接口，便于运行时更改对象的状态(通过提供新的builder)
- 通过剥离复杂对象的构建和表示，提升代码的模块性(遵循开闭原则)
- 简化多参数的复杂对象实例化，并且隐藏内部实例的状态、内部结构和组装方式
- 通常用于构建Composite
- 可通过单例实现

FACTORY METHOD

- 抽象接口提供一个占位符，延迟实例化子类
- 具体抽象接口提供默认实现
- 扁平化类结构(遵循里氏替换/继承原则)
- 通常在模版方法中调用

代码请忽略一百行。。。



PROTOTYPE

- 通过原型实例复制创建新对象
- 常与AbstractFactory、Composite、Adapter配合使用达到良好效果

适用场景：

- 运行时实例化动态加载类
- 避免建立与父类相同的层次结构
- 实例化有状态的对象
- 参数变化较少时，初始化比实例化更高效

UML图参考工厂方法

跟工厂方法大同小异。。。

PROTOTYPE 示例代码

```
interface ClientSession{
    ClientSession clone();
    String getSessionId();
    void execute();
}
```

提供clone方法

```
class SessionModule{
    private static List<ClientSession> clientSessions = new ArrayList<>();

    public static void addSession(ClientSession cs){
        clientSessions.add(cs);
    }
    public static ClientSession createSession(String sessionId){
        for (ClientSession cs : clientSessions) {
            if(cs.getSessionId().equals(sessionId)){
                return cs.clone();
            }
        }
        System.out.println("session id: "+sessionId +" is not exists");
        return null;
    }
}
```

PROTOTYPE 示例代码

```
class AppClient implements ClientSession{
    private String sessionId = "App";
   @Override
    public ClientSession clone(){ return new AppClient(); }
    @Override
    public String getSessionId(){ return sessionId; }
    @Override
    public void execute(){
        System.out.println("session:"+sessionId +" does something of app");
    }
}
```

每个具体抽象

负责实例化

```
class WebClient implements ClientSession{
    private String sessionId = "Web";
   @Override
    public ClientSession clone(){ return new WebClient(); }
    @Override
    public String getSessionId(){ return sessionId; }
    @Override
    public void execute(){
        System.out.println("session:"+sessionId +" does something of web");
    }
}
```

PROTOTYPE 示例代码

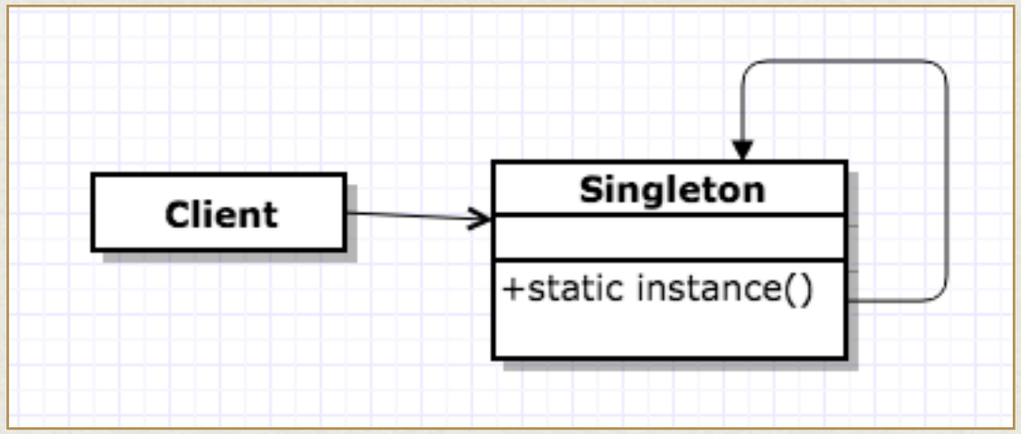
```
public static void main(String[] args){
    String[] a = {"App","Native","Web"};
    SessionModule.addSession(new AppClient());
    SessionModule.addSession(new WebClient());
    List<ClientSession> css = new ArrayList<>();
    for(String c:a){
        ClientSession cs = SessionModule.createSession(c);
        if(cs != null){
            css.add(cs);
        }
    }

    for(ClientSession cs: css){
        cs.execute();
    }
}
```

```
session id: Native is not exists
session:App does something of app
session:Web does something of web
```

延迟返回的

SINGLETON

- 类只有一个实例
- 多用于全局配置、统一访问入口的对象
- 许多模式用单例实现，如AbstractFactory、Builder、Prototype、Facade、State
- 但是大多数时候用不着



单例实现：

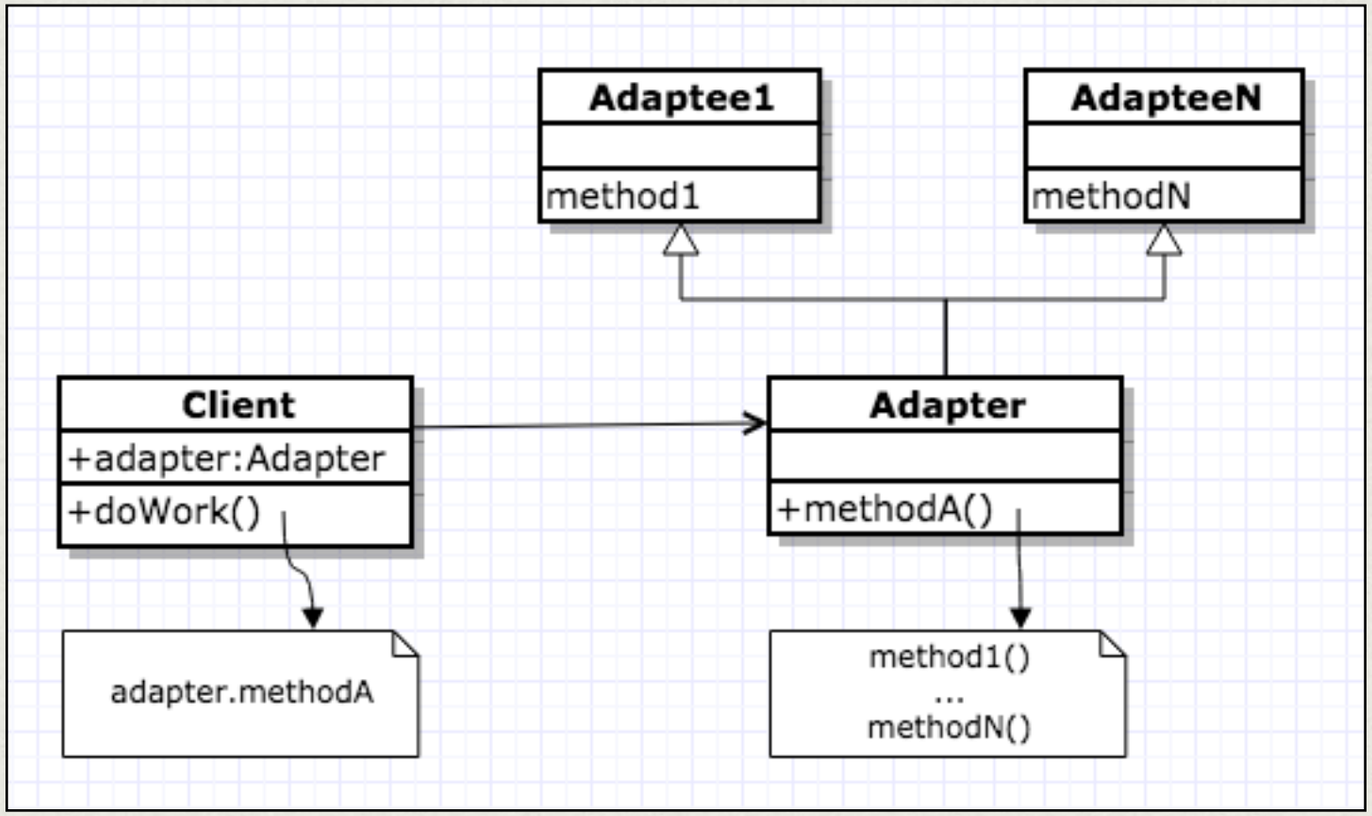
- 构造器是private或protected
- 定义static字段和访问该对象的static 方法
- 可延迟初始化

```
public class Singleton {
    private static volatile Singleton instance;

    private Singleton() {}

    public static Singleton getInstance(String value) {
        if (instance == null) {
            synchronized (Singleton.class) {
                if (instance == null) {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```

结构化设计模式关注如何组织对象

ADAPTER

一个转换不同接口的包装类



ADAPTER: 示例代码

```
class Line{
    public void draw(int x1, int y1, int x2, int y2){
        System.out.println("draw line from ("+x1+","+y1+") to ("+x2+","+y2+") ");
    }}
class Rectangle{
    public void draw(int x1, int y1, int width, int height){
        System.out.println("draw rectangle at point ("+x1+","+y1+") with width:"+width+" height:"+height
+" ");
    }}
```

转换方法

```
interface Shape{
   void draw(int x1,int y1,int x2,int y2);
   }
class LineAdapter implements Shape {
    private Line adaptee;
    public LineAdapter(Line line) {
        this.adaptee = line;
    }
    @Override
    public void draw(int x1, int y1, int x2, int y2) {
        adaptee.draw(x1, y1, x2, y2);
    }
}
```

```
class RectangleAdapter implements Shape {
    private Rectangle adaptee;
    public RectangleAdapter(Rectangle rectangle) {
        this.adaptee = rectangle;
    }
    @Override
    public void draw(int x1, int y1, int x2, int y2) {
        int x = Math.min(x1, x2);
        int y = Math.min(y1, y2);
        int width = Math.abs(x2 - x1);
        int height = Math.abs(y2 - y1);
        adaptee.draw(x, y, width, height);
    }
```

ADAPTER: 示例代码

包装不同对象进来

```
public static void main(String[] args) {
    Shape[] shapes = {new RectangleAdapter(new Rectangle()), new LineAdapter(new Line())};
    int x1 = 10, y1 = 20;
    int x2 = 30, y2 = 60;
    for(Shape shape:shapes){
        shape.draw(x1,y1,x2,y2);
    }
}
```

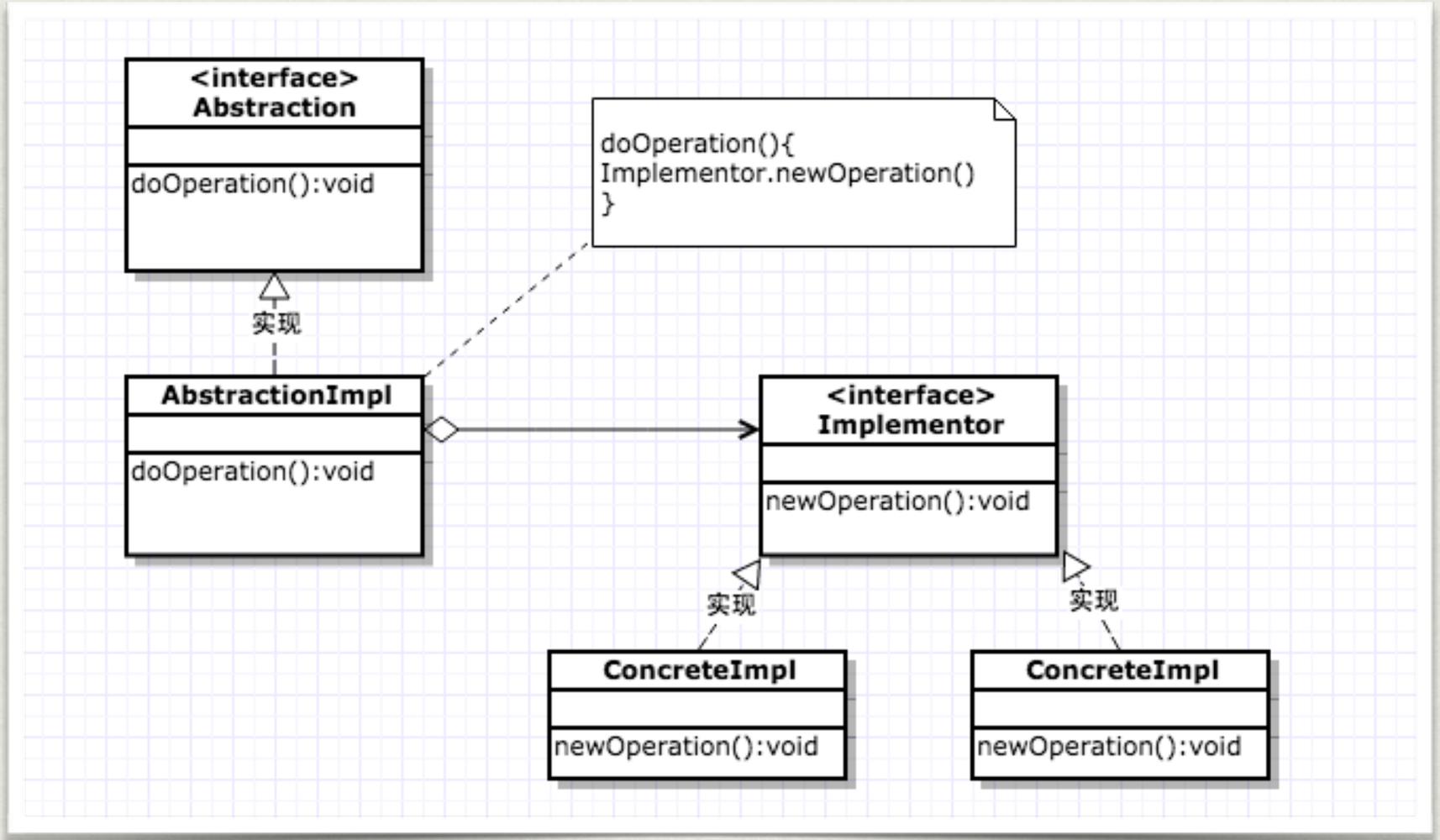
综述：

- Adapter跟Bridge、Facade、Proxy很像，都是wrapper，实现依赖反转功能
- Adapter常常是一对一的包装，提供不同的接口，主要用于兼容旧接口
- Bridge比Adapter更抽象
- Facade一对多的包装，常常使用新接口实现
- Proxy主要提供相同接口的代理
- Decorator也是一个wrapper，主要增强功能，可递归组合，不改变对象本身；Adapter改变对象本身

都是套路。。。

BRIDGE

比Adapter更抽象一层的wrapper，主要为了解耦，实现抽象之间的连接



BRIDGE:示例

```
interface Persistence{
    public String persist(Object object);
    public Object findById(String objectId);
    public void deleteById(String id);
}
class PersistenceImp implements Persistence {
    private PersistenceImplementor implementor = null;
    public PersistenceImp(PersistenceImplementor imp) {
        this.implementor = imp;
    }
    @Override
    public void deleteById(String id) {
        implementor.deleteObject(Long.parseLong(id));
    }
    @Override
    public Object findById(String objectId) {
        return implementor.getObject(Long.parseLong(objectId));
    }
    @Override
    public String persist(Object object) {
        return Long.toString(implementor.saveObject(object));
    }
}
```

这里可以运行时自由切换实现接口

BRIDGE:示例

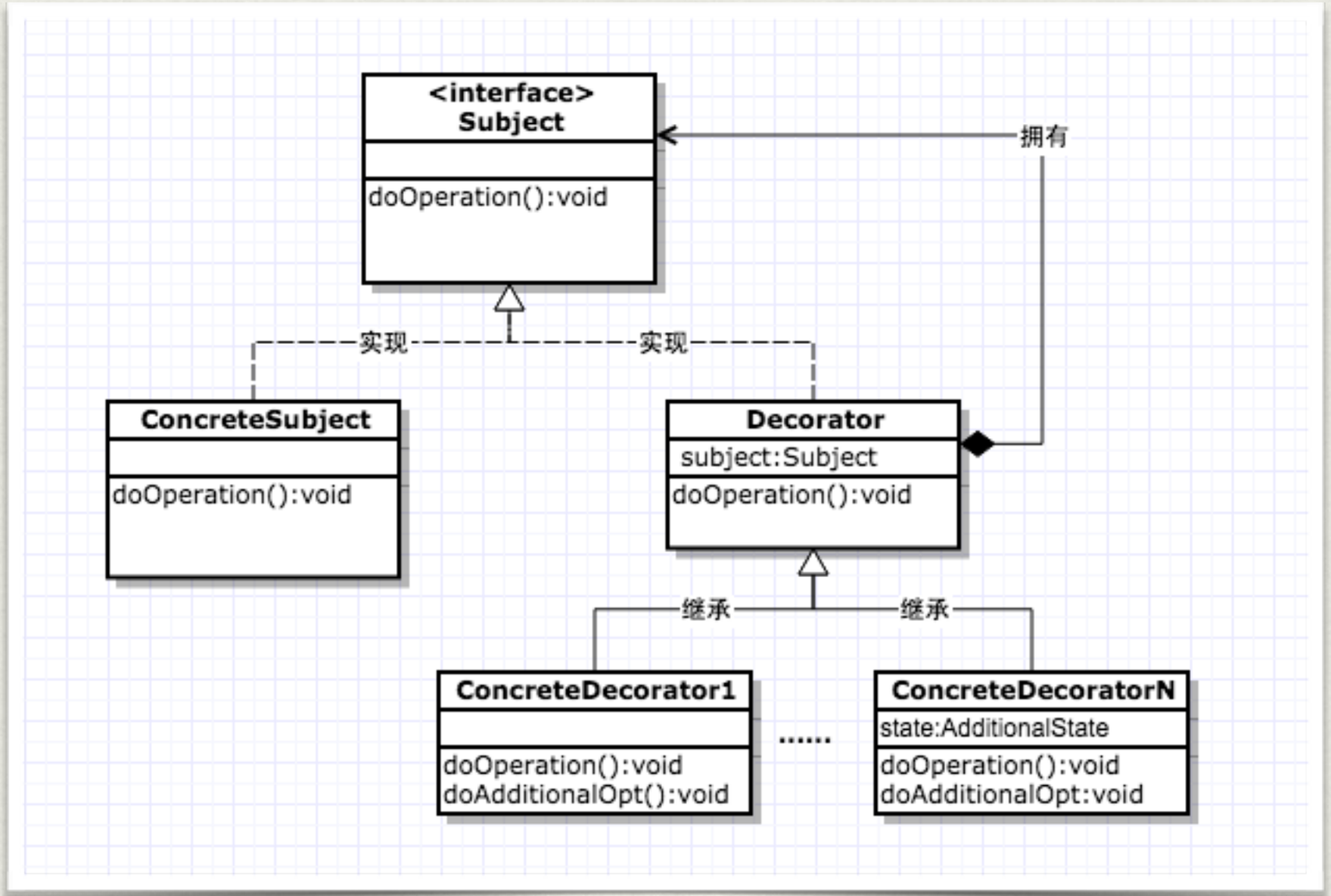
```
public static void main(String[] args){
    PersistenceImplementor implementor = null;
    if(databaseDriverExists()){
        implementor = new DabatasePersistenceImplementor();
    } else {
        implementor = new FilePersistenceImplementor();
    }
    Persistence persistenceAPI = new PersistenceImp(implementor);
    Object o = persistenceAPI.findById("12343755");
    //更改对象，存储
    persistenceAPI.persist(o);

    persistenceAPI = new PersistenceImp(new DabatasePersistenceImplementor());
    persistenceAPI.deleteById("2323");
}
private static boolean databaseDriverExists() {
    return false;
}
```

具体实现类

抽象类包装了各种具体实现，这样实现接口可自由扩展不影响上层调用

DECORATOR

还是一个wrapper，层层包装主体对象，不需要扩展子类而增加额外功能，链条的最末端最先执行。每个wrapper执行一项功能，满足单一职责原则、继承原则



DECORATOR:示例

```
interface Widget {
    void draw();
}
class TextField implements Widget {
    private int width, height;

    public TextField(int width, int height) {
        this.width = width;
        this.height = height;
    }
    public void draw() {
        System.out.println("TextField: " + width + ", " +
height);
    }
}
abstract class Decorator implements Widget {
    private Widget widget;

    public Decorator(Widget widget) {
        this.widget = widget;
    }

    public void draw() {
        widget.draw();
    }
}
```

并未改变主题对象，只是附加功能

拥有主题对象的引用

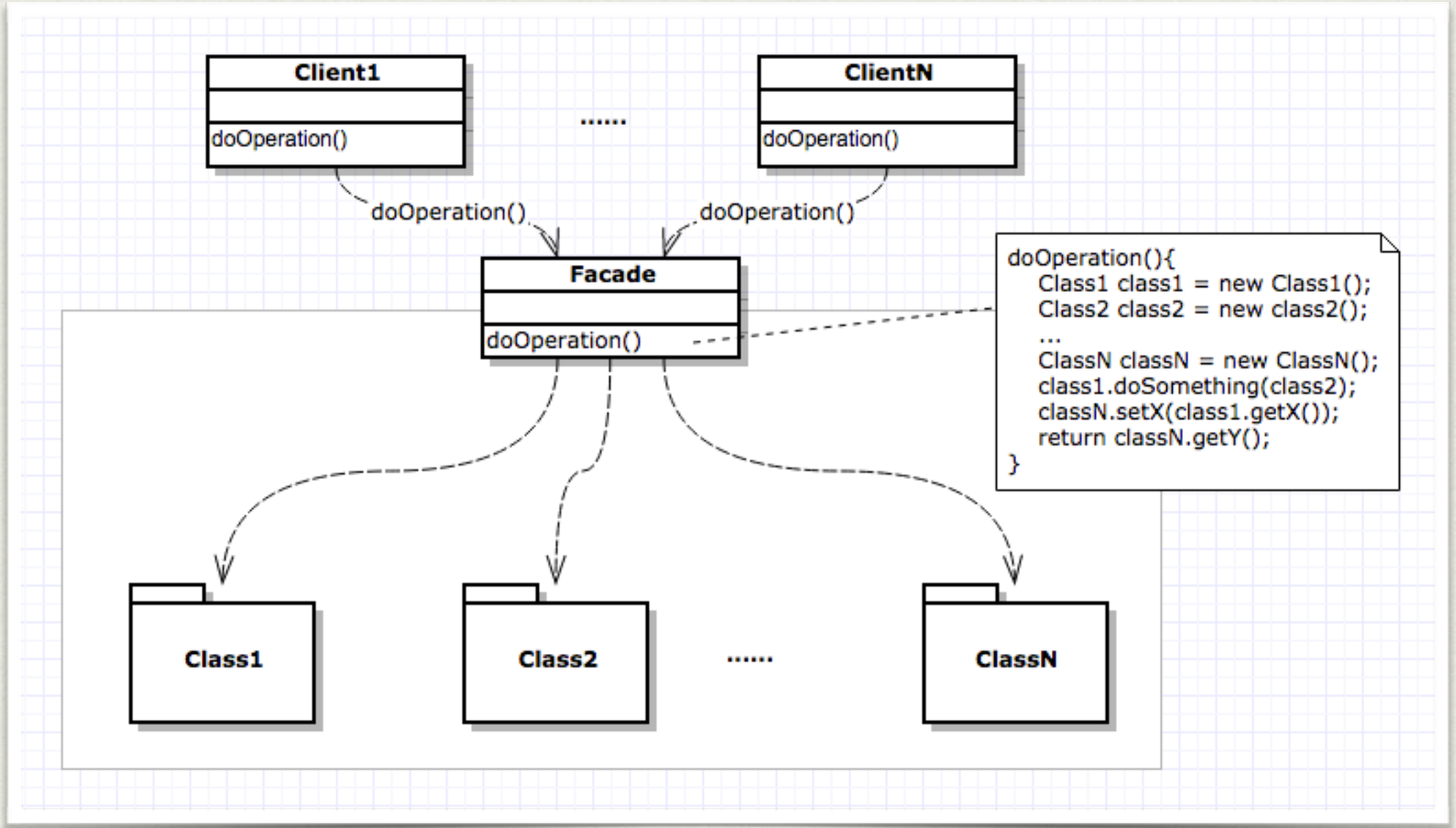
代理执行

层层包装，最末端 new TextFiled先执行

```
class BorderDecorator extends Decorator {
    public BorderDecorator(Widget widget) {
        super(widget);
    }
    public void draw() {
        super.draw();
        System.out.println("Add border");
    }
}
class ScrollDecorator extends Decorator {
    public ScrollDecorator(Widget widget) {
        super(widget);
    }
    public void draw() {
        super.draw();
        System.out.println("Add scrollbar");
    }
}
public class DecoratorDemo {
    public static void main(String[] args) {
        Widget widget = new BorderDecorator(
            new BorderDecorator(
                new ScrollDecorator(
                    new TextField(80, 24))));
        widget.draw();
    }
}
```

FACADE

还是wrapper，为系统中的一组接口提供统一的接口，减少客户端与子系统的依赖



WIKIPEDIA的FACADE经典案例

```
class CPU {
    CPU(){ ("initialize CPU”); }
    public void freeze() { System.out.println(“freeze"); }
    public void jump(long position) { System.out.println(“jump:"+String.valueOf(position)); }
    public void execute() { System.out.println(“execute"); }
}
class HardDrive {
    HardDrive(){
        System.out.println("initialize HardDrive");
    }
    public byte[] read(long lba, int size) {
        byte[] b = new byte[size];
        System.out.println("HardDrive read sector:"+String.valueOf(lba)+" with size:"+String.valueOf(size));
        return b;
    }
}
class Memory {
    Memory(){
        System.out.println("initialize Memory");
    }
    public void load(long position, byte[] data) {
        System.out.println("load from position:"+String.valueOf(position) + " "+data.length);
    }
}
```

复杂对象或子系统

WIKIPEDIA的FACADE经典案例

```
class ComputerFacade {
    private long BOOT_ADDRESS = 0,BOOT_SECTOR = 20;
    private int SECTOR_SIZE = 256;
    private CPU processor;
    private Memory ram;
    private HardDrive hd;
    public ComputerFacade() {
        this.processor = new CPU();
        this.ram = new Memory();
        this.hd = new HardDrive();
    }
    public void start() {
        processor.freeze();
        ram.load(BOOT_ADDRESS, hd.read(BOOT_SECTOR, SECTOR_SIZE));
        processor.jump(BOOT_ADDRESS);
        processor.execute();
    }
}
public class FacadeDemo {
    public static void main(String[] args) {
        ComputerFacade computer = new ComputerFacade();
        computer.start();
    }
}
```

对客户端简化了操作

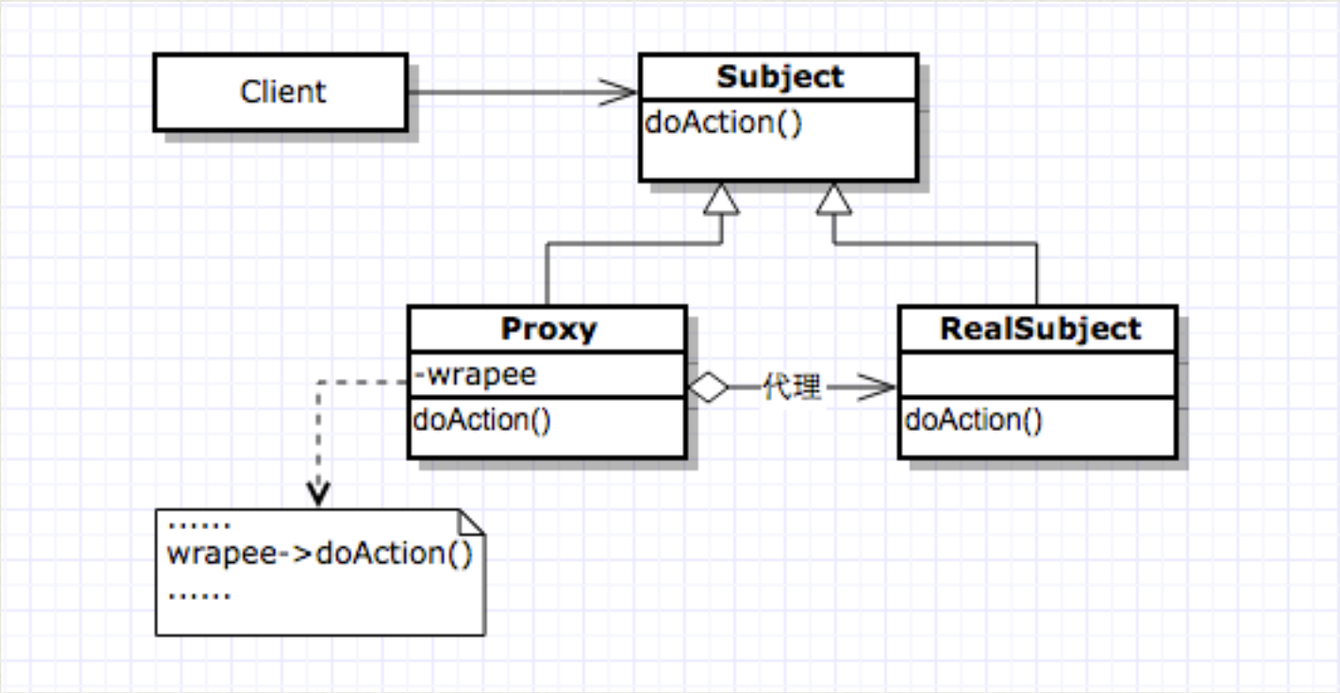
PROXY

为对象提供一个占位符的wrapper，专注于对象如何访问的疑难杂症

代理远程对象?

对象引用?

便于对象访问前加锁?



PROXY:示例

```java
interface SocketInterface {
    String readLine();
    void  writeLine(String str);
    void  dispose();
}
class SocketProxy implements SocketInterface {
    private Socket socket;
    private BufferedReader in;
    private PrintWriter out;
    public SocketProxy(String host, int port, boolean wait) {
        try {
            if (wait) {
                ServerSocket server = new ServerSocket(port);
                socket = server.accept();
            } else {
                socket = new Socket(host, port);
            }
            in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
            out = new PrintWriter(socket.getOutputStream(), autoFlush: true);
        } catch(IOException e) {
            e.printStackTrace();
        }
    }
    public String readLine() {
        String str = null;
        try {
            str = in.readLine();
        } catch( IOException e ) {
            e.printStackTrace();
        }
        return str;
    }
    public void writeLine(String str) {
        out.println(str);
    }
    public void dispose() {
        try {
            socket.close();
        } catch(IOException e) {
            e.printStackTrace();
        }
```

占位符

封装了比较复杂的对象在这里

代理执行

PROXY:示例

```
public static void main( String[] args ) {
    SocketInterface socket = new SocketProxy( host: "127.0.0.1", port: 8089, args[0].equals("first") ? true : false
    String  str;
    boolean skip = true;
    while (true) {
        if (args[0].equals("second") && skip) {
            skip = !skip;
        } else {
            str = socket.readLine();
            System.out.println("Receive - " + str);
            if (str.equals(null)) {
                break;
            }
        }
        System.out.print( "Send ---- " );
        str = new Scanner(System.in).nextLine();
        socket.writeLine( str );
        if (str.equals("quit")) {
            break;
        }
    }
    socket.dispose();
}
```

对客户端屏蔽复杂逻辑

按需创建对象

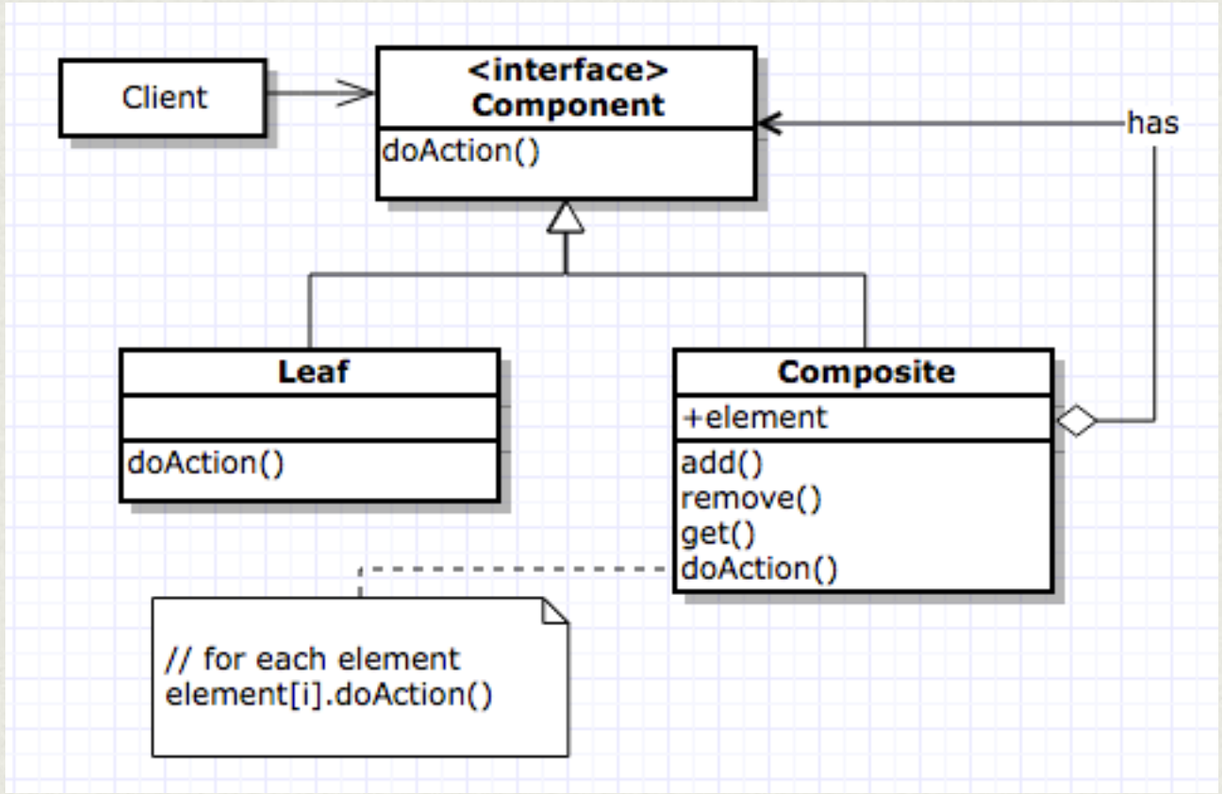
更安全

更高效

COMPOSITE

对象的结构和组成；把相同操作的对象抽象出来归类组合

- 常与其它模式配合使用
- 通过Iterator遍历节点；
- 可通过Visitor增加操作；
- 与Flyweight共享节点；
- 责任链模式通过根节点访问对象；
- 通过Decorator增强对象属性；
- 可使用Observer在对象间传递消息；
- 通过State管理对象的内部状态；
- Mediator



- 常与其它模式配合使用常用于MVC中的View、Web前端框架的组件化(react.js)、UI的状态管理(redux)
- 可用各种算法实现

COMPOSITE

抽象file和directory相同类型操作为接口，file和directory实现此接口

```java
//component
interface AbstractFile {
    void ls();
}
//leaf
class File implements AbstractFile {
    private String name;

    public File(String name) {
        this.name = name;
    }
    public void ls() {
        System.out.println(CompositeDemo.compositeBuilder + name
    }
}
//composite
class Directory implements AbstractFile {
    private String name;
    private ArrayList includedFiles = new ArrayList();

    public Directory(String name) {
        this.name = name;
    }
    public void add(Object obj) {
        includedFiles.add(obj);
    }
    public void ls() {
        System.out.println(CompositeDemo.compositeBuilder + name);
        CompositeDemo.compositeBuilder.append("   ");
        for (Object includedFile : includedFiles) {
            AbstractFile obj = (AbstractFile) includedFile;
            obj.ls();
        }
        CompositeDemo.compositeBuilder.setLength(
                CompositeDemo.compositeBuilder.length() - 3);
    }
```

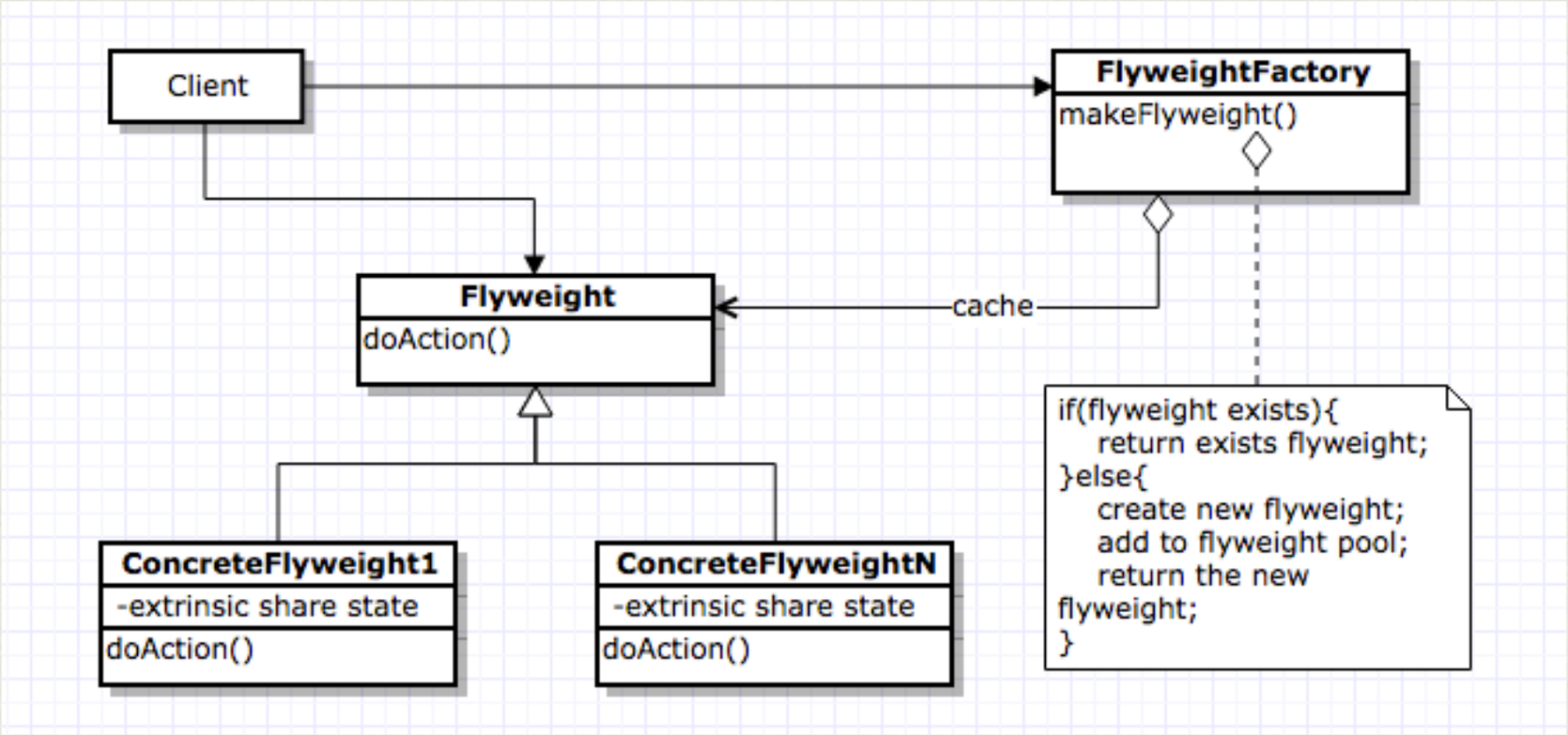
```java
public static StringBuffer compositeBuilder = new StringBuffer();

public static void main(String[] args) {
    Directory music = new Directory( name: "music");
    Directory scorpions = new Directory( name: "Scorpions");
    Directory jon = new Directory( name: "Jon");
    File track1 = new File( name: "Forrest Gump Suite.mp3");
    File track2 = new File( name: "Last Goodbye.mp3");
    File track3 = new File( name: "Love Story meets Viva La Vida.mp3"
    File track4 = new File( name: "Long Long Ago.mp3");
    File track5 = new File( name: "Peanuts Medley Live.mp3");
    File track6 = new File( name: "Wind of change.mp3");
    music.add(track1);
    music.add(jon);
    music.add(track2);
    jon.add(track3);
    jon.add(track4);
    jon.add(track5);
    music.add(scorpions);
    scorpions.add(track6);
    music.ls();
```

ls 对于file和directory的操作都是一致的

```
objc[4856]: Class JavaLaunchHelper is implemented in both /Library/
music
   Forrest Gump Suite.mp3
   Jon
      Love Story meets Viva La Vida.mp3
      Long Long Ago.mp3
      Peanuts Medley Live.mp3
   Last Goodbye.mp3
   Scorpions
      Wind of change.mp3

Process finished with exit code 0
```

FLYWEIGHT

缓存并管理一组小对象



FLYWEIGHT:示例

```java
//一个细力度的小对象
interface Page{
    String getPage();
    void setPage(String pageSize);
}
class MyPage implements Page{
    String pageSize;
    MyPage(String pageSize){
        this.pageSize = pageSize;
    }
    public String getPage(){
        return this.pageSize;
    }
    public void setPage(String pageSize){
        this.pageSize = pageSize;
    }
}
```

```java
public class FlyweightDemo {
    public static void main(String[] args){
        String[] pages =
{"256","64","128","128","256","312","100"};
        for(String page:pages){
            PageFactory.getPage(page);
        }
        PageFactory.report();
    }
}
```

缓存对象到外部
存储treemap

```java
//制作工厂
class PageFactory{
    private static Map treeMap = new TreeMap();
    private static Page page;
    private static int sharedPage = 0;
    public static Page getPage(String num){
        if (treeMap.containsKey(num)) {
            sharedPage++;
        } else {
            page = new MyPage(num);
            treeMap.put(num,page.hashCode());
        }
        return page;
    }
    public static void report() {
        System.out.print("new Page - " + treeMap.size()
                + ", \"shared\" Page - " + sharedPage + ", ");
        for (Object o : treeMap.keySet()) {
            System.out.print(o + " ");
        }
        System.out.println();
    }
}
```

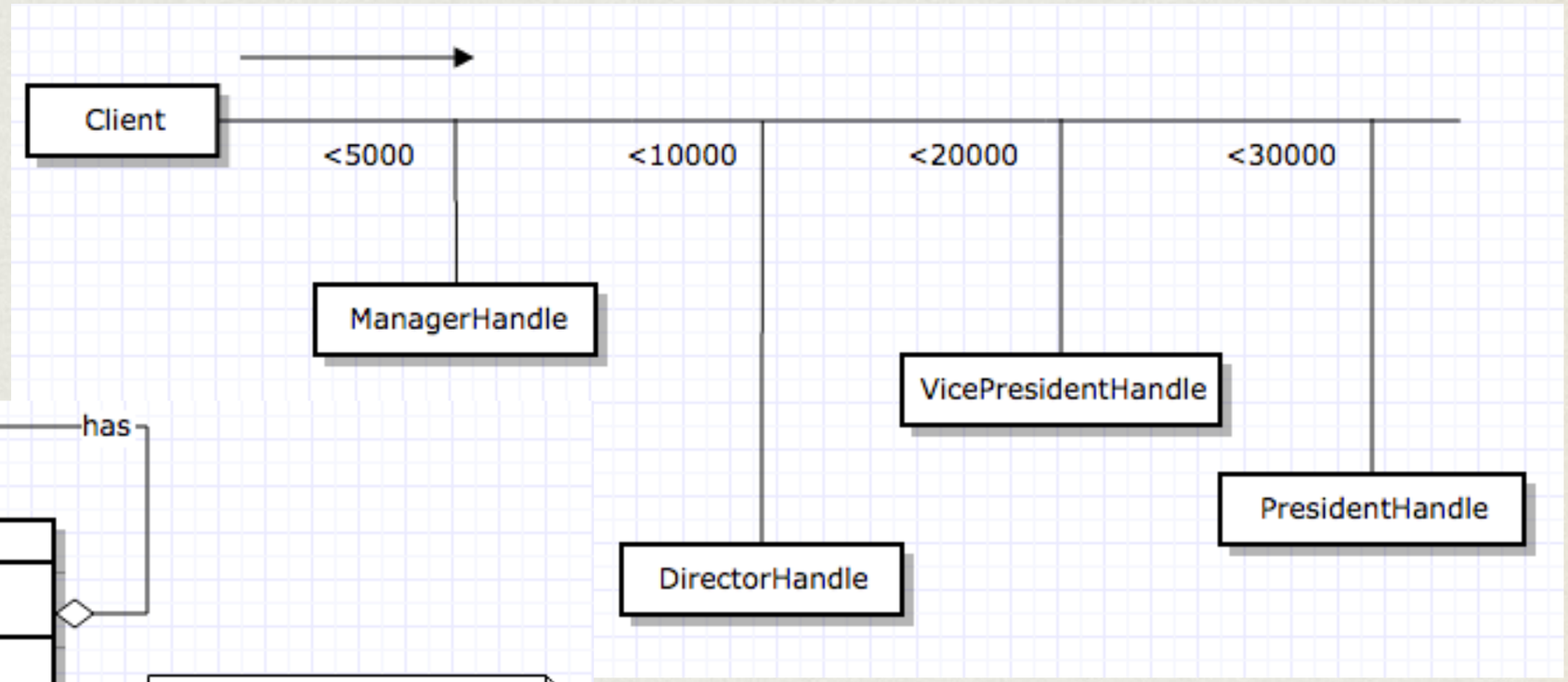
```
new Page - 5, "shared" Page - 2, 100 128 256 312 64
———
```

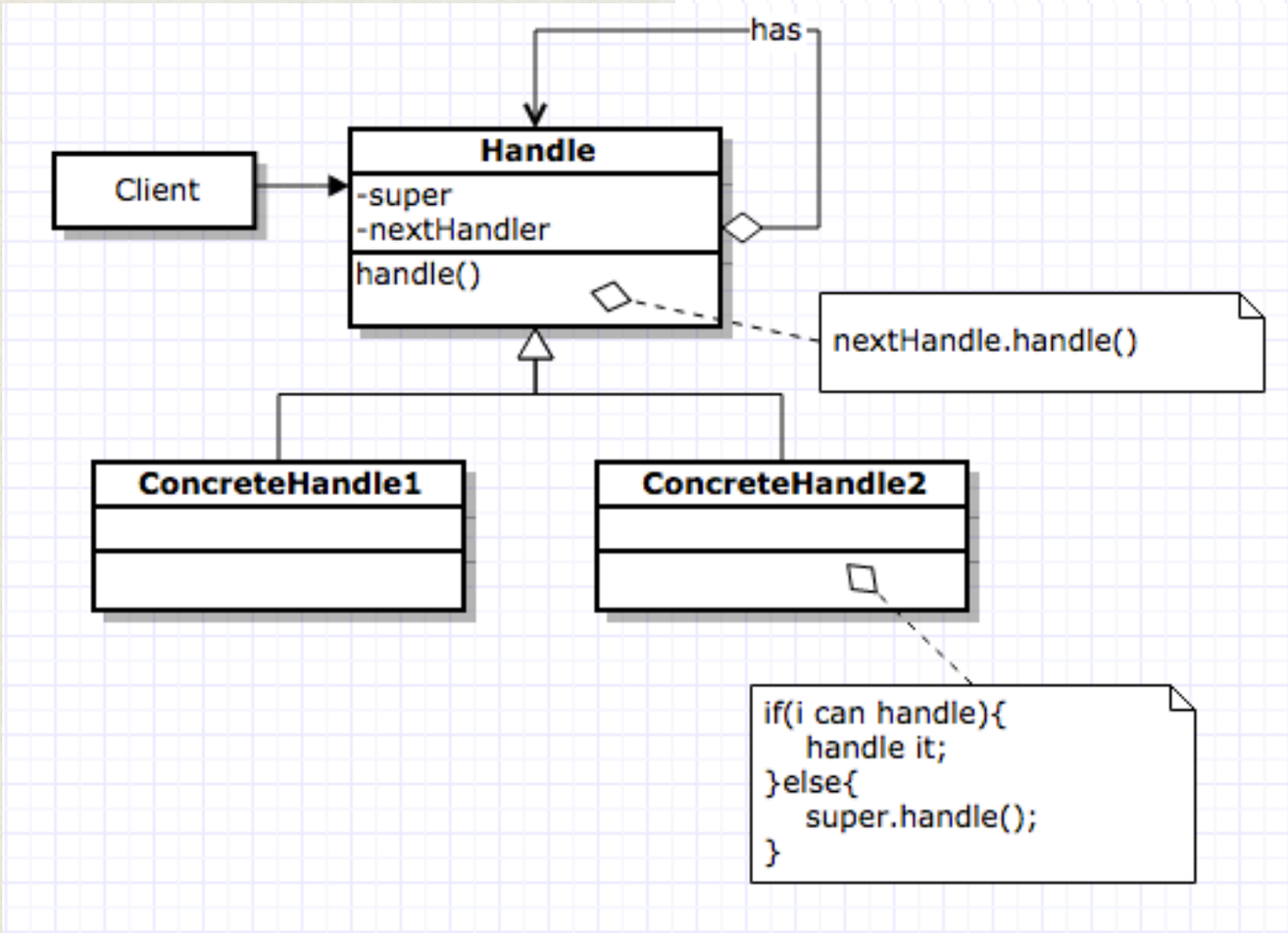
行为类设计模式抽象对象如何通信(分离发送者和接收者)

CHAIN OF RESPONSIBILITY

责任链模式：往对象递归发送请求，直到有人能处理，不知道有哪个节点、有多少个节点能处理

与装饰者模式不同，
包装的是一个请求





链条上节点只做一件
事，遵循单一职责

CHAIN OF RESPONSIBILITY

```
abstract class PurchaseHandle{
    protected static final double BASE = 500;
    protected PurchaseHandle successor;

    abstract protected double getAllowable();
    abstract protected String getRole();

    public void setSuccessor(PurchaseHandle successor) {
        this.successor = successor;
    }

    public void processRequest(PurchaseRequest request) {
        if (request.getAmount() < this.getAllowable()) {
            System.out.println(this.getRole() + " will approve $" + request.getAmount());
        } else if (successor != null) {
            successor.processRequest(request);
        }
    }
}
```

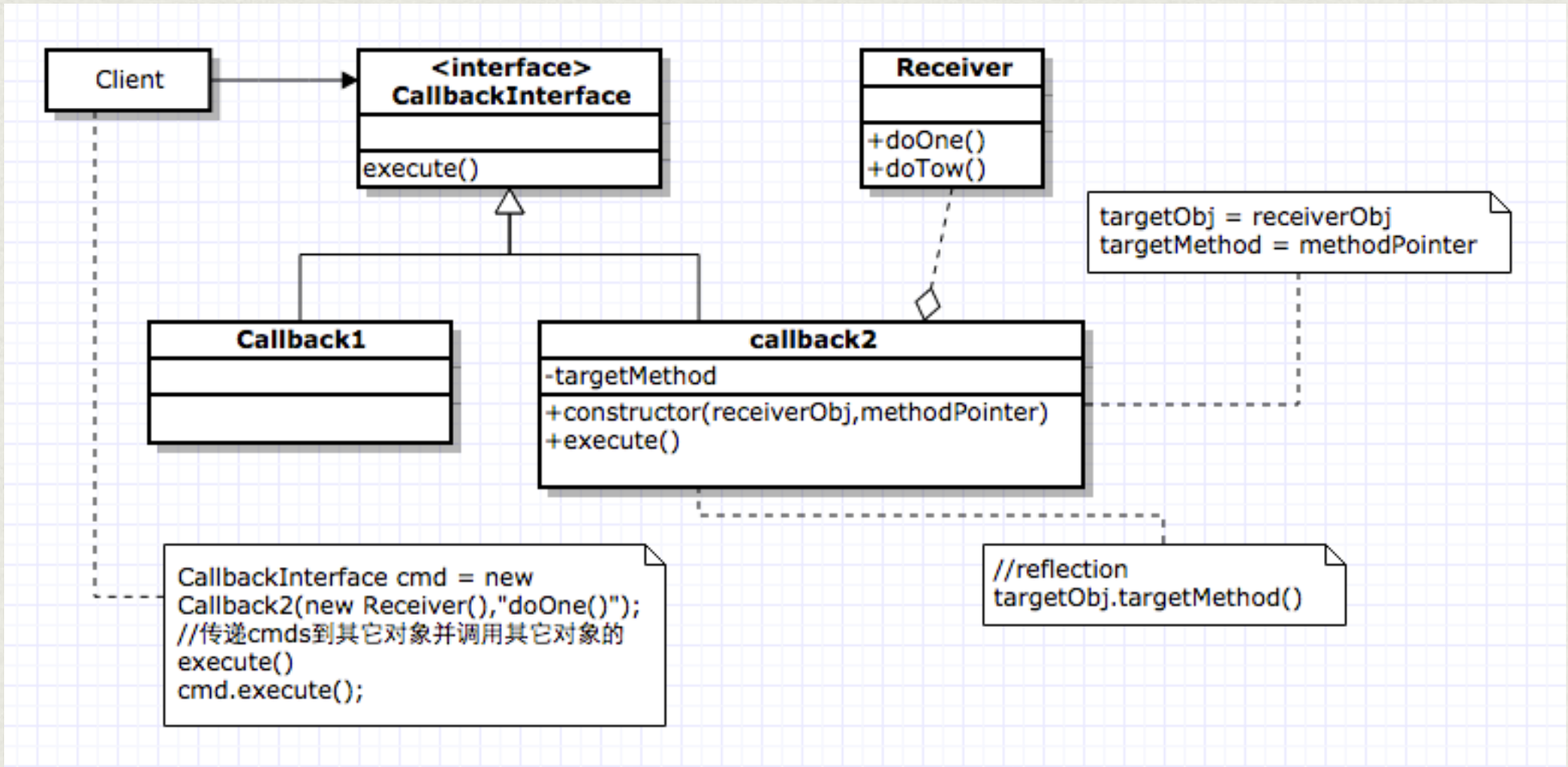
满足条件的执行

不满足条件的执行往当前节点发

不会掉链子？？？

COMMAND

命令模式：封装一组为对象，也称为事务；Composite的一个实例



COMMAND

```java
//receiver
interface FileSystemReceiver {
    void openFile();
    void writeFile();
    void closeFile();
}
class FileReceiver implements FileSystemReceiver{
   // ... ...
}
//invoker
class Macro {
    private final List actions;
    public Macro() {
        actions = new ArrayList<>();
    }
    public void record(Action action) {
        actions.add(action);
    }
   public void run() {
        actions.forEach(Action::perform);
    }
}
```

```java
interface Action {
    public void perform();
}
//concrete command
class OpenFile implements Action {
    private final FileReceiver fileReceiver;
    public OpenFile(FileReceiver fileReceiver) {
        this.fileReceiver = fileReceiver;
    }
    public void perform() {
        fileReceiver.openFile();
    }
}
//... ...
    public static void main(String[] args){
        FileReceiver fileReceiver = new FileReceiver();
        Macro macro = new Macro();
        macro.record(fileReceiver::openFile);
        macro.record(fileReceiver::writeFile);
        macro.record(fileReceiver::closeFile);
        macro.run();
    }
//... ...
```

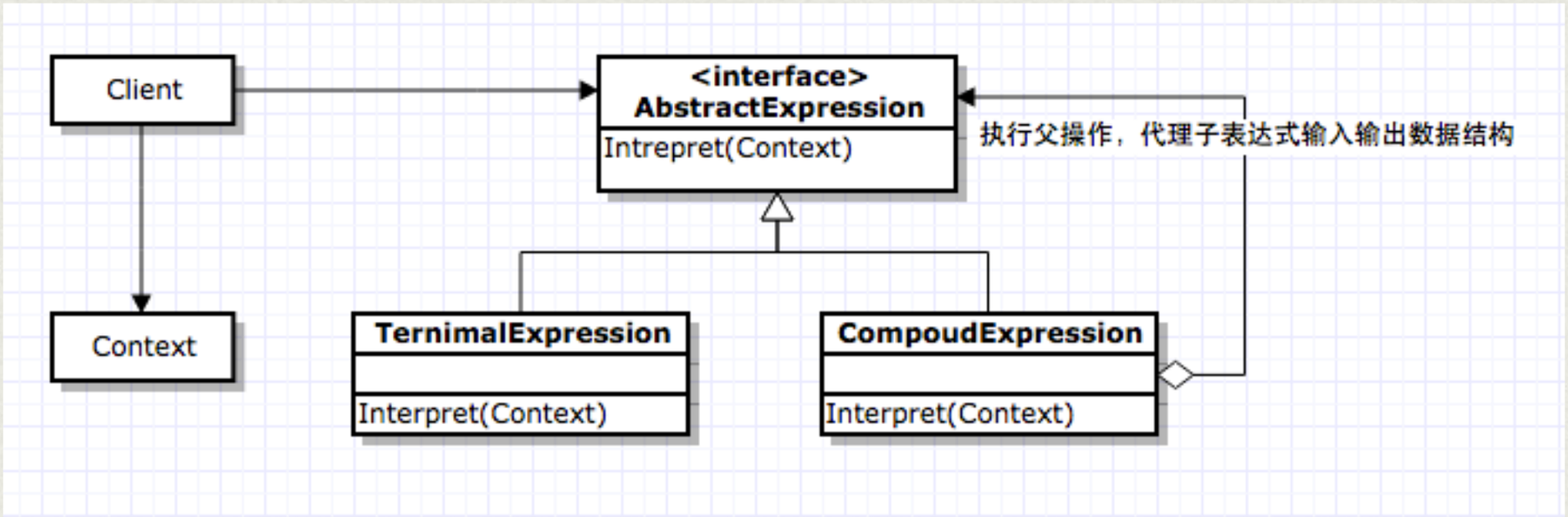
代理执行

run command

java的callback回调怎么弄哟?

INTERPRETER

解析器模式：定义一套语法引擎，应用于面向对象编程语法和语言解释



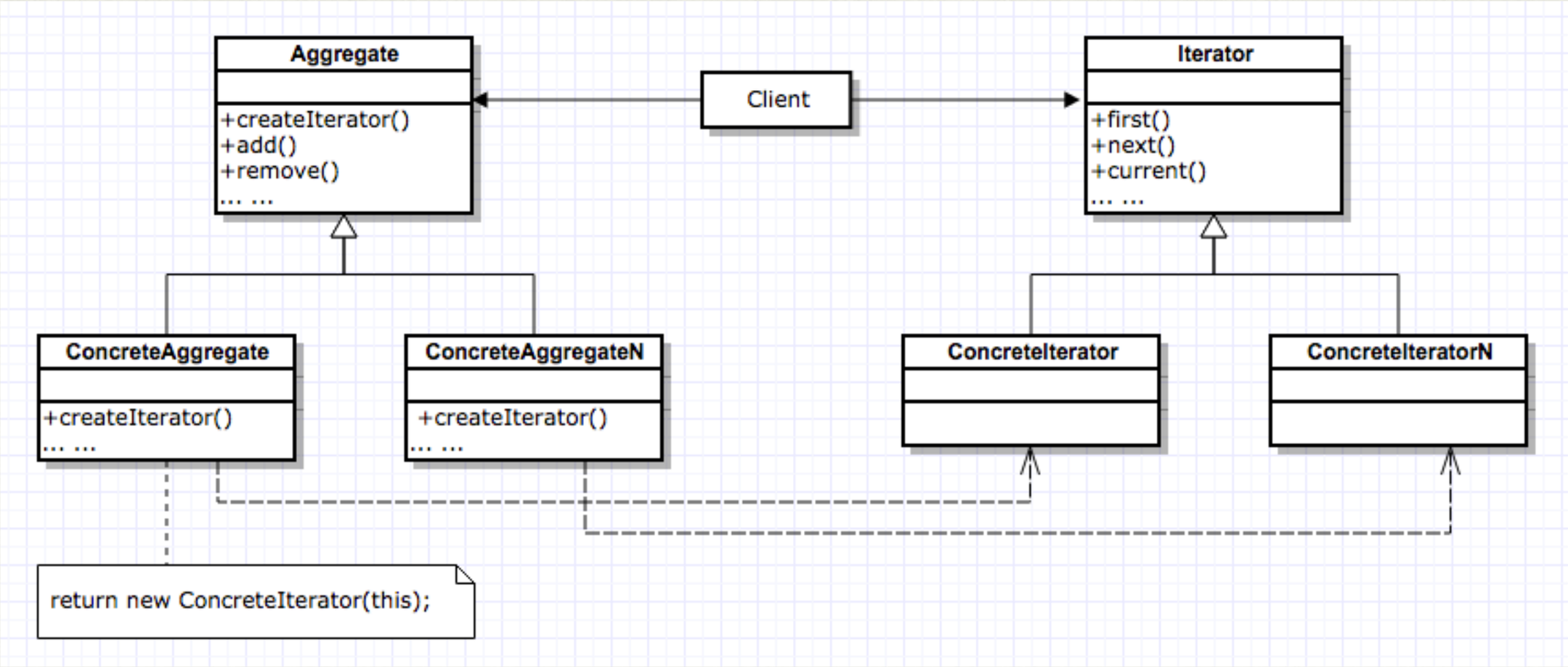
与其它模式的关联：

- 属于Composite模式的一个实例
- 可用Flyweight共享抽象语法树的终端表达式
- 使用Iterator遍历数据结构
- 使用Visitor执行表达式操作

代码请忽略两百行。。。

ITERATOR

迭代器模式：把遍历集合对象的过程抽象出来，这样可以遍历多种数据结构的集合



ITERATOR

JDK源码中的Iterator接口

App.java | List.java | Queue.java | Iterator.java | ListIterator.java | ArrayList.java | HashI

```
 2⊕ * Copyright (c) 1997, 2013, Oracle and/or its affiliates. All rights reserved.
25
26 package java.util;
27
28 import java.util.function.Consumer;
29
31⊕ * An iterator over a collection.  {@code Iterator} takes the place of
54 public interface Iterator<E> {
56⊕     * Returns {@code true} if the iteration has more elements.
62     boolean hasNext();
63
65⊕     * Returns the next element in the iteration.
70     E next();
71
73⊕     * Removes from the underlying collection the last element returned
92⊖     default void remove() {
93         throw new UnsupportedOperationException("remove");
94     }
95
97⊕     * Performs the given action for each remaining element until all elements
113⊖    default void forEachRemaining(Consumer<? super E> action) {
114        Objects.requireNonNull(action);
115        while (hasNext())
116            action.accept(next());
117    }
118 }
119
```

va | ListIterator.java | ArrayList.java | HashMap.java

```java
 * Returns an iterator over the elements in this list in prop
public Iterator<E> iterator() {
    return new Itr();
}

 * An optimized version of AbstractList.Itr
private class Itr implements Iterator<E> {
    int cursor;       // index of next element to return
    int lastRet = -1; // index of last element returned; -1 i
    int expectedModCount = modCount;

    public boolean hasNext() {
        return cursor != size;
    }

    @SuppressWarnings("unchecked")
    public E next() {
        checkForComodification();
        int i = cursor;
        if (i >= size)
            throw new NoSuchElementException();
        Object[] elementData = ArrayList.this.elementData;
        if (i >= elementData.length)
            throw new ConcurrentModificationException();
        cursor = i + 1;
        return (E) elementData[lastRet = i];
    }

    public void remove() {
        if (lastRet < 0)
            throw new IllegalStateException();
        checkForComodification();

        try {
            ArrayList.this.remove(lastRet);
            cursor = lastRet;
            lastRet = -1;
            expectedModCount = modCount;
```

均实现Iterator接口

va | ListIterator.java | ArrayList.java | HashMap.java | Hashtable.java

```java
final class KeySet extends AbstractSet<K> {
    public final int size()                 { return size; }
    public final void clear()               { HashMap.this.clear(); }
    public final Iterator<K> iterator()     { return new KeyIterator(); }
    public final boolean contains(Object o) { return containsKey(o); }
    public final boolean remove(Object key) {
        return removeNode(hash(key), key, null, false, true) != null;
    }
    public final Spliterator<K> spliterator() {
        return new KeySpliterator<>(HashMap.this, 0, -1, 0, 0);
    }
    public final void forEach(Consumer<? super K> action) {
}

abstract class HashIterator {
    Node<K,V> next;        // next entry to return
    Node<K,V> current;     // current entry
    int expectedModCount;  // for fast-fail
    int index;             // current slot

    HashIterator() {

    public final boolean hasNext() {

    final Node<K,V> nextNode() {

    public final void remove() {
}

final class KeyIterator extends HashIterator
    implements Iterator<K> {
    public final K next() { return nextNode().key; }
}

final class ValueIterator extends HashIterator
    implements Iterator<V> {
    public final V next() { return nextNode().value; }
}

final class EntryIterator extends HashIterator
    implements Iterator<Map.Entry<K,V>> {
    public final Map.Entry<K,V> next() { return nextNode(); }
}
```

创建迭代器方法
-各自实现

ITERATOR

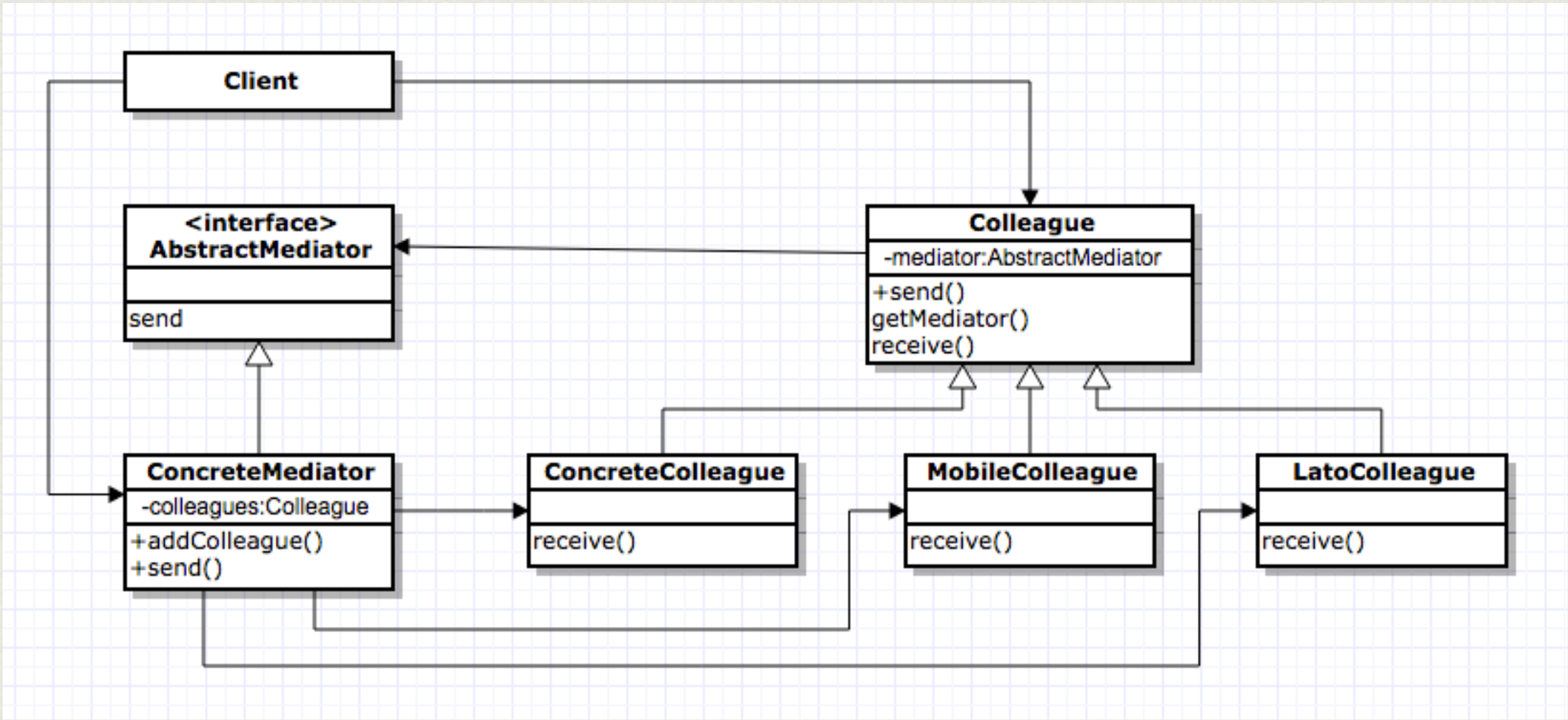
```java
899        final void checkForComodification() {
900            if (modCount != expectedModCount)
901                throw new ConcurrentModificationException();
902        }
903    }
```

```java
926        @SuppressWarnings("unchecked")
927        public E previous() {
928            checkForComodification();
929            int i = cursor - 1;
930            if (i < 0)
931                throw new NoSuchElementException();
932            Object[] elementData = ArrayList.this.elementData;
933            if (i >= elementData.length)
934                throw new ConcurrentModificationException();
935            cursor = i;
936            return (E) elementData[lastRet = i];
937        }
938
939        public void set(E e) {
940            if (lastRet < 0)
941                throw new IllegalStateException();
942            checkForComodification();
943
944            try {
945                ArrayList.this.set(lastRet, e);
946            } catch (IndexOutOfBoundsException ex) {
947                throw new ConcurrentModificationException();
948            }
949        }
950
951        public void add(E e) {
952            checkForComodification();
953
954            try {
955                int i = cursor;
956                ArrayList.this.add(i, e);
957                cursor = i + 1;
958                lastRet = -1;
959                expectedModCount = modCount;
960            } catch (IndexOutOfBoundsException ex) {
961                throw new ConcurrentModificationException();
```

防止遍历元素时更新元素出异常

MEDIATOR

抽象一组对象的交互，解藕发送者和接受者



MEDIATOR

```java
interface AbstractMediator{
    public void send(String msg, Colleague colleague);
}
class ApplicationMediator implements AbstractMediator{
    private List<Colleague> colleagueList;
    public ApplicationMediator(){
        colleagueList = new ArrayList<>();
    }
    public void addColleague(Colleague colleague){
        colleagueList.add(colleague);
    }
    public void send(String msg, Colleague originator){
        System.out.println(originator.toString()+" send: "+msg
        for(Colleague colleague:colleagueList){
            if(colleague != originator){
                colleague.receive(msg);
            }
        }
    }
}
```

```java
abstract class Colleague{
    private AbstractMediator mediator;
    public Colleague(AbstractMediator m){this.mediator = m;}
    public void send(String msg){
        mediator.send(msg, colleague: this);
    }
    public AbstractMediator getMediator(){return mediator;}
    abstract void receive(String msg);
}
class ConcreteColleague extends Colleague{
    public ConcreteColleague(AbstractMediator m) {super(m);}
    public void receive(String msg){
        System.out.println("Colleague received: "+msg);
    }
}
class MobileColleague extends Colleague{
    public MobileColleague(AbstractMediator m) {super(m);}
sg){
le received: "+msg);
```

```java
public static void main(String[] args){
    ApplicationMediator mediator = new ApplicationMediator();
    ConcreteColleague desktop = new ConcreteColleague(mediator);
    MobileColleague mobile = new MobileColleague(mediator);
    LatoColleague lato = new LatoColleague(mediator);
    mediator.addColleague(desktop);
    mediator.addColleague(mobile);
    mediator.addColleague(lato);
    desktop.send( msg: "hello! i am desktop");
    mobile.send( msg: "hi! i am mobile");
```

MediatorDemo

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_131.jdk/Contents/Home/bi
objc[19477]: Class JavaLaunchHelper is implemented in both /Library/
ConcreteColleague@60e53b93 send: hello! i am desktop
Mobile received: hello! i am desktop
Lato received: hello! i am desktop
MobileColleague@5e2de80c send: hi! i am mobile
Colleague received: hi! i am mobile
Lato received: hi! i am mobile
```

统一管理控制

一个Object发送，所有Object接收消息

增加对象只需要继承或实现(这里可以抽象为接口)

MEDIATOR

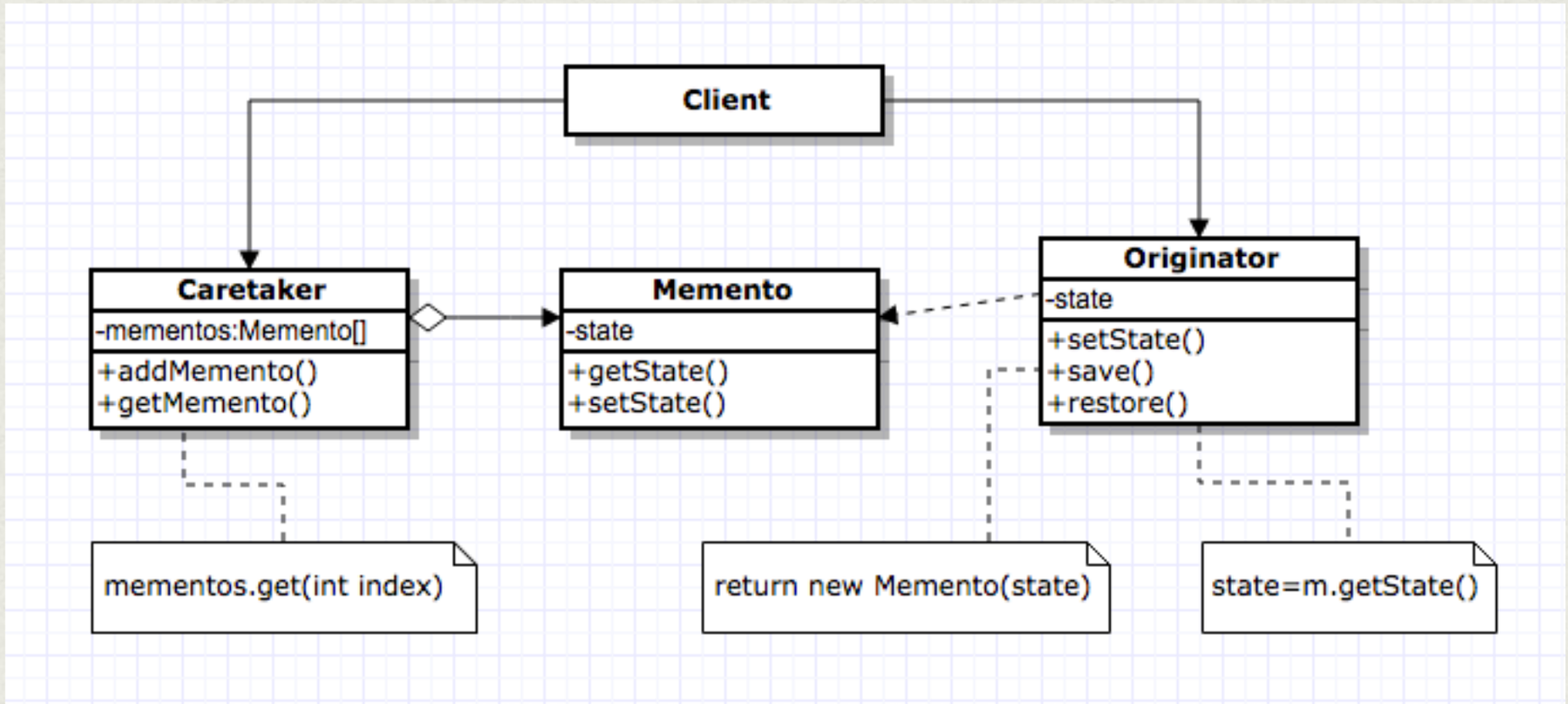
- 避免需要子类化继承
- 减少对象之间的相互依赖，便于系统分层、分模块
- 与Facade不同的是，Facade抽象子系统相关联的操作，对外提供统一的接口；
- Mediator是对一组对象交互的抽象，增加系统复杂度

使用案例：

- java.util.Timer: scheduleXXX()
- java.util.concurrent.Executor#execute()
- java.lang.reflect.Method#invoke()

MEMENTO

抽象对象的内部状态到外部存储(看图就好了0_0)



MEMENTO

抽象出来的对象内部状态

```java
class Memento {
    private String state;
    public Memento(String state) { this.state = state; }
    public String getState() { return state; }
}
class Originator {
    private String state;

    public void setState(String state) {
        System.out.println("Set state:" + state);
        this.state = state;
    }
    public Memento save() {
        System.out.println("Saving to Memento.");
        return new Memento(state);
    }
    public void restore(Memento m) {
        state = m.getState();
        System.out.println("Restore state from: " + state);
    }
}
class Caretaker {
    private ArrayList<Memento> mementos = new ArrayList<>();
    public void addMemento(Memento m) { mementos.add(m); }
    public Memento getMemento(int index) {
        return mementos.get(index);
    }
}
```

外部存储

```java
public class MementoDemo {
    public static void main(String[] args) {
        Caretaker caretaker = new Caretaker();
        Originator originator = new Originator();
        originator.setState("State1");
        originator.setState("State2");
        caretaker.addMemento( originator.save() );
        originator.setState("State3");
        caretaker.addMemento( originator.save() );
        originator.setState("State4");
        originator.restore( caretaker.getMemento( index: 0) )
    }
}
```

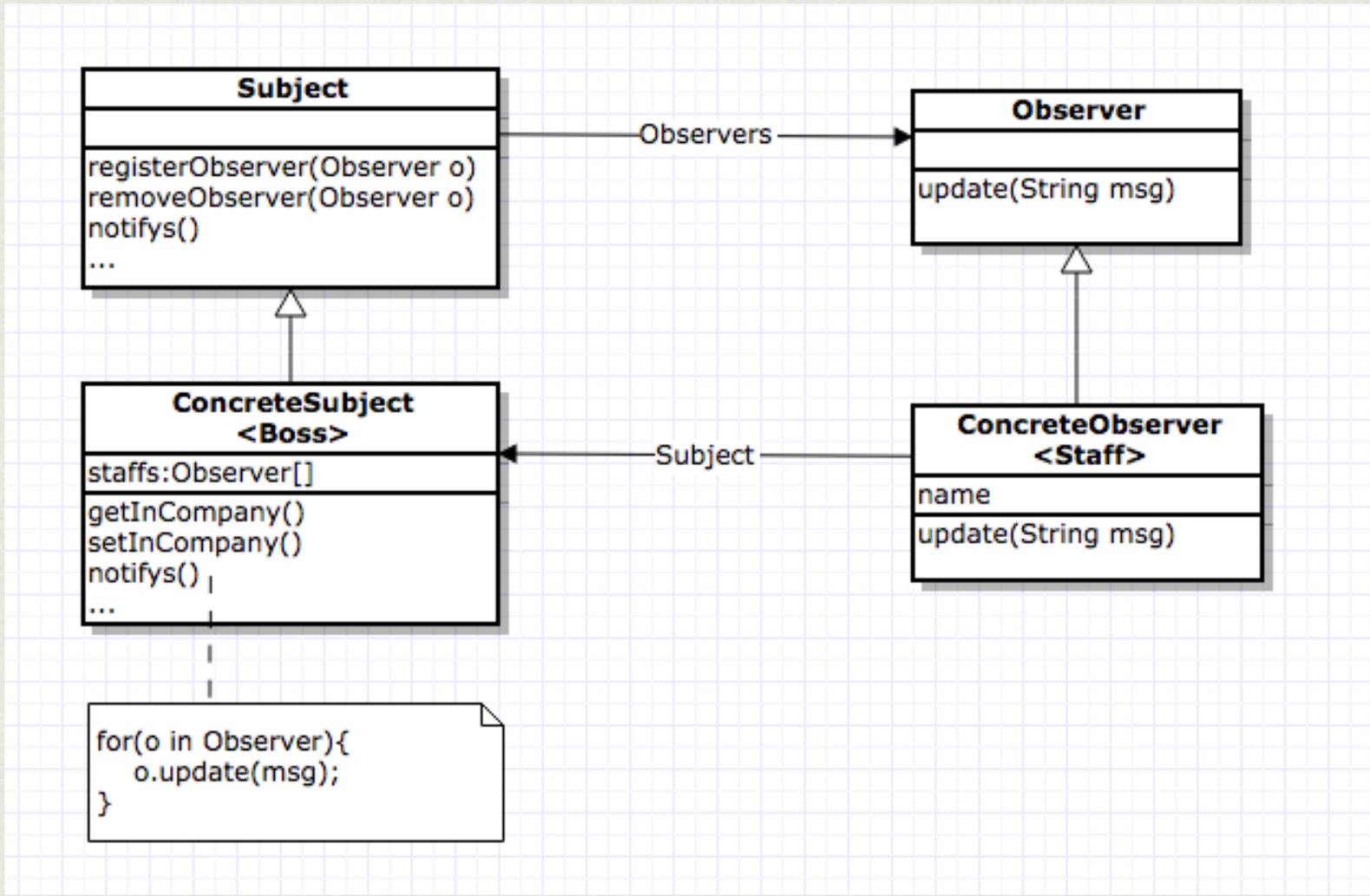
```
ementoDemo
/Library/Java/JavaVirtualMachines/jdk1.8.0_131.jdk/Contents/H
objc[25854]: Class JavaLaunchHelper is implemented in both /L
Set state:State1
Set state:State2
Saving to Memento.
Set state:State3
Saving to Memento.
Set state:State4
Restore state from: State2

Process finished with exit code 0
```

回滚到第一个状态
（注意只存了两个）

- Command包装一个请求对象，利用多态动态匹配；Memento只包装状态，接口有限
- 与Command配合常用于实现数据库事务操作或需要支持撤销、回滚操作的场景

OBSERVER

定义对象间一对多的依赖关系进行消息通讯；观察者模式也称为Pub/Sub模式；
观察者模式常见于MVC框架



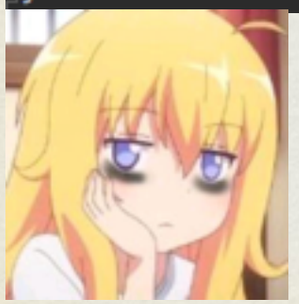
OBSERVER

```java
//pull模式
interface Subject{
    void registerObserver(Observer o);
    void removeObserver(Observer o);
    void notifys();
    boolean isInCompany();
    void setInCompany(boolean b);
}
class Boss implements Subject{
    private List<Observer> staffs = new ArrayList<>();
    private boolean inCompany = true;
    public boolean isInCompany(){
        return this.inCompany;
    }
    public void setInCompany(boolean inCompany){
        this.inCompany = inCompany;
        notifys();
    }
    public void registerObserver(Observer o){
        staffs.add(o);
    }
    public void removeObserver(Observer o){
        this.staffs.remove(o);
    }
    public void notifys(){
        for(Observer staff:staffs){
            staff.update(this.inCompany?
                    "boss is in Company":"boss is not in Company")
        }
    }
}
```

状态更新后通知所有观察者

```java
interface Observer{
    void update(String msg);
}
class Staff implements Observer{
    String name = "";
    public Staff(String name){
        this.name = name;
    }
    public void update(String msg){
        System.out.println(this.name+" receive message:"+msg);
    }
}
public class ObserverDemo {
    public static void main(String[] args){
        Subject boss = new Boss();
        Observer john = new Staff( name: "John");
        boss.registerObserver(john);
        boss.registerObserver(new Staff( name: "Alias"));
        boss.registerObserver(new Staff( name: "Sam"));
        boss.setInCompany(false);
        System.out.println(" ------------------------- ");
        boss.removeObserver(john);
        boss.setInCompany(true);
    }
```

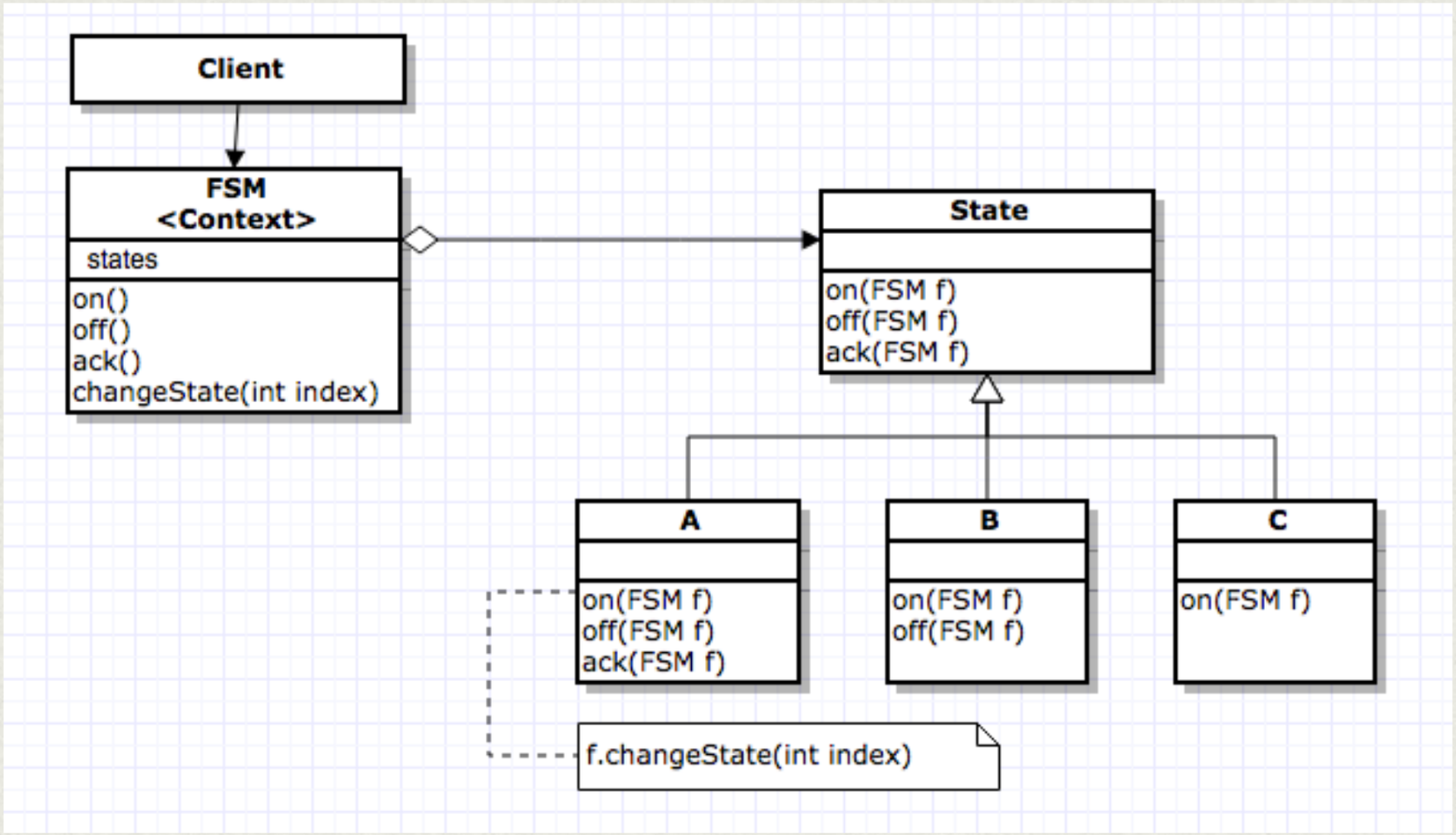
```
serverDemo
John receive message:boss is not in Company
Alias receive message:boss is not in Company
Sam receive message:boss is not in Company
 -------------------------
Alias receive message:boss is in Company
Sam receive message:boss is in Company
```



push模式有时间再做。。。

STATE

状态机：根据更改的自身状态执行不同的操作(Wrapper + 多态 + 交互匹配对象)



使用案例：

- TCP协议

```java
class FSM {
    private State[] states  = {new A(), new B(), new C()};
    private int currentState = 0;

    public void on()  {
        states[currentState].on(this);
    }
    public void off() {
        states[currentState].off(this);
    }
    public void ack() {
        states[currentState].ack(this);
    }
    public void changeState(int index) {
        currentState = index;
    }
}
```

```java
abstract class State {
    public void on(FSM fsm) {
        System.out.println("error");
    }
    public void off(FSM fsm) {
        System.out.println("error");
    }

    public void ack(FSM fsm) {
        System.out.println("error");
    }
}

class A extends State {
    public void on(  FSM fsm ) {
        System.out.println("A + on");
        fsm.changeState(2);
    }
    public void off(FSM fsm) {
        System.out.println("A + off");
        fsm.changeState(1);
    }
    public void ack(FSM fsm) {
        System.out.println("A + ack");
        fsm.changeState(0);
    }
}
```

```java
public static void main(String[] args) {
    FSM fsm  = new FSM();
    int[] msgs = {2, 1, 2, 1, 0, 2, 0, 0};
    for (int msg : msgs) {
        if (msg == 0) {
            fsm.on();
        } else if (msg == 1) {
            fsm.off();
        } else if (msg == 2) {
            fsm.ack();
        }
    }
```

Wrapper

Context

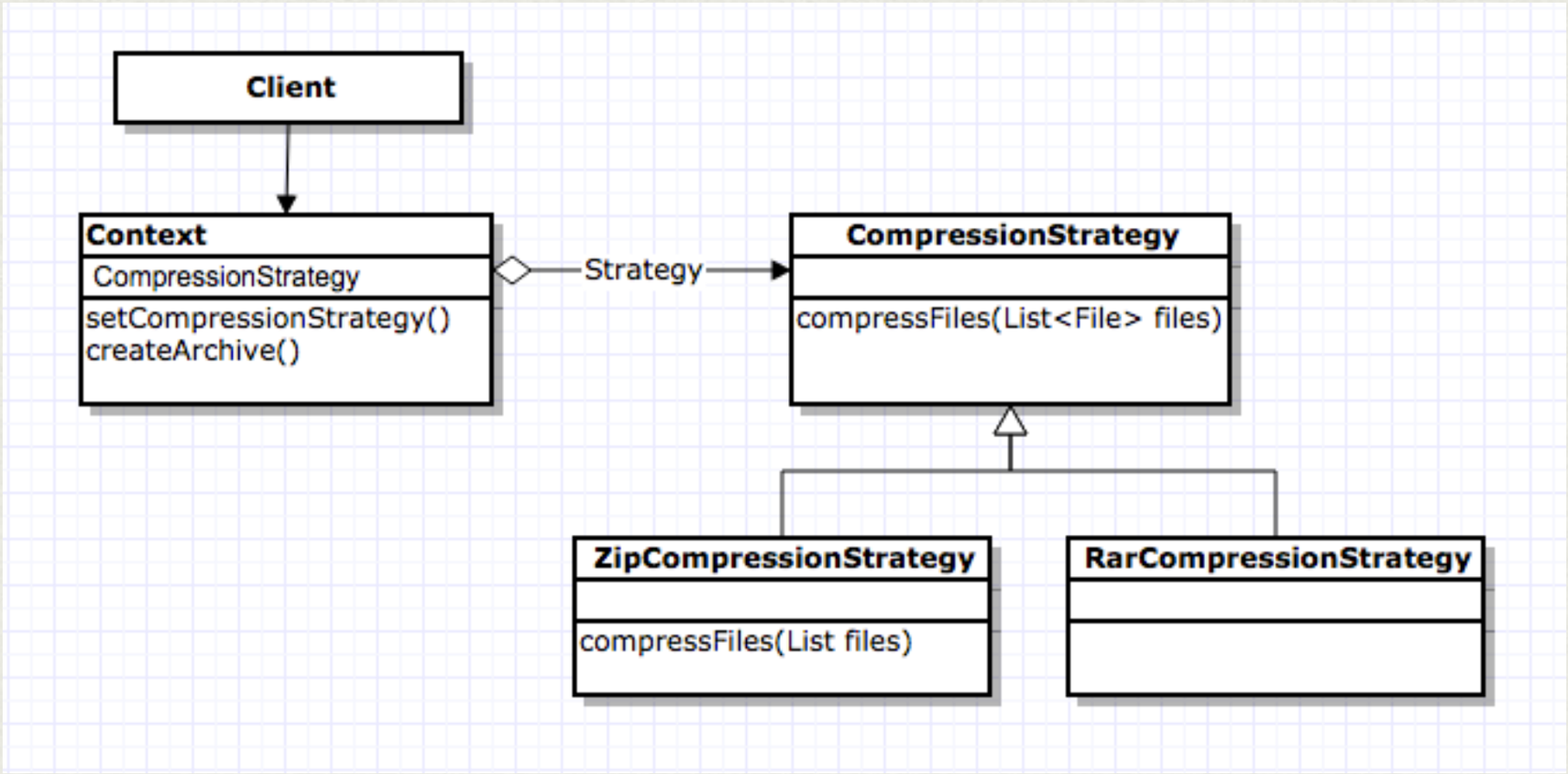
Callback

多态动态匹配到不同的状态对象上

STATE

STRATEGY

封装一系列算法，与State一样(通过wrapper+多态动态匹配算法类)



- 与模版方法、状态机相似，有相同的结构图，解决问题不同
- 策略可有多种算法实现

STRATEGY

```
interface CompressionStrategy {
   public void compressFiles(List<File> files);
}
class ZipCompressionStrategy implements CompressionStrategy {
   public void compressFiles(List<File> files) {
      System.out.println("ZIP approach");
   }
}
class RarCompressionStrategy implements Co
   public void compressFiles(List<File> files) {
      System.out.println("RAR approach");
   }
}
```

```
class CompressionContext {
   private CompressionStrategy strategy;

   public void setCompressionStrategy(CompressionStrategy strategy) {
      this.strategy = strategy;
   }

   public void createArchive(List<File> files) {
      strategy.compressFiles(files);
   }
}
```

运行时设置算法

匹配对应算法

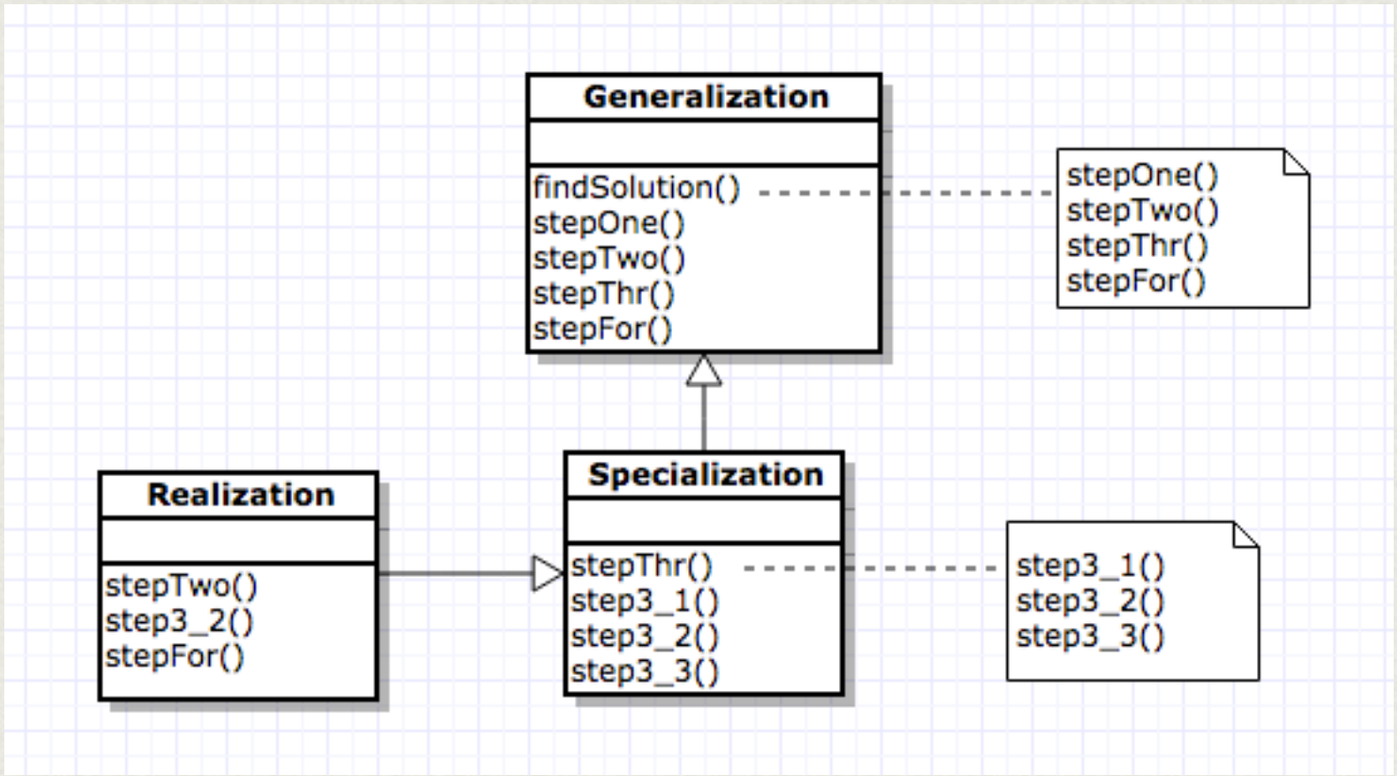
```
public static void main(String[] args) {
   List <File> fileList = new ArrayList<File>();
   fileList.add(new File("./a.png"));
   fileList.add(new File("./b.gif"));
   fileList.add(new File("./c.jpg"));
   CompressionContext ctx = new CompressionContext();
   ctx.setCompressionStrategy(new ZipCompressionStrategy());
   ctx.createArchive(fileList);
}
```

结构图与State一致

State更具灵活性

TEMPLATE METHOD

抽象算法操作步骤，提供占位符运行时变换算法步骤



- 模版方法与策略的区别在于抽象粒度
- 模版方法应用于类上，通过继承改变算法的一部分；策略应用于对象上，通过代理改变整个算法
- 模版修改整个类的逻辑，策略修改单个对象的逻辑
- 工厂方法是模版方法的一个特例
- 模版方法是典型的依赖反转

TEMPLATE METHOD

```
abstract class Generalization {
  void findSolution() {
      stepOne();
      stepTwo();
      stepThr();
      stepFor();
  }
  private void stepOne() {
      System.out.println("Generalization.stepOne");
  }

  abstract void stepTwo();
  abstract void stepThr();

  void stepFor() {
      System.out.println( "Generalization.stepFor" );
  }
}

public class TemplateMethodDemo {
  public static void main(String[] args) {
      Generalization algorithm = new Realization();
      algorithm.findSolution();
  }
}
```

算法步骤骨架

通用部分

占位符：自定义部分

替换小骨架 step3_2部分

```
abstract class Specialization extends Generalization {
  protected void stepThr() {
      step3_1();
      step3_2();
      step3_3();
  }
  //
  private void step3_1() {
      System.out.println("      Specialization.step3_1");
  }
  //
  abstract protected void step3_2();
  //
  private void step3_3() {
      System.out.println("      Specialization.step3_3");
  }
}
class Realization extends Specialization {
  protected void stepTwo() {
      System.out.println("    Realization.stepTwo");
  }
  protected void step3_2() {
      System.out.println("      Realization.step3_2");
  }
  protected void stepFor() {
      System.out.println("    Realization.stepFor");
      super.stepFor();
  }
}
```

替换骨架stepThr占位符,拆分新的框架

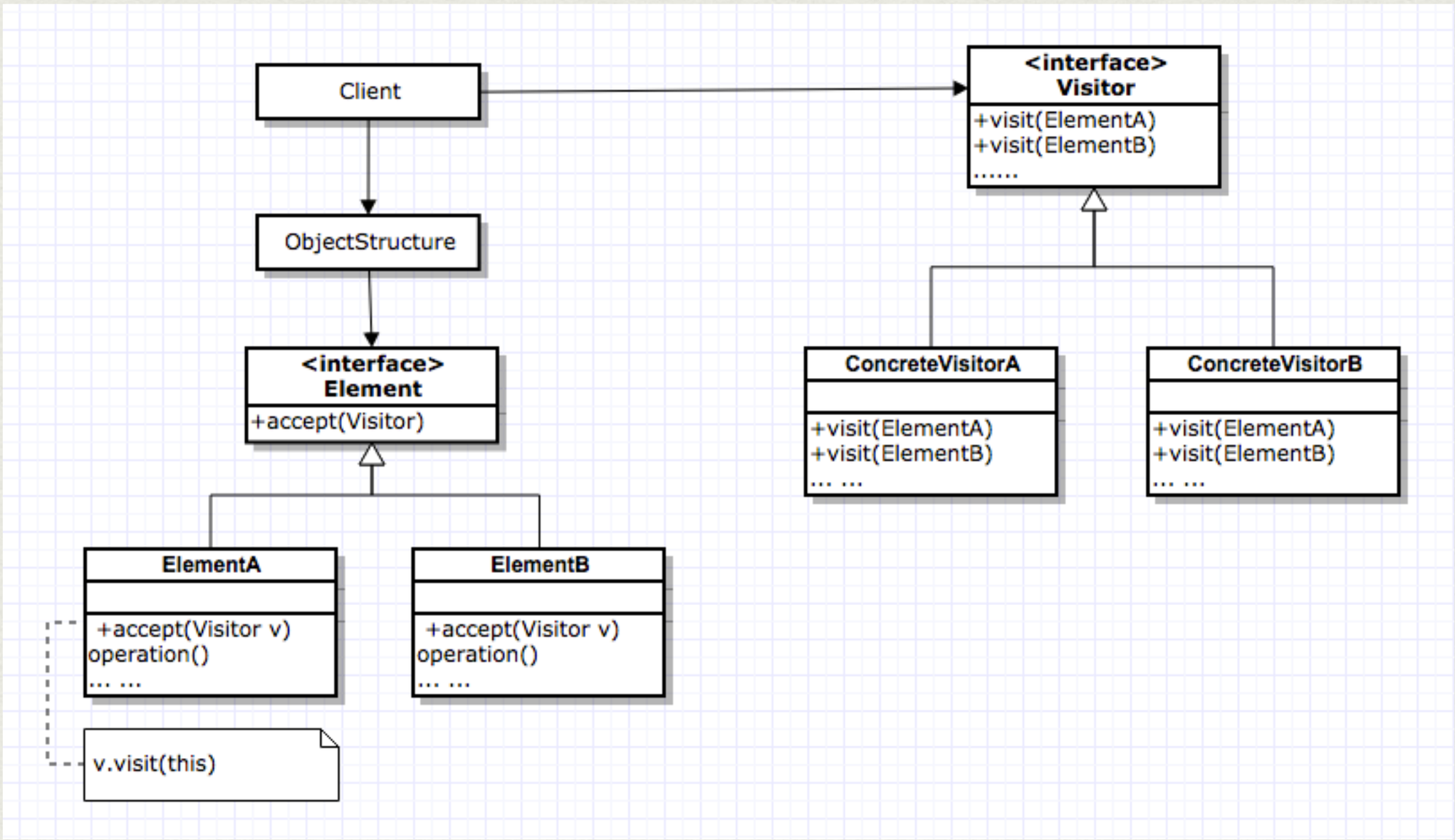
新的占位符

替换骨架 stepTwo 占位符

模版方法框架上使用大大的...

VISITOR

抽象一组代表不同数据结构的算法，应用于复杂结构的对象



VISITOR：示例

利用多态既校验Visitor的类型
又校验element的类型，确定
某个对象绑定执行那些操作

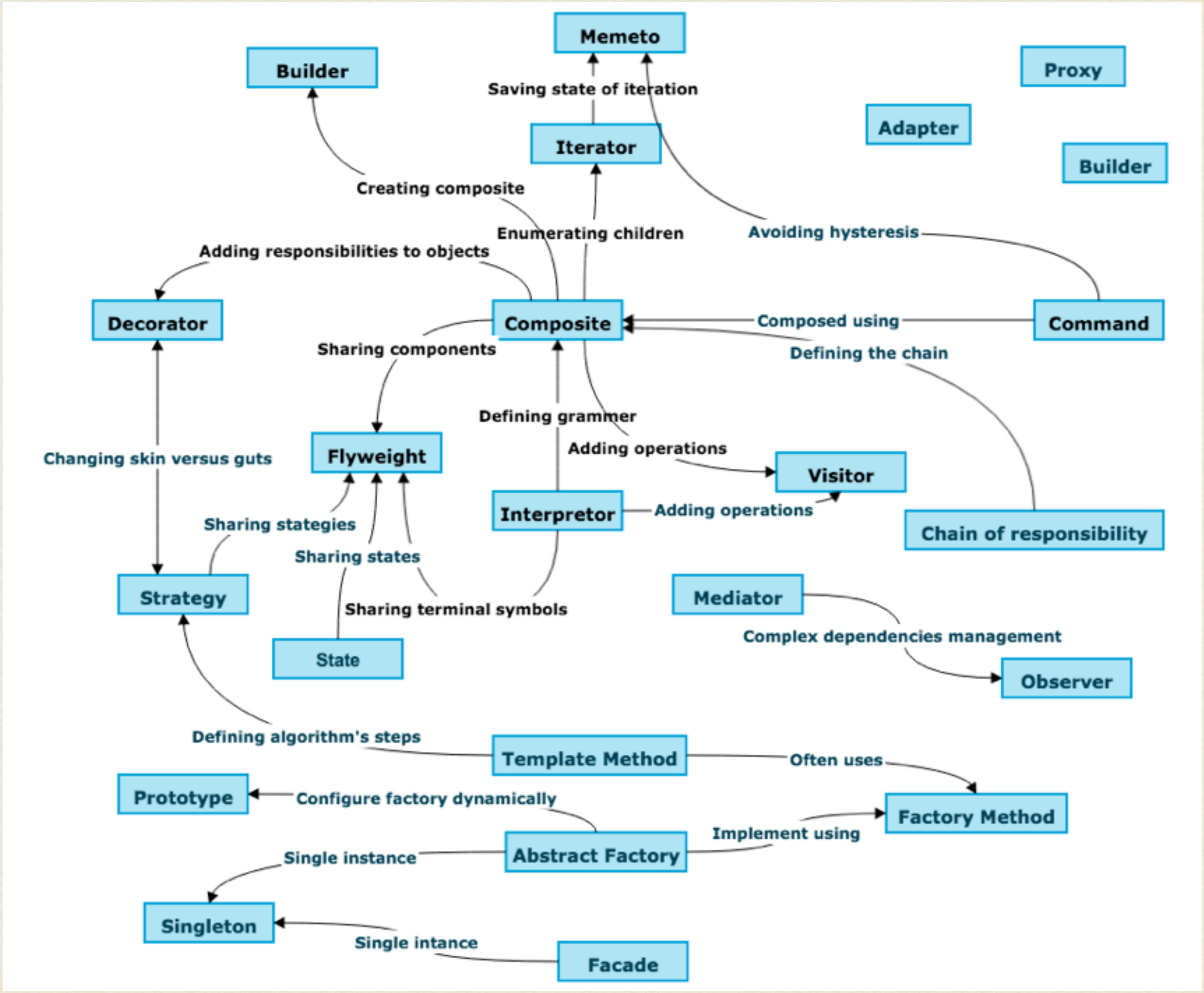
```
interface Visitor{
    void visit(One one);
    void visit(Two two);
    void visit(Three three);
}
class UpVisitor implements Visitor{
    public void visit(One one){
        System.out.println("do Up on "+one.toString());
    }
    public void visit(Two two){
        System.out.println("do Up on "+two.toString());
    }
    public void visit(Three three){
        System.out.println("do Up on "+three.toString());
    }
}
class DownVisitor implements Visitor{
    public void visit(One one){
        System.out.println("do Down on "+one.toString());
    }
    public void visit(Two two){
        System.out.println("do Down on "+two.toString());
    }
    public void visit(Three three){
        System.out.println("do Down on "+three.toString());
    }
}
```

```
interface Element{
    void accept(Visitor v);

class One implements Element{
    public void accept(Visitor v){
        v.visit(this);}
}
class Two implements Element{
    public void accept(Visitor v){
        v.visit(this);}
}
class Three implements Element{
    public void accept(Visitor v){
        v.visit(this);}
}

public class VisitorDemo {
    public static void main(String[] args){
        Element[] elements = {new One(),new Two(),new
Three()};
        UpVisitor up = new UpVisitor();
        DownVisitor down = new DownVisitor();
        for(Element elm:elements){
            elm.accept(up);
            elm.accept(down);
        }
    }
}
```

VISITOR

- 不同结构的元素使用不同的算法，遵循单一职责原则
- 抽象相似算法，分离算法与数据结构，更易于扩展
- 可维护复杂对象的操作
- 为了访问元素属性，需要方法是public，违法开放关闭原则

设计模式关系图


Memeto
Builder
Proxy
Saving state of iteration
Adapter
Iterator
Builder
Creating composite
Enumerating children
Avoiding hysteresis
Adding responsibilities to objects
Decorator
Composite
Composed using
Command
Sharing components
Defining the chain
Defining grammer
Adding operations
Changing skin versus guts
Flyweight
Visitor
Interpretor
Adding operations
Chain of responsibility
Sharing stategies
Sharing states
Strategy
Mediator
Sharing terminal symbols
Complex dependencies management
State
Observer
Defining algorithm's steps
Template Method
Often uses
Prototype
Configure factory dynamically
Factory Method
Implement using
Abstract Factory
Single instance
Singleton
Single intance
Facade

看到的理解的不理解的都凑一起了。。。

有些可用Lambda替换，不熟悉，需要多演练

我只能伤心的啃着咸鱼

Thank you!

-xieqiaoyun