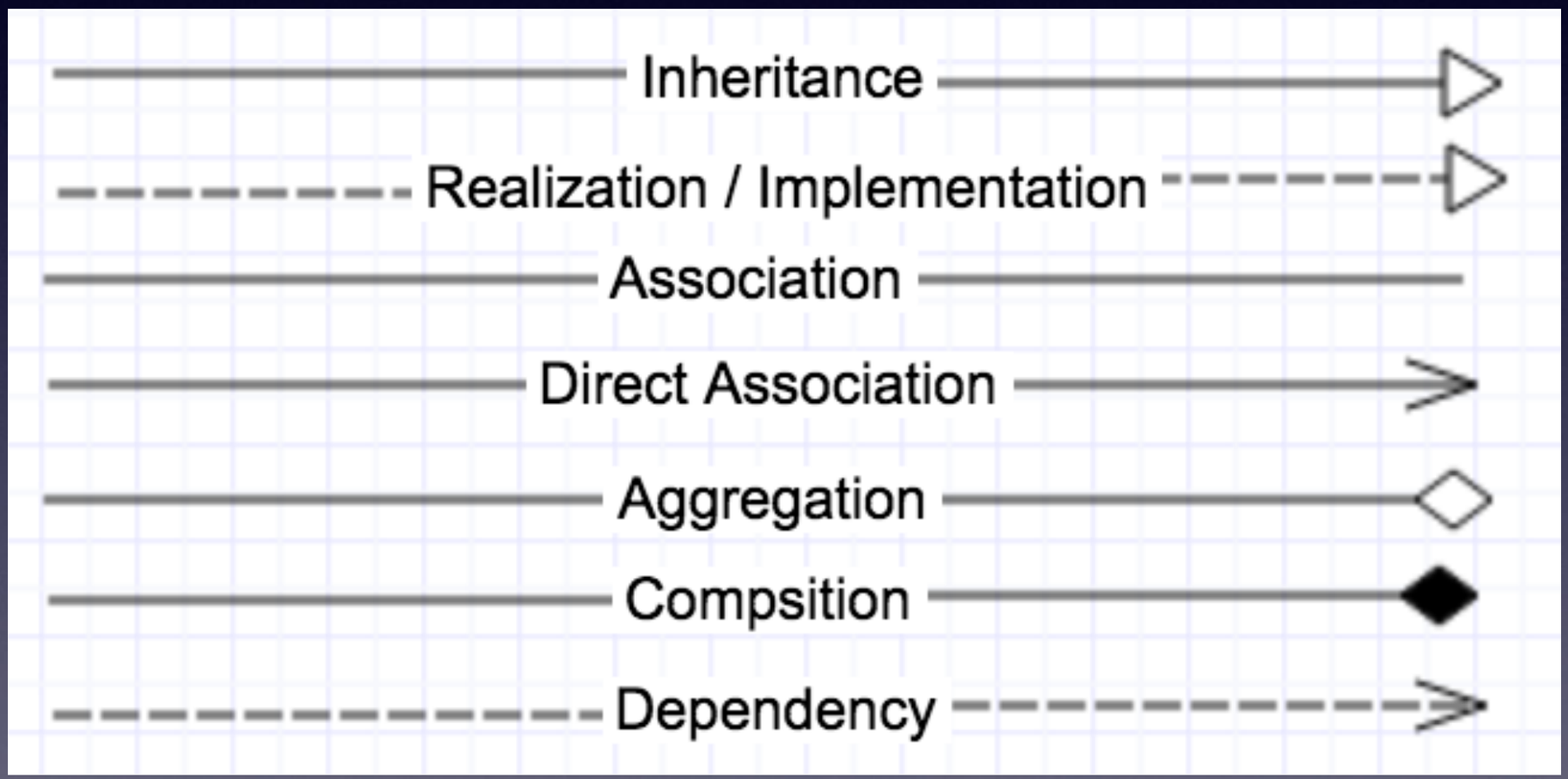
设计模式

@Tim Qi

UML, Relationship



模式分类

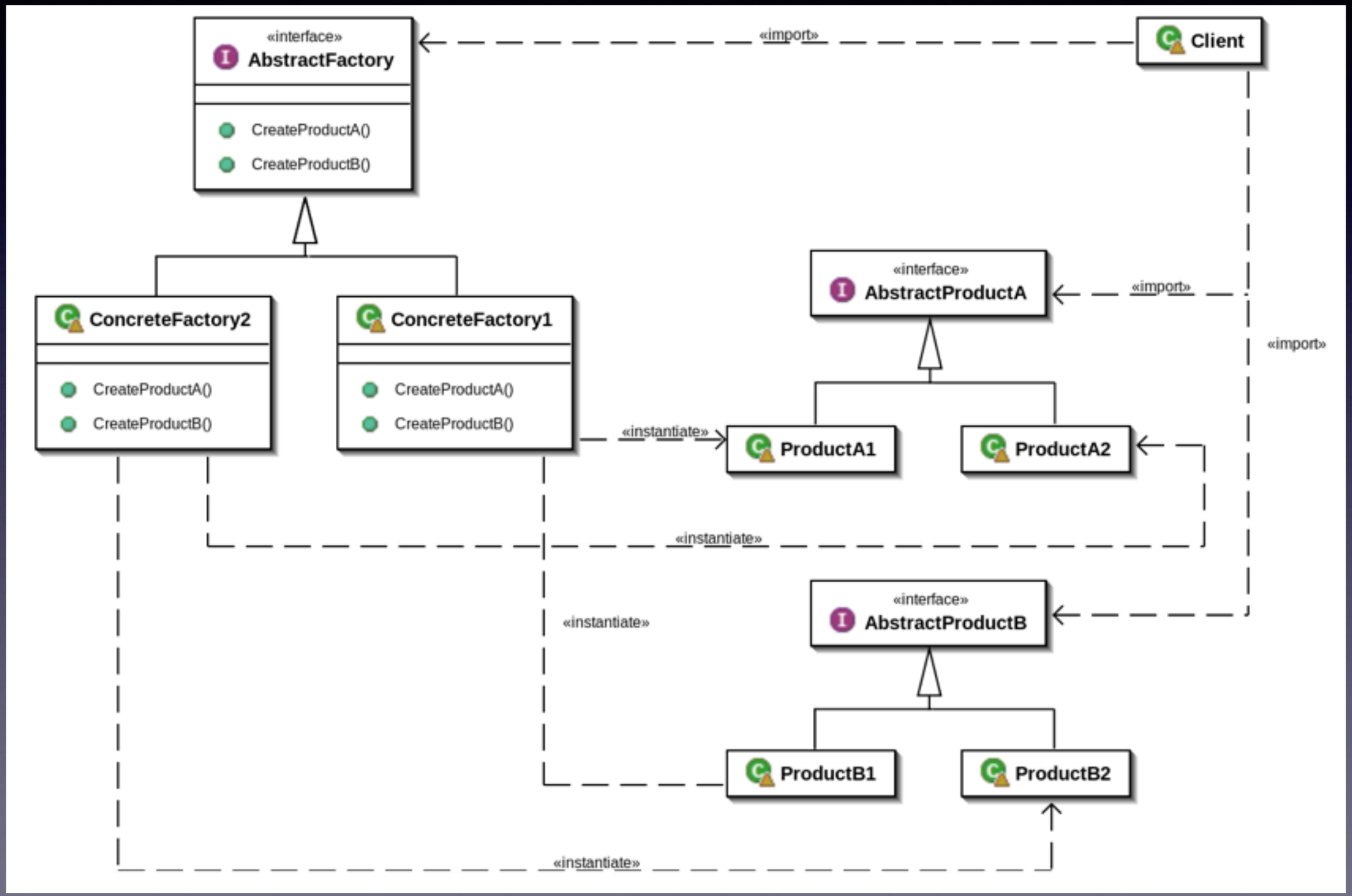
- 创建型
- 结构型
- 行为型

创建型设计模式

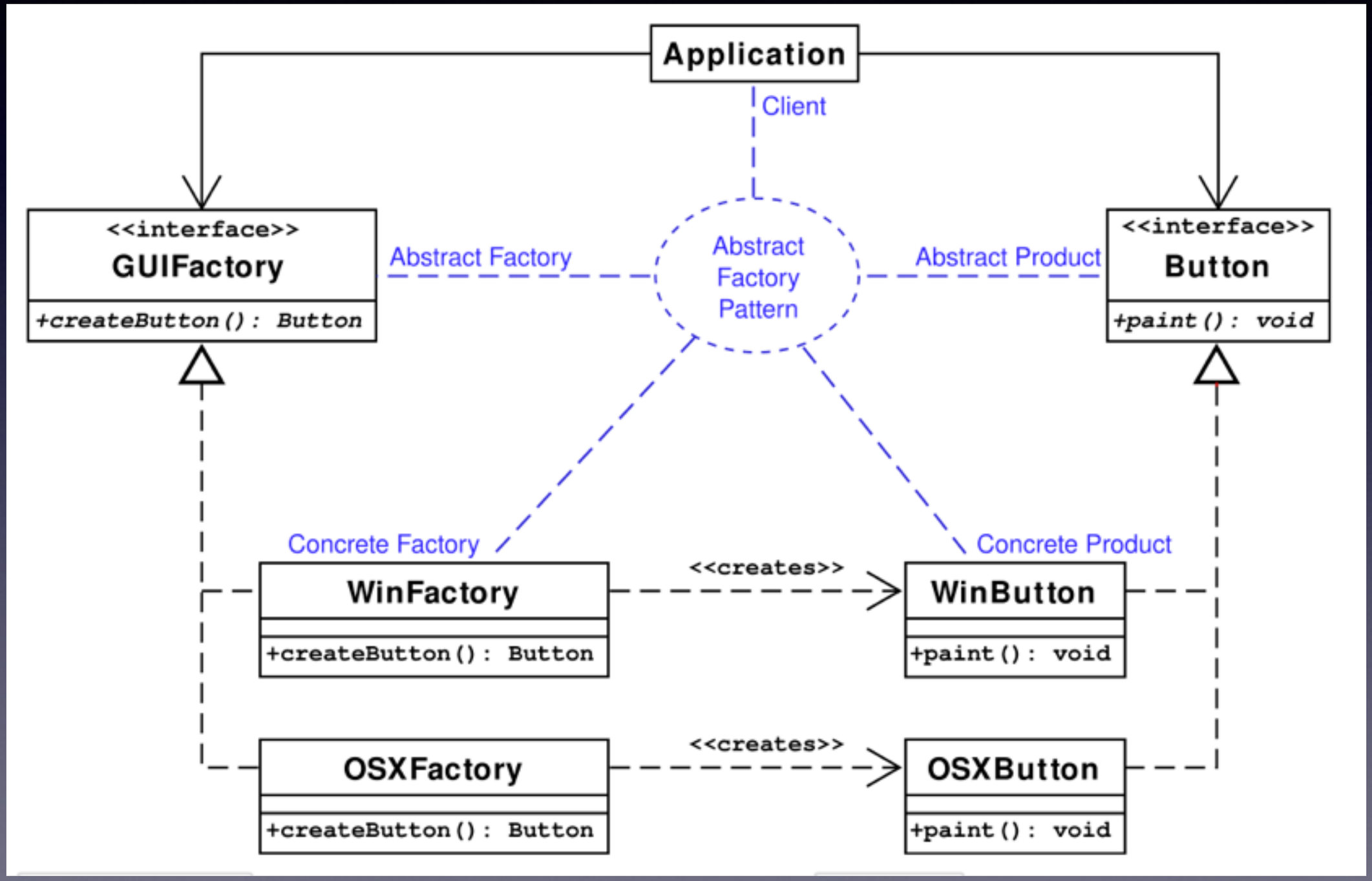
创建型设计模式

- Abstract Factory 抽象工厂
- Builder 生成器
- Factory Method 工厂方法
- Prototype 原型
- Singleton 单例

Abstract Factory



Abstract Factory



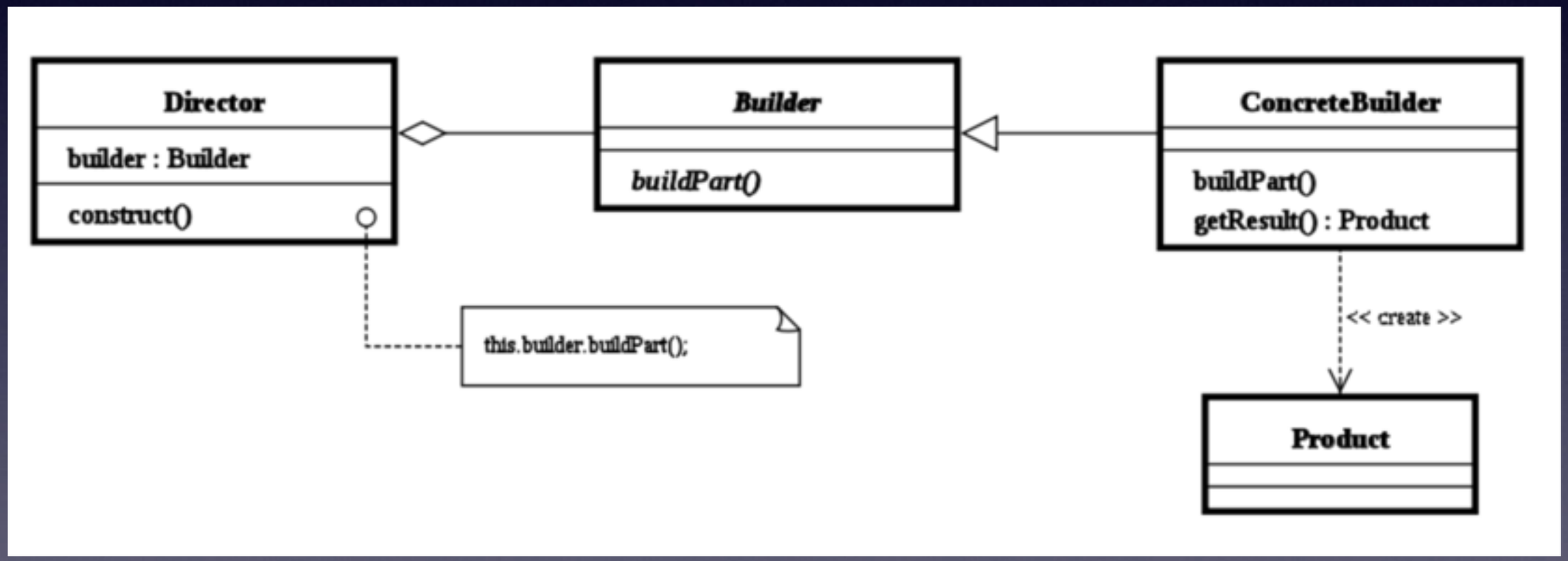
Abstract Factory

对一系列工厂方法进行「封装」；

对工厂的产品进行「封装」；

客户端不关心到底是哪个工厂生产了什么产品

Builder


Director
builder : Builder
construct()
Builder
buildPart()
ConcreteBuilder
buildPart()
getResult() : Product
this.builder.buildPart();
<< create >>
Product

Builder

```
class Car is
  Can have GPS, trip computer and various numbers of seats.
  Can be a city car, a sports car, or a cabriolet.

class CarBuilder is
  method getResult() is
      output:  a Car with the right options
    Construct and return the car.

  method setSeats(number) is
      input:  the number of seats the car may have.
    Tell the builder the number of seats.

method setCityCar() is
  Make the builder remember that the car is a city car.

method setCabriolet() is
  Make the builder remember that the car is a cabriolet.

method setSportsCar() is
  Make the builder remember that the car is a sports car.

method setTripComputer() is
  Make the builder remember that the car has a trip compute

method unsetTripComputer() is
  Make the builder remember that the car does not have a trip computer.

  method unsetGPS() is
    Make the builder remember that the car doe

Construct a CarBuilder called carBuilder
carBuilder.setSeats(2)
carBuilder.setSportsCar()
carBuilder.setTripComputer()
carBuilder.unsetGPS()
car := carBuilder.getResult()
```

Of course one could dispense with Builder and just do this:

```
car = new Car();
car.seats = 2;
car.type = CarType.SportsCar;
car.setTripComputer();
car.unsetGPS();
car.isValid();
```

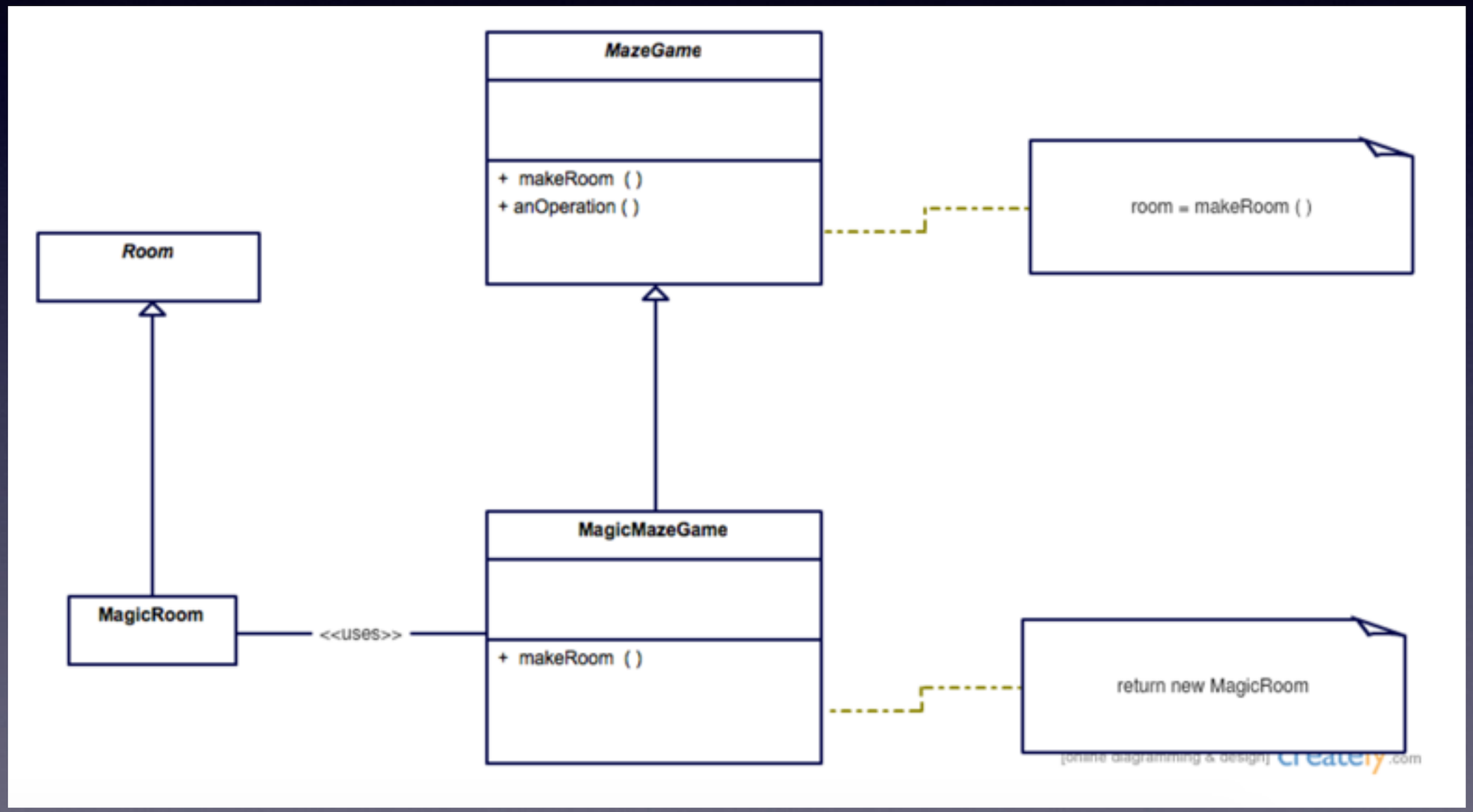
Builder

望远镜模式的改进，对大量参数逐步构建；

一次性构建对象，避免并发访问数据不一致；

Builder 类会占用额外的存储空间

Factory Method



Factory Method

```php
/* Factory and car interfaces */

interface CarFactory
{
    public function makeCar();
}

interface Car
{
    public function getType();
}
```

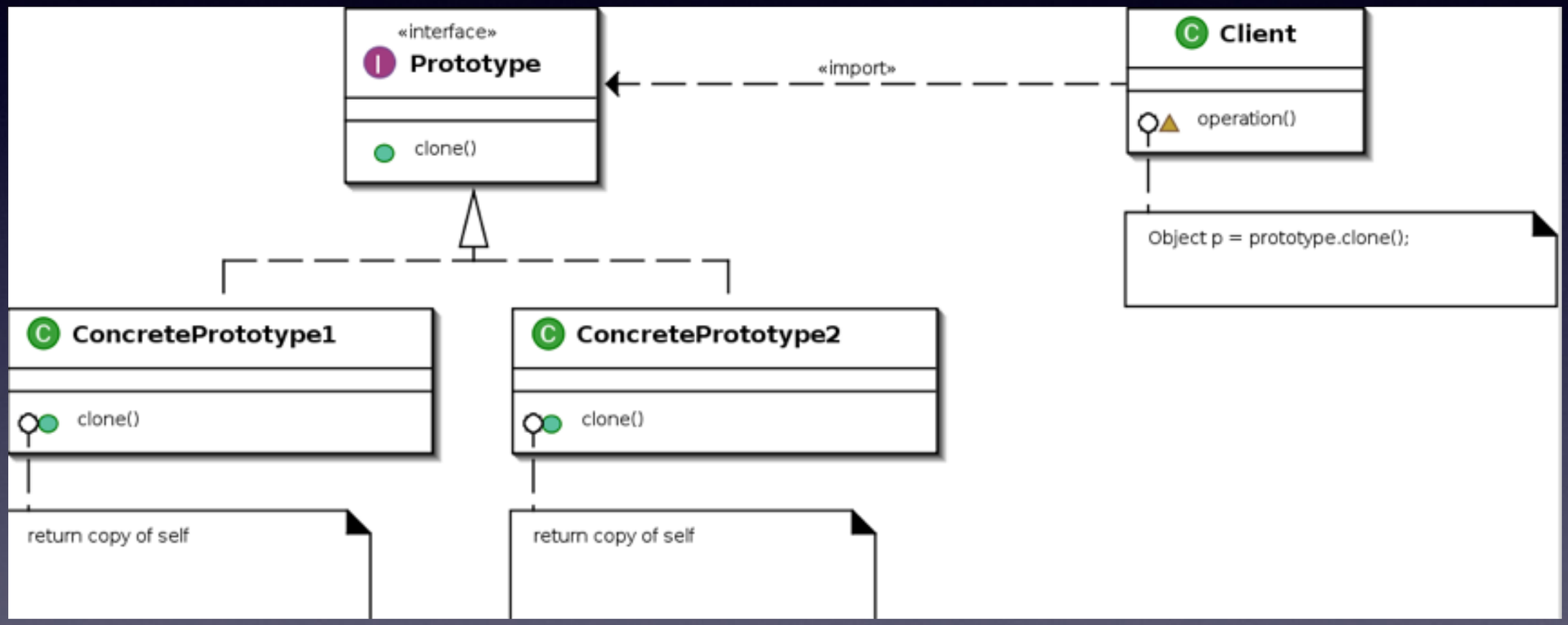
```php
/* Concrete implementations of the factory and car */

class SedanFactory implements CarFactory
{
    public function makeCar()
    {
        return new Sedan();
    }
}

class Sedan implements Car
{
    public function getType()
    {
        return 'Sedan';
    }
}
```

```php
/* Client */

$factory = new SedanFactory();
$car = $factory->makeCar();
print $car->getType();
```

Factory Method

将对象创建的职责单一化到一个类中，
隐藏创建细节，外部不需要了解如何创建相关对象

Prototype


«interface»
Prototype
clone()
«import»
Client
operation()
Object p = prototype.clone();
ConcretePrototype1
clone()
return copy of self
ConcretePrototype2
clone()
return copy of self

Prototype

使用 clone 的方法创建新对象而不是 new，
提高效率，
避免了客户端对工厂方法继承引发错误

Singleton

```
public final class Singleton {
    private static final Singleton instance = new Singleton();
    private Singleton() {}

    public static Singleton getInstance() {
        return instance;
    }
}
```

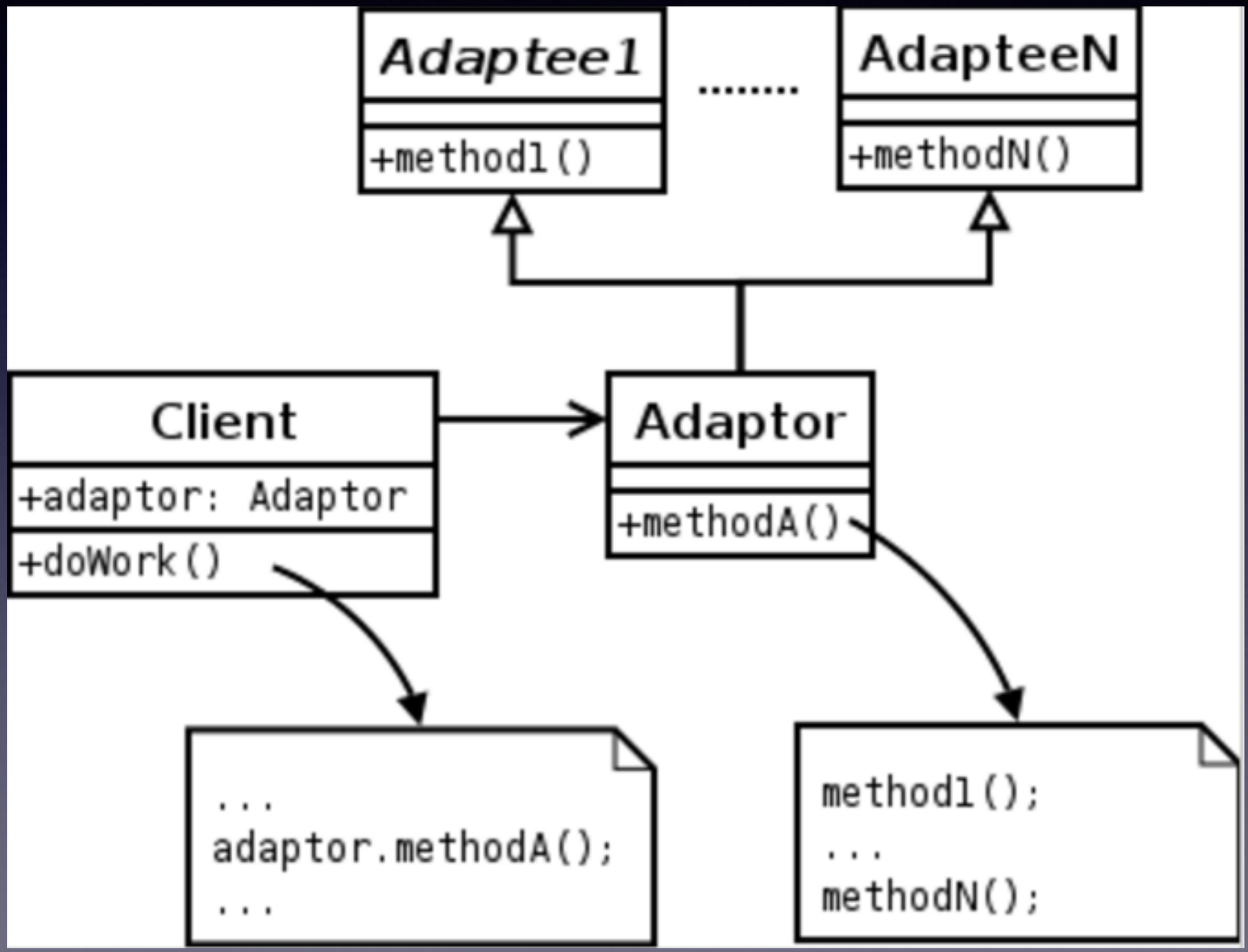
```
public final class Singleton {
    private static Singleton instance = null;
    private Singleton() {}

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

结构型设计模式

结构型设计模式

- Adapter 适配器
- Bridge 桥接
- Composite 组合
- Decorator 装饰
- Facade 外观
- Flyweight 享元
- Proxy 代理

Adapter (or Wrapper)



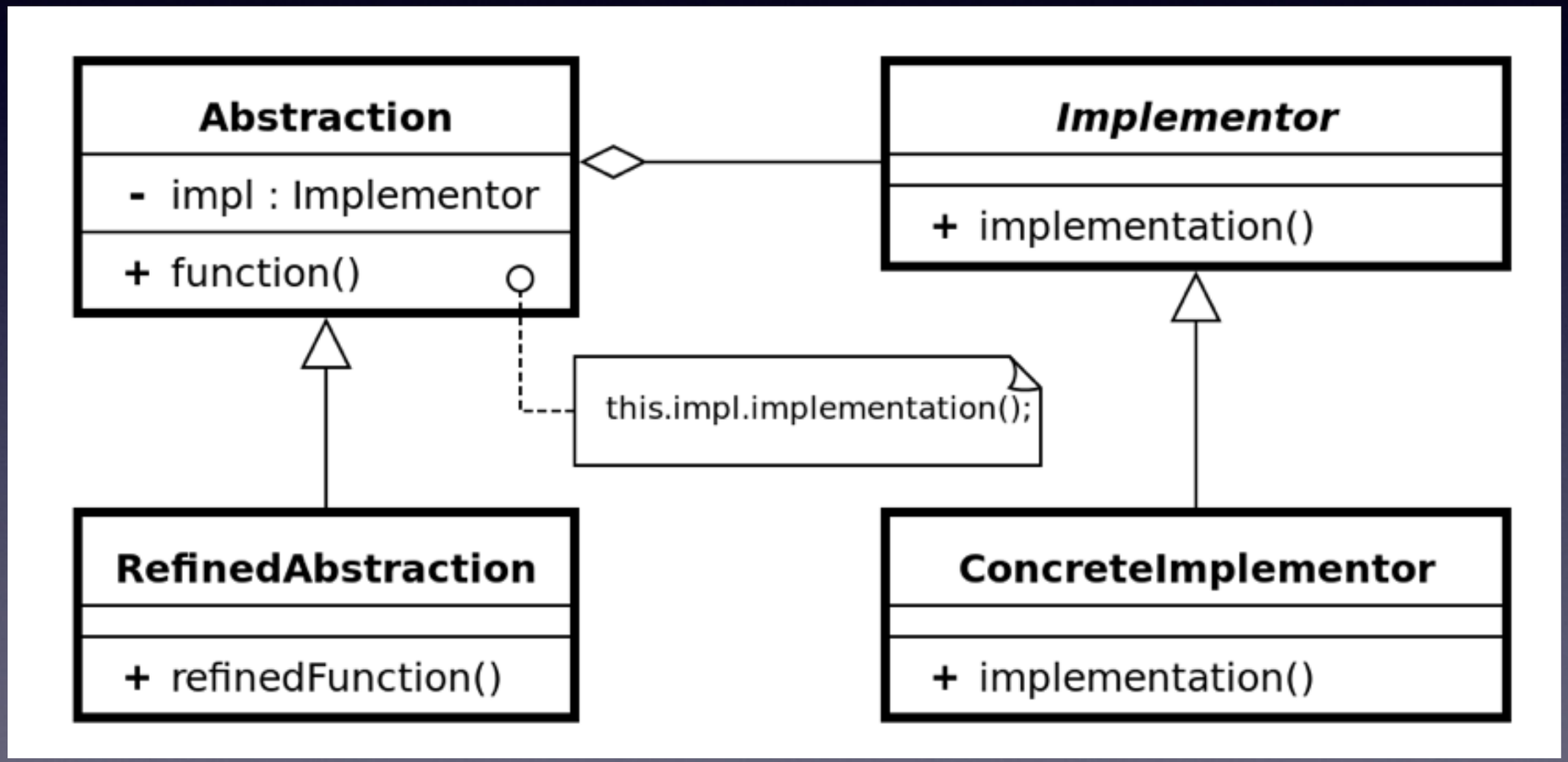
Adapter (or Wrapper)

```
public class AdapteeToClientAdapter implements Adapter {

    private final Adaptee instance;

    public AdapteeToClientAdapter(final Adaptee instance) {
         this.instance = instance;
    }

    @Override
    public void clientMethod() {
       // call Adaptee's method(s) to implement Client's clientMethod
    }

}
```

Adapter (or Wrapper)

使一个现有类似功能类能够适应新接口；

Bridge


Abstraction
- impl : Implementor
+ function()
Implementor
+ implementation()
this.impl.implementation();
RefinedAbstraction
+ refinedFunction()
ConcreteImplementor
+ implementation()

Bridge

```
/** "Implementor" */
interface DrawingAPI {
    public void drawCircle(double x, double y, double radius);
}

/** "ConcreteImplementor"  1/2 */
class DrawingAPI1 implements DrawingAPI {
    public void drawCircle(double x, double y, double radius) {
        System.out.printf("API1.circle at %f:%f radius %f\n", x, y, radius);
    }
}

/** "ConcreteImplementor" 2/2 */
class DrawingAPI2 implements DrawingAPI {
    public void drawCircle(double x, double y, double radius) {
        System.out.printf("API2.circle at %f:%f radius %f\n", x, y, radius);
    }
}
```

Bridge

```
/** "Abstraction" */
abstract class Shape {
    protected DrawingAPI drawingAPI;

    protected Shape(DrawingAPI drawingAPI){
        this.drawingAPI = drawingAPI;
    }

    public abstract void draw();                                    // low-level
    public abstract void resizeByPercentage(double pct);     // high-level
}

/** "Refined Abstraction" */
class CircleShape extends Shape {
    private double x, y, radius;
    public CircleShape(double x, double y, double radius, DrawingAPI drawingAPI) {
        super(drawingAPI);
        this.x = x;  this.y = y;  this.radius = radius;
    }

    // low-level i.e. Implementation specific
    public void draw() {
        drawingAPI.drawCircle(x, y, radius);
    }
    // high-level i.e. Abstraction specific
    public void resizeByPercentage(double pct) {
        radius *= (1.0 + pct/100.0);
    }
}
```

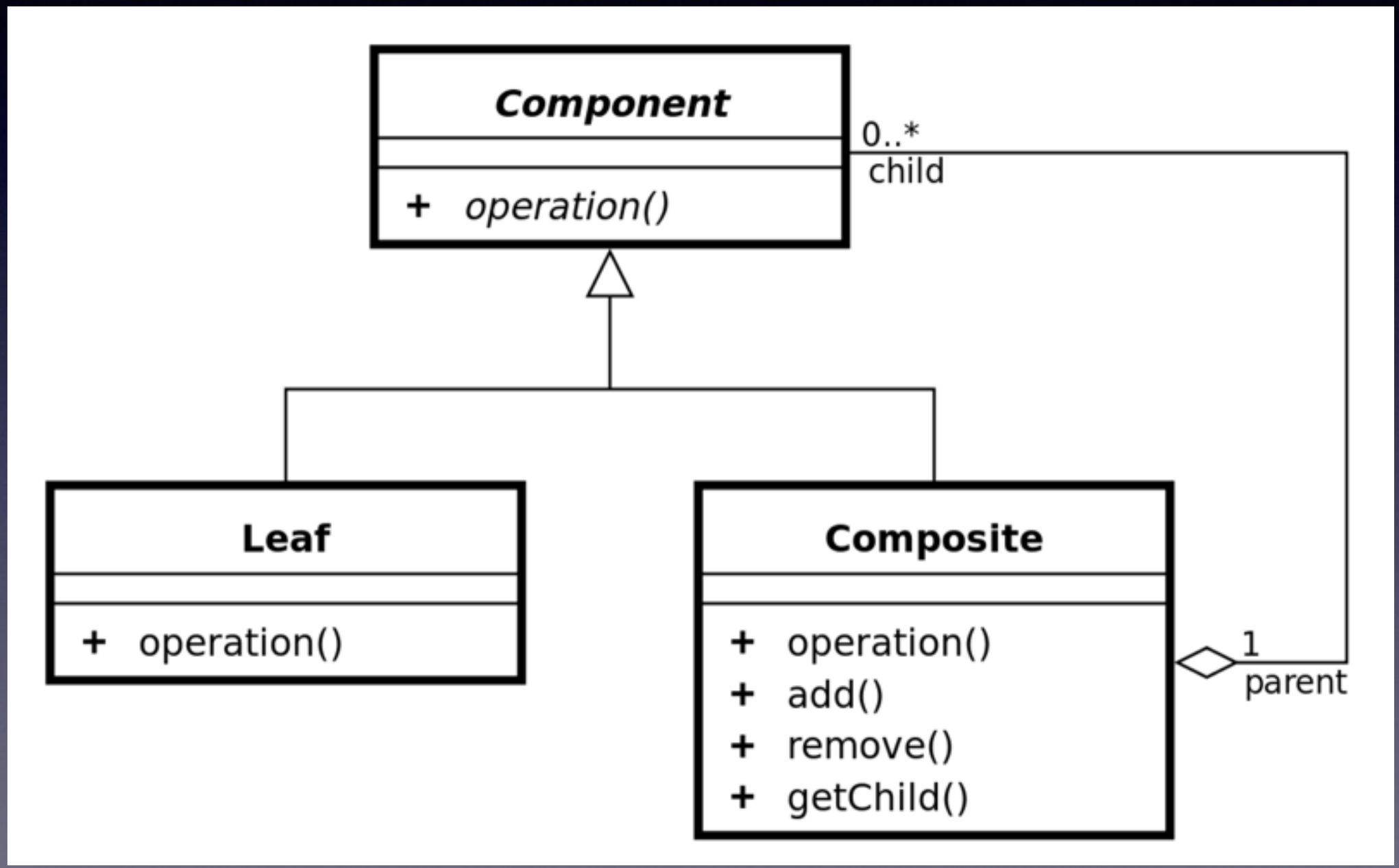
Bridge

```
/** "Client" */
class BridgePattern {
    public static void main(String[] args) {
        Shape[] shapes = new Shape[] {
            new CircleShape(1, 2, 3, new DrawingAPI1()),
            new CircleShape(5, 7, 11, new DrawingAPI2())
        };

        for (Shape shape : shapes) {
            shape.resizeByPercentage(2.5);
            shape.draw();
        }
    }
}
```

Bridge

使类的定义与实现之间解耦；
客户端与服务端单独编译；
能够组织复杂的类的层次机构；

Composite


Component
+ operation()
0..*
child
Leaf
+ operation()
Composite
+ operation()
+ add()
+ remove()
+ getChild()
1
parent

Composite

```java
/** "Component" */
interface Graphic {

    //Prints the graphic.
    public void print();
}

/** "Leaf" */
class Ellipse implements Graphic {

    //Prints the graphic.
    public void print() {
        System.out.println("Ellips
    }
}
```

```java
/** "Composite" */
import java.util.List;
import java.util.ArrayList;
class CompositeGraphic implements Graphic {

    //Collection of child graphics.
    private List<Graphic> childGraphics = new ArrayList<Gr

    //Prints the graphic.
    public void print() {
        for (Graphic graphic : childGraphics) {
            graphic.print();
        }
    }

    //Adds the graphic to the composition.
    public void add(Graphic graphic) {
        childGraphics.add(graphic);
    }

    //Removes the graphic from the composition.
    public void remove(Graphic graphic) {
        childGraphics.remove(graphic);
    }
}
```

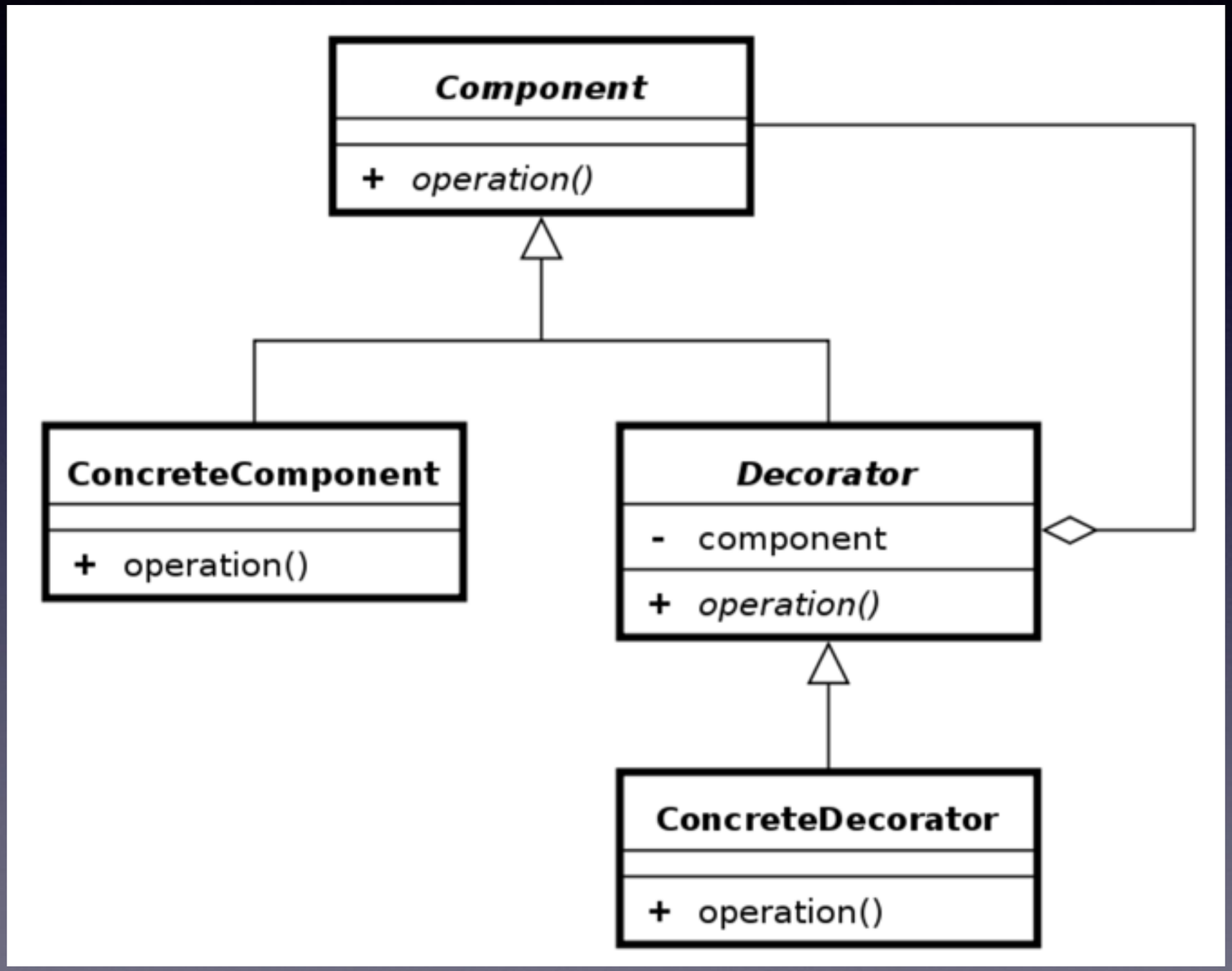
Composite

```
/** "Leaf" */
class Ellipse implements Graphic {

    //Prints the graphic.
    public void print() {
        System.out.println("Ellipse");
    }
}

/** Client */
public class Program {

    public static void main(String[] args) {
        //Initialize four ellipses
        Ellipse ellipse1 = new Ellipse();
        Ellipse ellipse2 = new Ellipse();
        Ellipse ellipse3 = new Ellipse();
        Ellipse ellipse4 = new Ellipse();

        //Initialize three composite graphics
        CompositeGraphic graphic = new CompositeGraphic();
        CompositeGraphic graphic1 = new CompositeGraphic();
        CompositeGraphic graphic2 = new CompositeGraphic();

        //Composes the graphics
        graphic1.add(ellipse1);
        graphic1.add(ellipse2);
        graphic1.add(ellipse3);

        graphic2.add(ellipse4);

        graphic.add(graphic1);
        graphic.add(graphic2);

        //Prints the complete graphic (four times the string "Ellipse").
        graphic.print();
    }
}
```

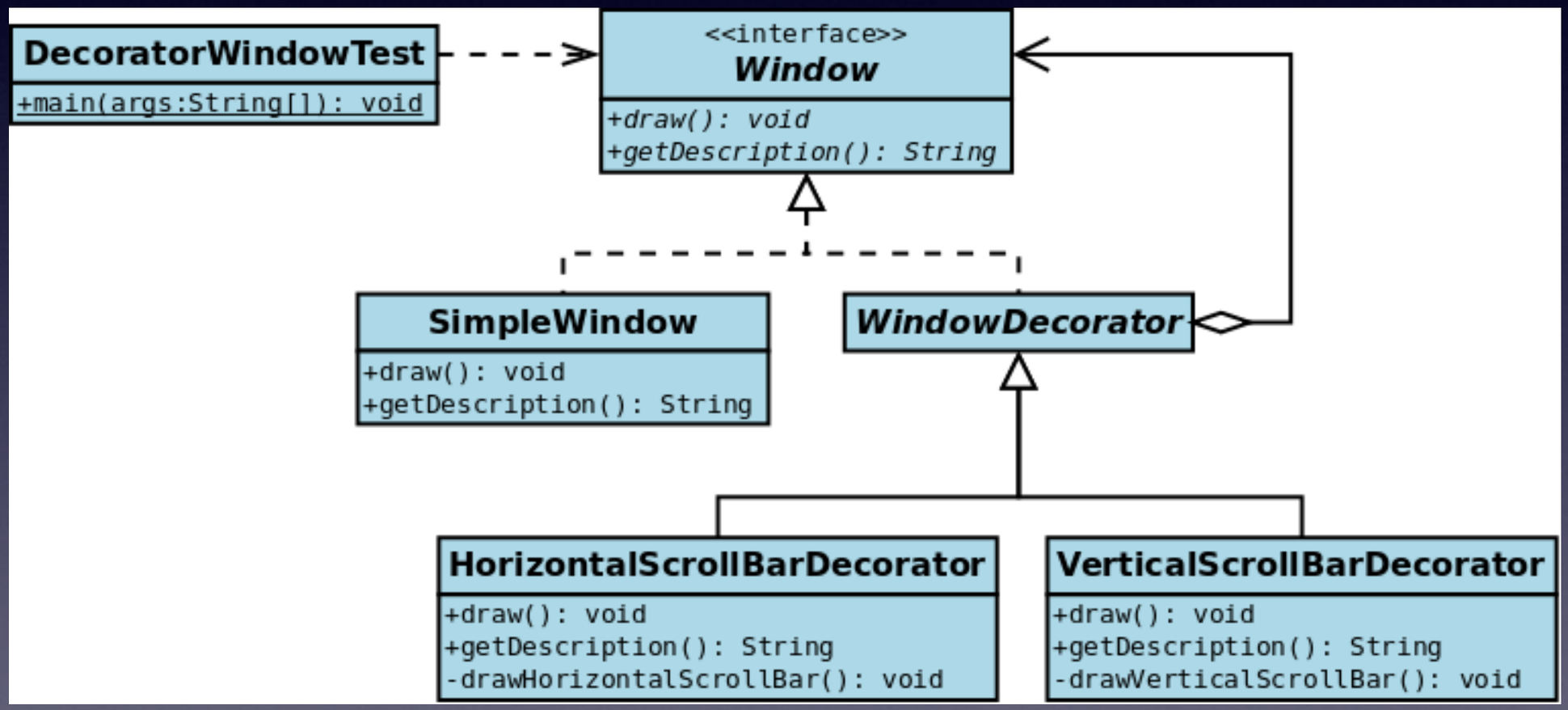
Composite

使一组对象与单个对象的暴露相同的接口，
则客户端可以忽略组合对象与单个对象的不同；
统一的使用组合中的所有对象

Decorator (or Wrapper)



Decorator (or Wrapper)



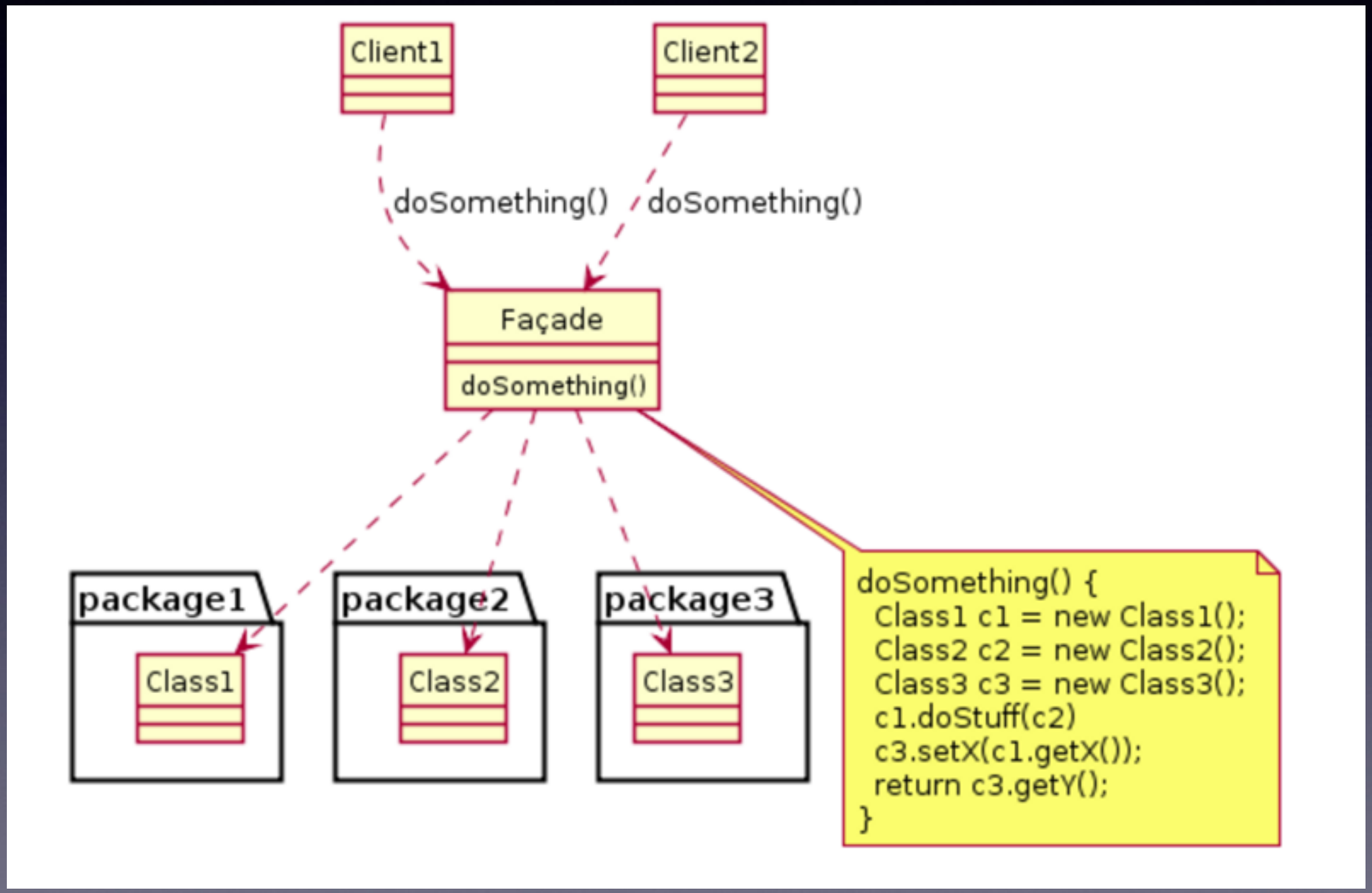
Decorator (or Wrapper)

能够方便的扩充原类的功能

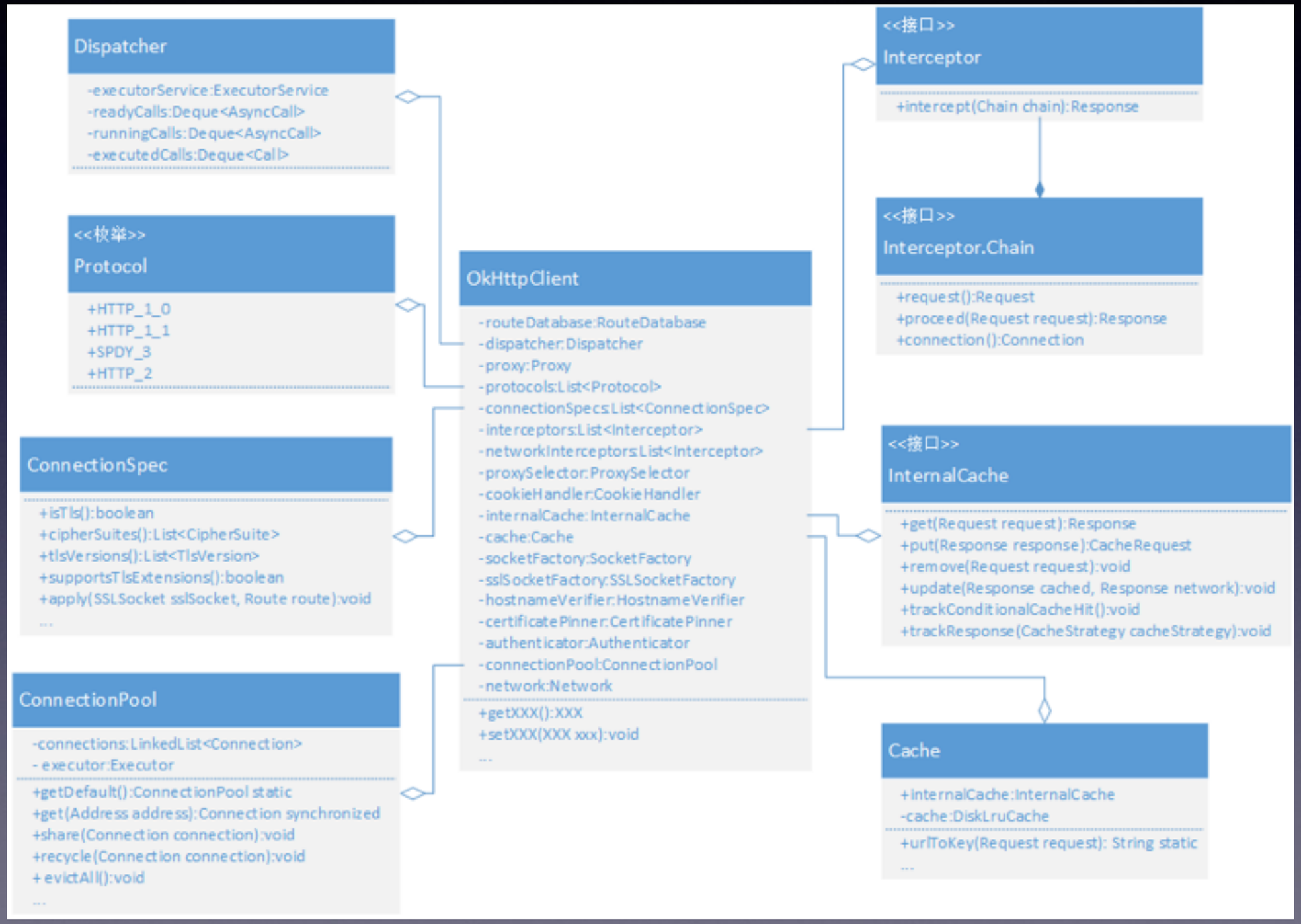
Decorator，Adapter

- 两者都使用组合，而不是继承
- Adapter 主要为了接口转换，Decorator 主要是为了添加新功能
- 通常 Adapter 会知道原类的详情，而 Decorator 并不一定清楚原类的情况

Facade


Client1
Client2
doSomething()
doSomething()
Façade
doSomething()
package1
Class1
package2
Class2
package3
Class3
doSomething() {
Class1 c1 = new Class1();
Class2 c2 = new Class2();
Class3 c3 = new Class3();
c1.doStuff(c2)
c3.setX(c1.getX());
return c3.getY();
}

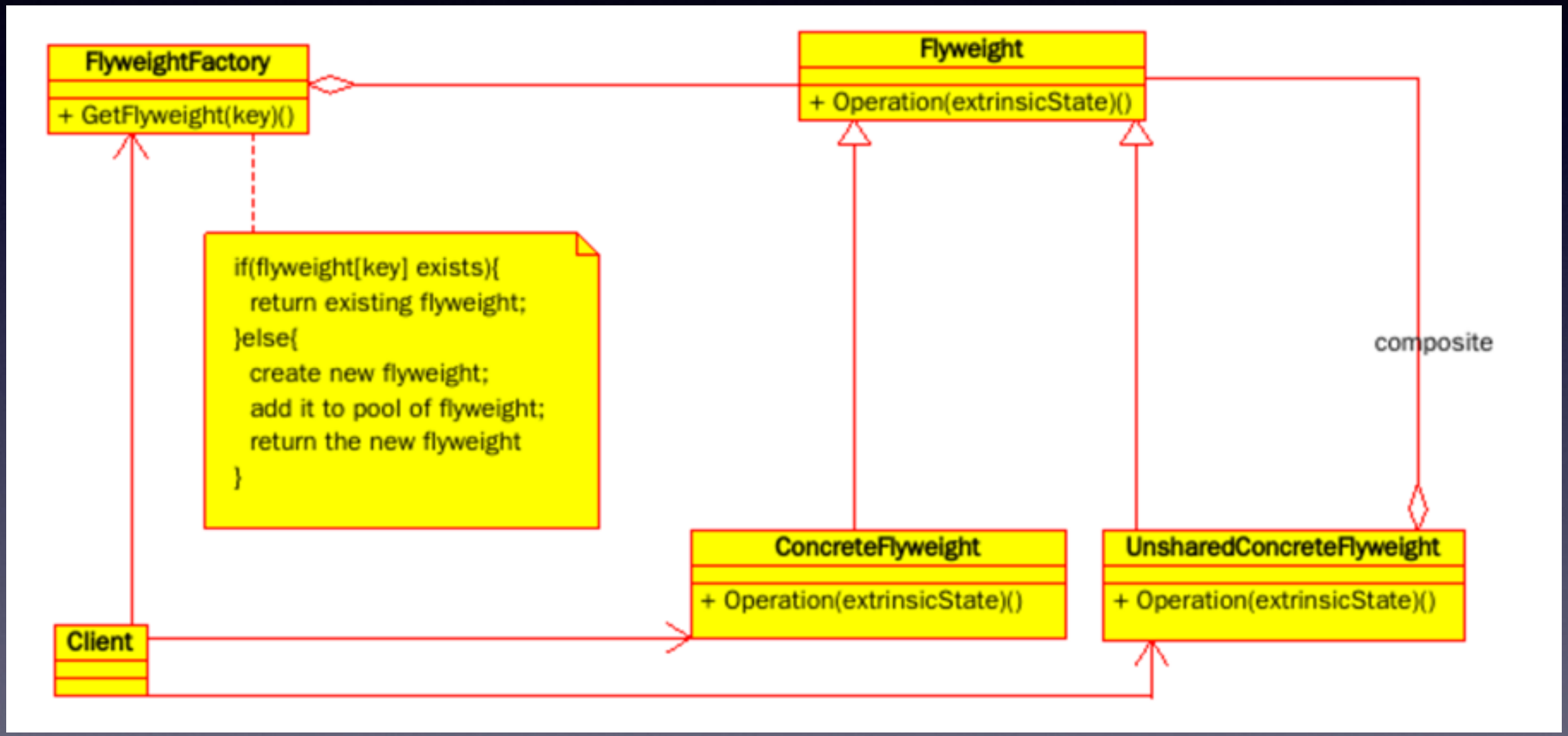
Facade


Dispatcher
-executorService:ExecutorService
-readyCalls:Deque<AsyncCall>
-runningCalls:Deque<AsyncCall>
-executedCalls:Deque<Call>
<<枚举>>
Protocol
+HTTP_1_0
+HTTP_1_1
+SPDY_3
+HTTP_2
OkHttpClient
-routeDatabase:RouteDatabase
-dispatcher:Dispatcher
-proxy:Proxy
-protocols:List<Protocol>
-connectionSpecs:List<ConnectionSpec>
-interceptors:List<Interceptor>
-networkInterceptors:List<Interceptor>
-proxySelector:ProxySelector
-cookieHandler:CookieHandler
-internalCache:InternalCache
-cache:Cache
-socketFactory:SocketFactory
-sslSocketFactory:SSLSocketFactory
-hostnameVerifier:HostnameVerifier
-certificatePinner:CertificatePinner
-authenticator:Authenticator
-connectionPool:ConnectionPool
-network:Network
+getXXX():XXX
+setXXX(XXX xxx):void
...
<<接口>>
Interceptor
+intercept(Chain chain):Response
<<接口>>
Interceptor.Chain
+request():Request
+proceed(Request request):Response
+connection():Connection
ConnectionSpec
+isTls():boolean
+cipherSuites():List<CipherSuite>
+tlsVersions():List<TlsVersion>
+supportsTlsExtensions():boolean
+apply(SSLSocket sslSocket, Route route):void
...
<<接口>>
InternalCache
+get(Request request):Response
+put(Response response):CacheRequest
+remove(Request request):void
+update(Response cached, Response network):void
+trackConditionalCacheHit():void
+trackResponse(CacheStrategy cacheStrategy):void
ConnectionPool
-connections:LinkedList<Connection>
- executor:Executor
+getDefault():ConnectionPool static
+get(Address address):Connection synchronized
+share(Connection connection):void
+recycle(Connection connection):void
+evictAll():void
...
Cache
+internalCache:InternalCache
-cache:DiskLruCache
+urlToKey(Request request): String static
...

Facade

为一个复杂的子系统提供一个简单统一的接口；
提高了子系统的独立性和可移植性；
简化了子系统之间的依赖

Flyweight


FlyweightFactory
+ GetFlyweight(key)()
Flyweight
+ Operation(extrinsicState)()
if(flyweight[key] exists){
return existing flyweight;
}else{
create new flyweight;
add it to pool of flyweight;
return the new flyweight
}
composite
ConcreteFlyweight
+ Operation(extrinsicState)()
UnsharedConcreteFlyweight
+ Operation(extrinsicState)()
Client

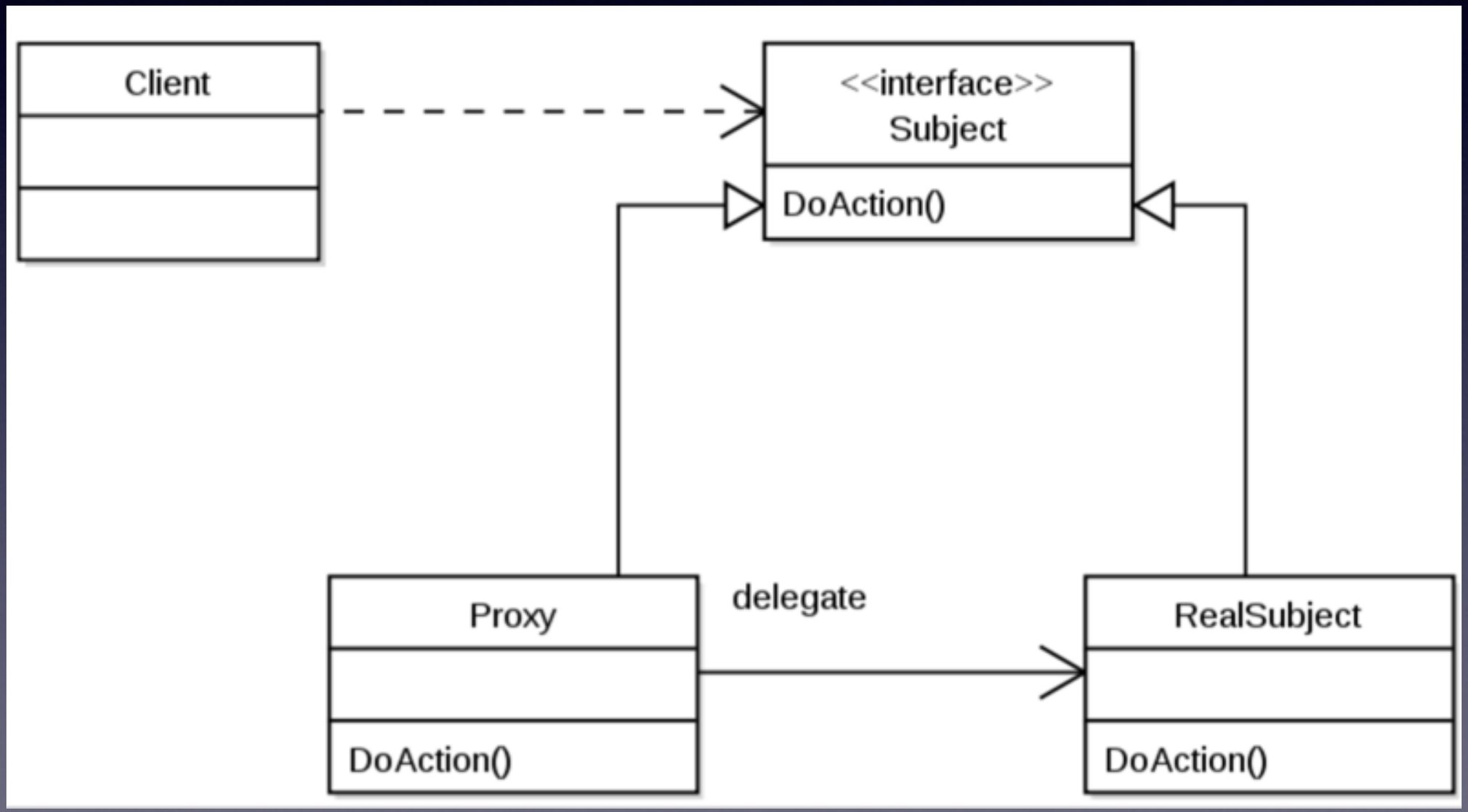
Flyweight

```
public class FlyweightFactory{
    private HashMap flyweights = new HashMap();

    public Flyweight getFlyweight(String key){
        if(flyweights.containsKey(key)){
            return (Flyweight)flyweights.get(key);
        }
        else{
            Flyweight fw = new ConcreteFlyweight();
            flyweights.put(key,fw);
            return fw;
        }
    }
}
```

Flyweight

适用于程序中使用了大量相同或本质上相关的对象，
共享这些对象能够降低内存开销

Proxy


Client
<<interface>>
Subject
DoAction()
Proxy
delegate
RealSubject
DoAction()
DoAction()

Proxy

```
interface Image {
    public void displayImage();
}

//on System A
class RealImage implements Image {

    private String filename = null;
    /**
     * Constructor
     * @param filename
     */
    public RealImage(final String filename) {
        this.filename = filename;
        loadImageFromDisk();
    }

    /**
     * Loads the image from the disk
     */
    private void loadImageFromDisk() {
        System.out.println("Loading   " + filename);
    }

    /**
     * Displays the image
     */
    public void displayImage() {
        System.out.println("Displaying " + filename);
    }

}
```

```
//on System B
class ProxyImage implements Image {

    private RealImage image = null;
    private String filename = null;
    /**
     * Constructor
     * @param filename
     */
    public ProxyImage(final String filename) {
        this.filename = filename;
    }

    /**
     * Displays the image
     */
    public void displayImage() {
        if (image == null) {
            image = new RealImage(filename);
        }
        image.displayImage();
    }

}
```

Proxy

```
class ProxyExample {

    /**
     * Test method
     */
    public static void main(String[] args) {
        final Image IMAGE1 = new ProxyImage("HiRes_10MB_Photo1");
        final Image IMAGE2 = new ProxyImage("HiRes_10MB_Photo2");

        IMAGE1.displayImage(); // loading necessary
        IMAGE1.displayImage(); // loading unnecessary
        IMAGE2.displayImage(); // loading necessary
        IMAGE2.displayImage(); // loading unnecessary
        IMAGE1.displayImage(); // loading unnecessary
    }

}
```

Proxy

代理模式为客户端程序提供了一个统一的使用接口，
将例如网络，缓存，文件等对象的操作封装为不可感知的；

Proxy，Decorator

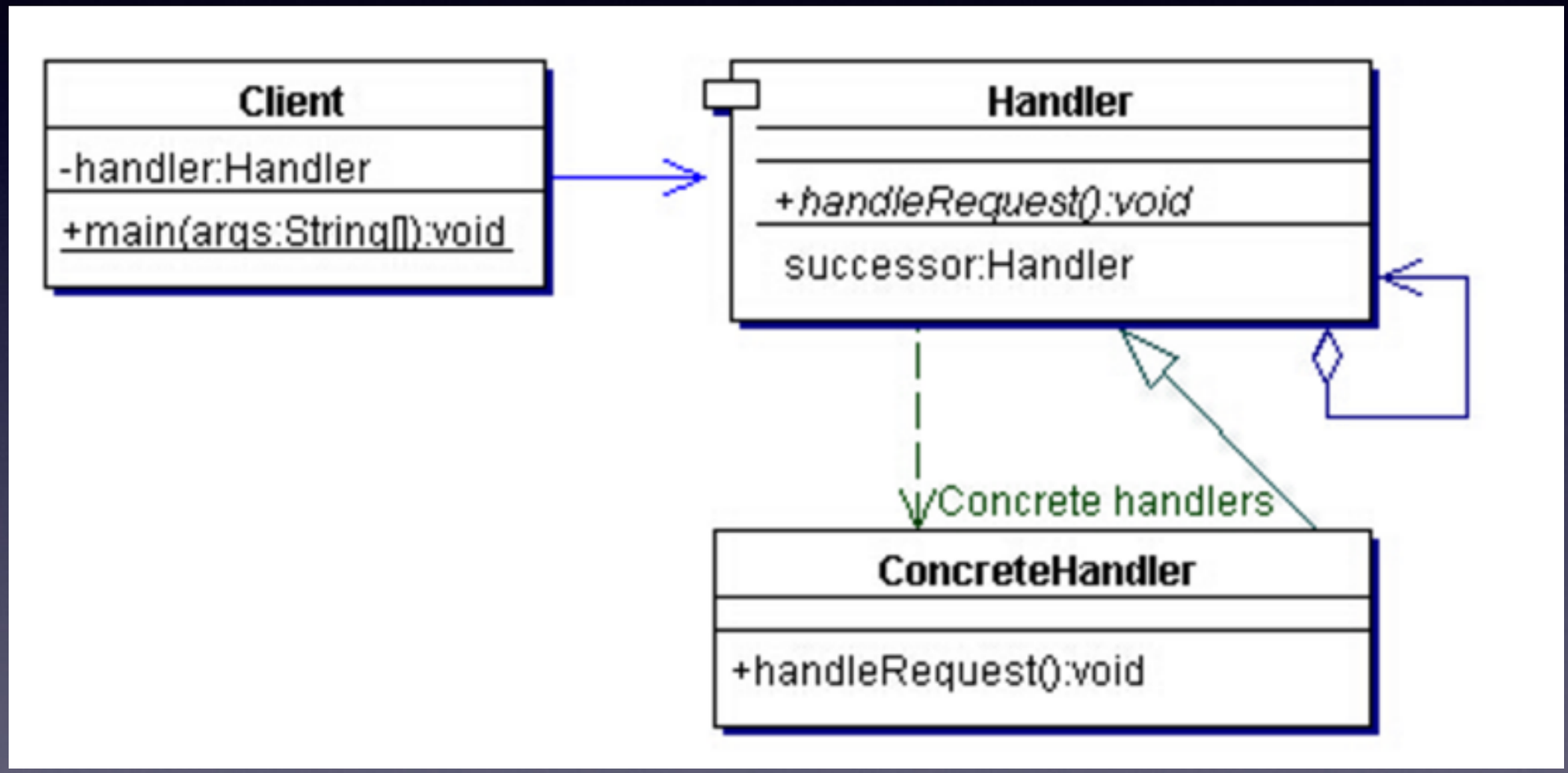
- Proxy 更关注对原类的访问控制，Decorator 更关注对原类进行扩充
- Proxy 通常会在内部创建原类，而 Decorator 通常将原类作为构造函数参数从外部传入
- Proxy 更适用于网络，文件等高延迟情景

行为型设计模式

行为型设计模式

- Chain of Responsibility 责任链
- Command 命令
- Interpreter 解释器
- Iterator 迭代器
- Mediator 中介者
- Memento 备忘录
- Observer 观察者
- State 状态
- Strategy 策略
- Template Method 模仿方法
- Visitor 访问者

Chain of Responsibility



Chain of Responsibility

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

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

    public void setSuccessor(PurchasePower 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);
        }
    }
}
```

Chain of Responsibility

```
class ManagerPPower extends PurchasePower {

    protected double getAllowable(){
        return BASE*10;
    }

    protected String getRole(){
        return "Manager";
    }
}

class DirectorPPower extends PurchasePower {

    protected double getAllowable(){
        return BASE*20;
    }

    protected String getRole(){
        return "Director";
    }
}
```

```
class VicePresidentPPower extends PurchasePower {

    protected double getAllowable(){
        return BASE*40;
    }

    protected String getRole(){
        return "Vice President";
    }
}

class PresidentPPower extends PurchasePower {

    protected double getAllowable(){
        return BASE*60;
    }

    protected String getRole(){
        return "President";
    }
}
```

Chain of Responsibility

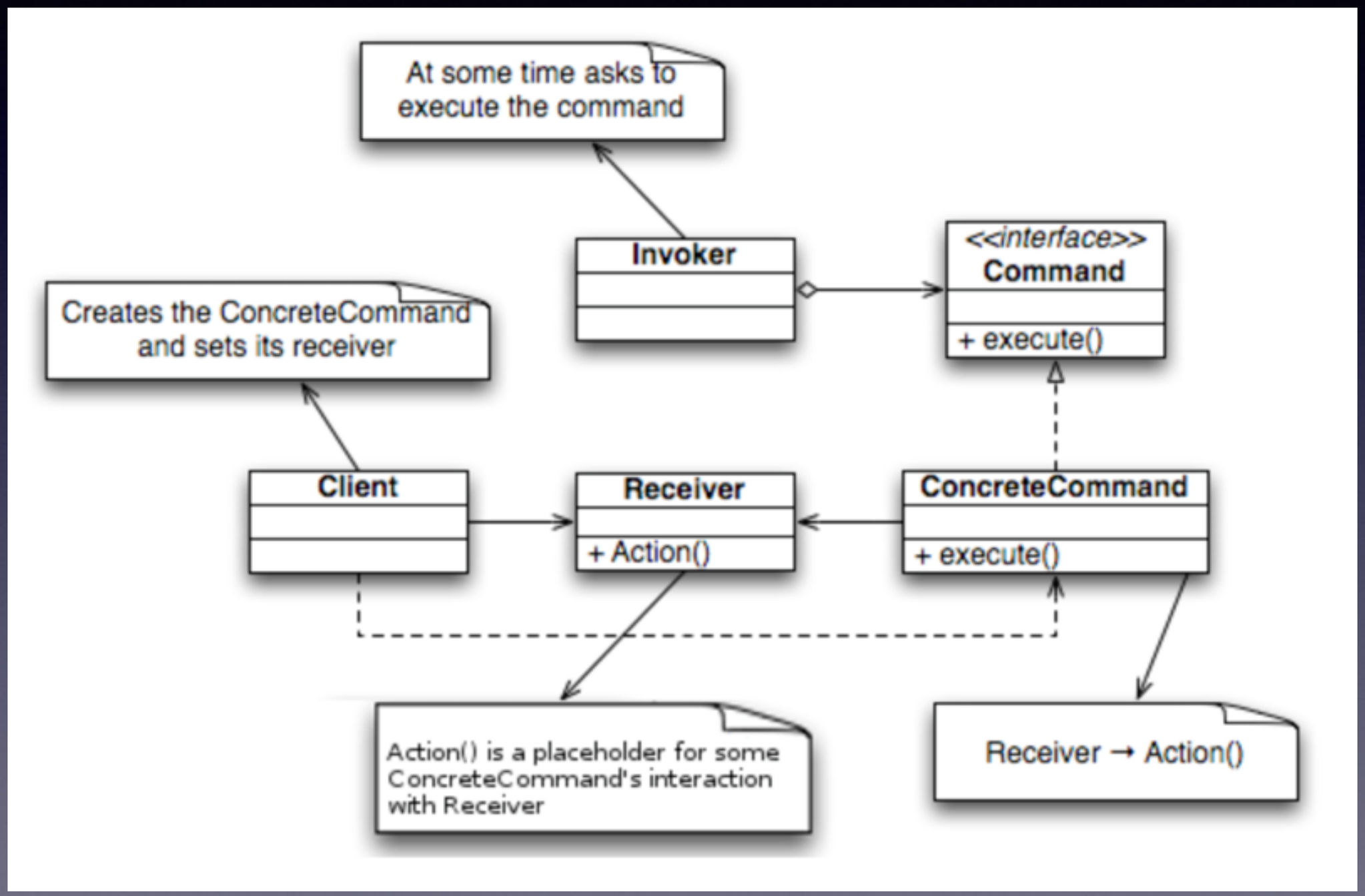
```
class CheckAuthority {
    public static void main(String[] args) {
        ManagerPPower manager = new ManagerPPower();
        DirectorPPower director = new DirectorPPower();
        VicePresidentPPower vp = new VicePresidentPPower();
        PresidentPPower president = new PresidentPPower();
        manager.setSuccessor(director);
        director.setSuccessor(vp);
        vp.setSuccessor(president);

        // Press Ctrl+C to end.
        try {
            while (true) {
                System.out.println("Enter the amount to check who should approve your expenditure.");
                System.out.print(">");
                double d = Double.parseDouble(new BufferedReader(new InputStreamReader(System.in)).readLine());
                manager.processRequest(new PurchaseRequest(d, "General"));
            }
        } catch(Exception e) {
            System.exit(1);
        }
    }
}
```

Chain of Responsibility

有多个对象可以处理某个请求，这些对象处理请求的条件间满足某种关系，将这个关系固化到对象中就是责任链模型；

这样客户端只需要掌握一个处理句柄即可

Command


At some time asks to execute the command
Invoker
<<interface>>
Command
+ execute()
Creates the ConcreteCommand and sets its receiver
Client
Receiver
+ Action()
ConcreteCommand
+ execute()
Action() is a placeholder for some ConcreteCommand's interaction with Receiver
Receiver → Action()

Command

```java
package com.command;
/**
 * 命令接口    [命令角色]
 */
public interface Command {
    public void execute();
    public void undo();
    public void redo();
}
```

```java
public class ConcreteCommandImpl1 implements Command{
    private ReceiverRole receiverRole1;

    public ConcreteCommandImpl1(ReceiverRole receiverRole1) {
        this.receiverRole1 = receiverRole1;
    }

    @Override
    public void execute() {
        /*
         * 可以加入命令排队等等，未执行的命令支持redo操作
         */
        receiverRole1.opActionUpdateAge(1001);//执行具体的命令操作
    }

    @Override
    public void undo() {
        receiverRole1.rollBackAge();//执行具体的撤销回滚操作
    }

    @Override
    public void redo() {
        //在命令执行前可以修改命令的执行
    }
}
```

Command

```
public class ConcreteCommandImpl2 implements Command{
    private ReceiverRole receiverRole1;

    public ConcreteCommandImpl2(ReceiverRole receiverRole1) {
        this.receiverRole1 = receiverRole1;
    }

    @Override
    public void execute() {
        /*
         * 可以加入命令排队等等，未执行的命令支持redo操作
         */
        receiverRole1.opActionUpdateName("lijunhuayc");//执行具体
    }

    @Override
    public void undo() {
        receiverRole1.rollBackName();//执行具体的撤销回滚操作
    }

    @Override
    public void redo() {
        //在命令执行前可以修改命令的执行
    }

}
```

```
public class InvokerRole {
    private Command command1;
    private Command command2;
    //持有多个命令对象[实际的情况也可能是一个命令对象的集合来保存命

    public void setCommand1(Command command1) {
        this.command1 = command1;
    }
    public void setCommand2(Command command2) {
        this.command2 = command2;
    }

    /**
     * 执行正常命令，1执行回滚命令
     */
    public void invoke(int args) {
        //可以根据具体情况选择执行某些命令
        if(args == 0){
            command1.execute();
            command2.execute();
        }else if(args == 1){
            command1.undo();
            command2.undo();
        }
    }

}
```

Command

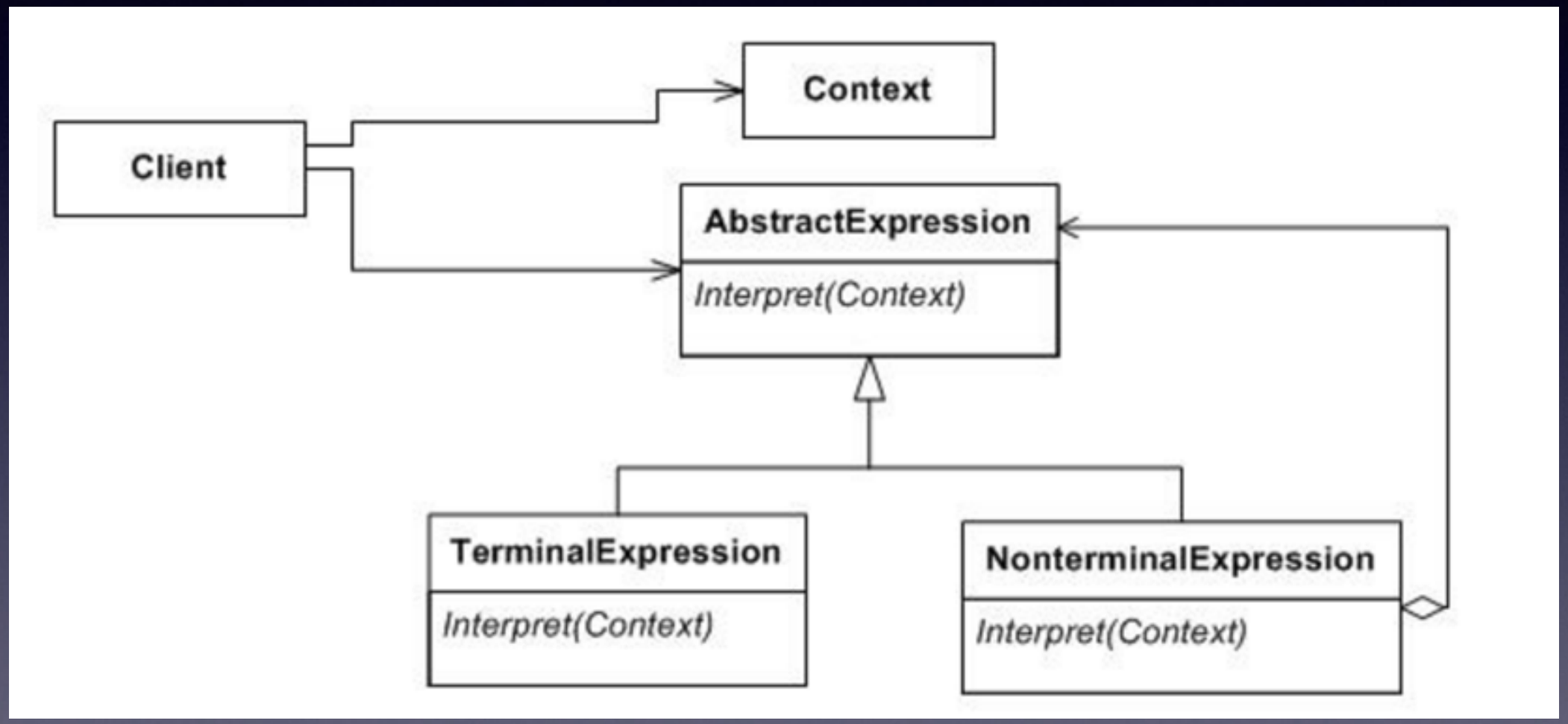
```java
public class ReceiverRole {
    private PeopleBean people;
    //具体命令操作的缓存栈，用于回滚。这里为了方便就用一个PeopleBean来代替
    private PeopleBean peopleCache = new PeopleBean();
    public ReceiverRole() {
        this.people = new PeopleBean(-1, "NULL");//初始化年龄为-1，姓名为NULL
    }

    public ReceiverRole(PeopleBean people) {
        this.people = people;
    }

/**
 * 具体操作方法[修改年龄和姓名]
 */
public void opActionUpdateAge(int age) {
    System.out.println("执行命令前: "+people.toString());
    this.people.update(age);
    System.out.println("执行命令后: "+people.toString()+"\n");
}

//修改姓名
public void opActionUpdateName(String name) {
    System.out.println("执行命令前: "+people.toString());
    this.people.update(name);
    System.out.println("执行命令后: "+people.toString()+"\n");
}
```

```java
public class ClientRole {
    /**
     * 组装操作
     */
    public void assembleAction() {
        //创建一个命令接收者
        ReceiverRole receiverRole1 = new ReceiverRole();
        Command command1 = new ConcreteCommandImpl1(receiverRole1

        InvokerRole invokerRole = new InvokerRole();//创建一个命令调
        invokerRole.setCommand1(command1);//为调用者指定命令对象1
        invokerRole.setCommand2(command2);//为调用者指定命令对象2
        invokerRole.invoke(0);                //发起调用命令请求
        invokerRole.invoke(1);                //发起调用命令请求
    }
}
```

Command

解耦了命令发送者与接收者；
新命令容易加入到系统中；
可以轻松的实现命令组合，同一命令能够与不同效果；
但是命令模式可能造成命令类较多

Interpreter



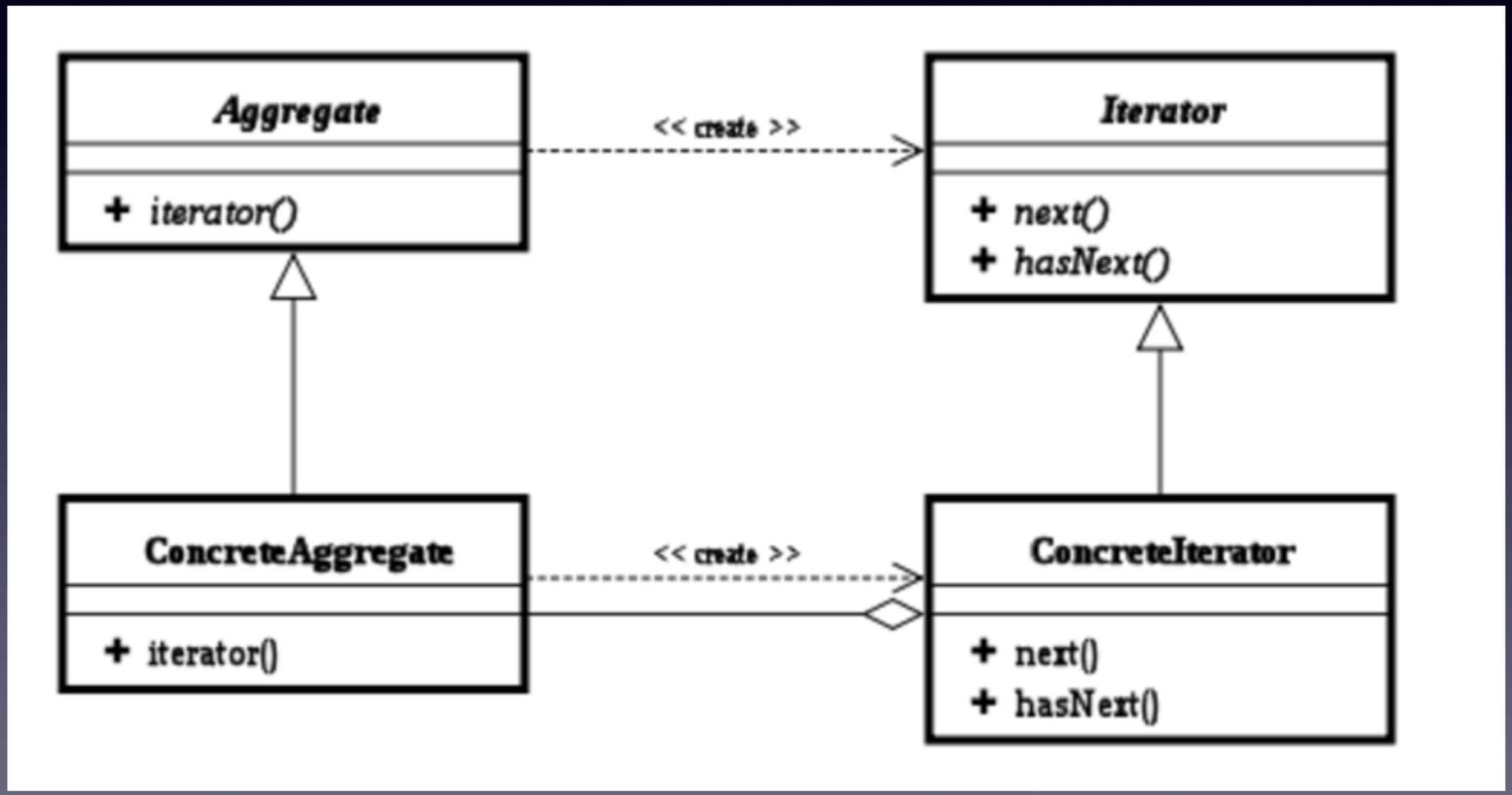
Interpreter

https://en.wikipedia.org/wiki/Interpreter_pattern#Java

Interpreter

解释器模式通常性能一般，
适用于性能不敏感的语言需要解释执行的场景

Iterator


Aggregate
+ iterator()
<< create >>
Iterator
+ next()
+ hasNext()
ConcreteAggregate
+ iterator()
<< create >>
ConcreteIterator
+ next()
+ hasNext()

Iterator

```
function reverseArrayIterator(array) {
    var index = array.length - 1;
    return {
        next: () =>
            index >= 0 ?
             {value: array[index--], done: false} :
             {done: true}
    }
}

const it = reverseArrayIterator(['three', 'two', 'one']);
console.log(it.next().value);  //-> 'one'
console.log(it.next().value);  //-> 'two'
console.log(it.next().value);  //-> 'three'
console.log(`Are you done? ${it.next().done}`);  //-> true
```

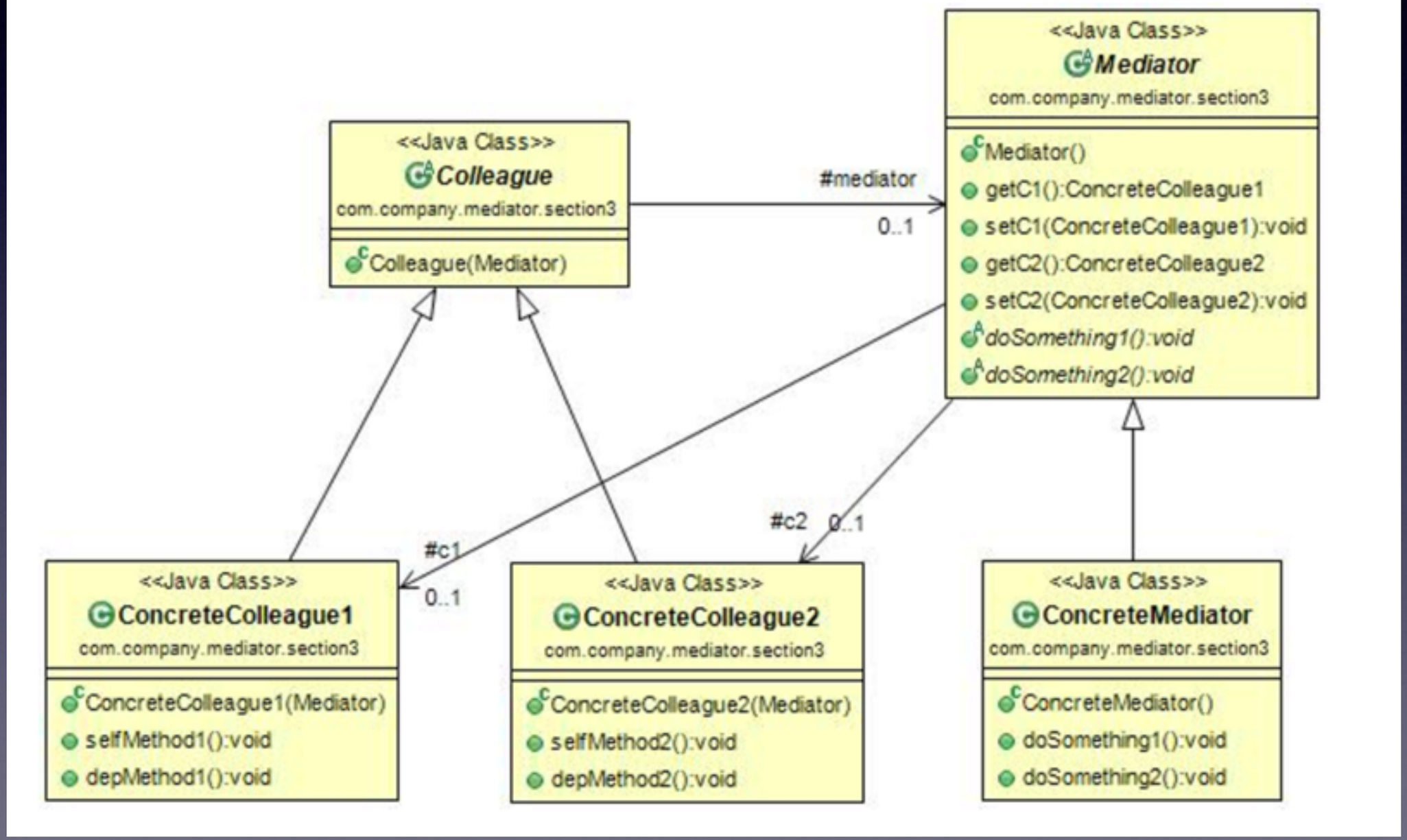
Iterator

Iterator模式就是分离了集合对象的遍历行为，抽象出一个迭代器类来负责，这样不暴露集合的内部结构，又可让外部代码透明的访问集合内部的数据；

消耗了更多内存，遍历是不可逆的，遍历过程中如果集合内容改变则会产生错误 ConcurrentModificationException

Mediator

中介者模式的通用类图如下图：



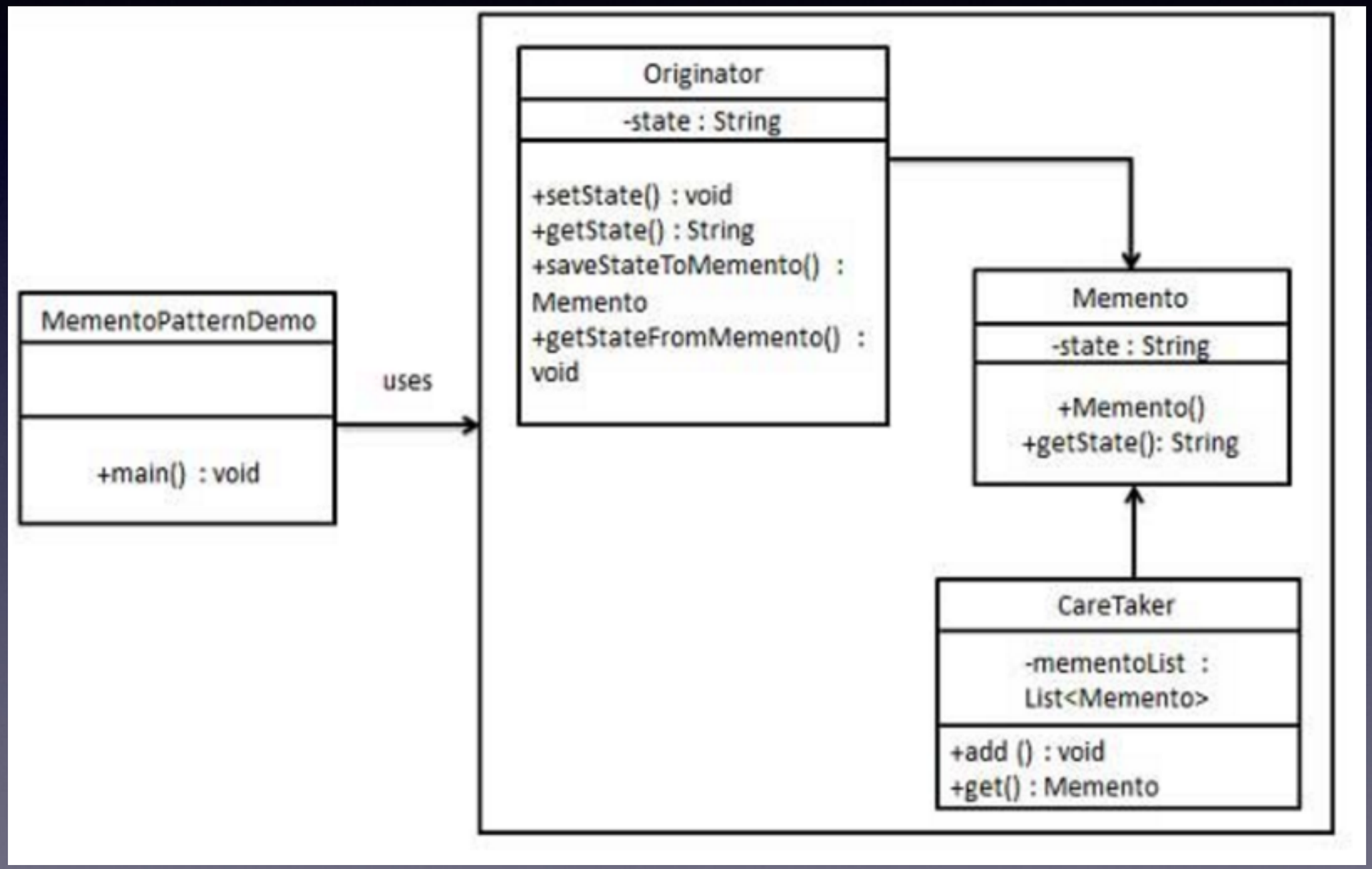
Mediator

https://en.wikipedia.org/wiki/Mediator_pattern#Java

Mediator

简化了个对象之间的通信联系，减少同事之间的耦合

Memento


Originator
-state : String
+setState() : void
+getState() : String
+saveStateToMemento() : Memento
+getStateFromMemento() : void
MementoPatternDemo
+main() : void
uses
Memento
-state : String
+Memento()
+getState(): String
CareTaker
-mementoList : List<Memento>
+add () : void
+get() : Memento

Memento

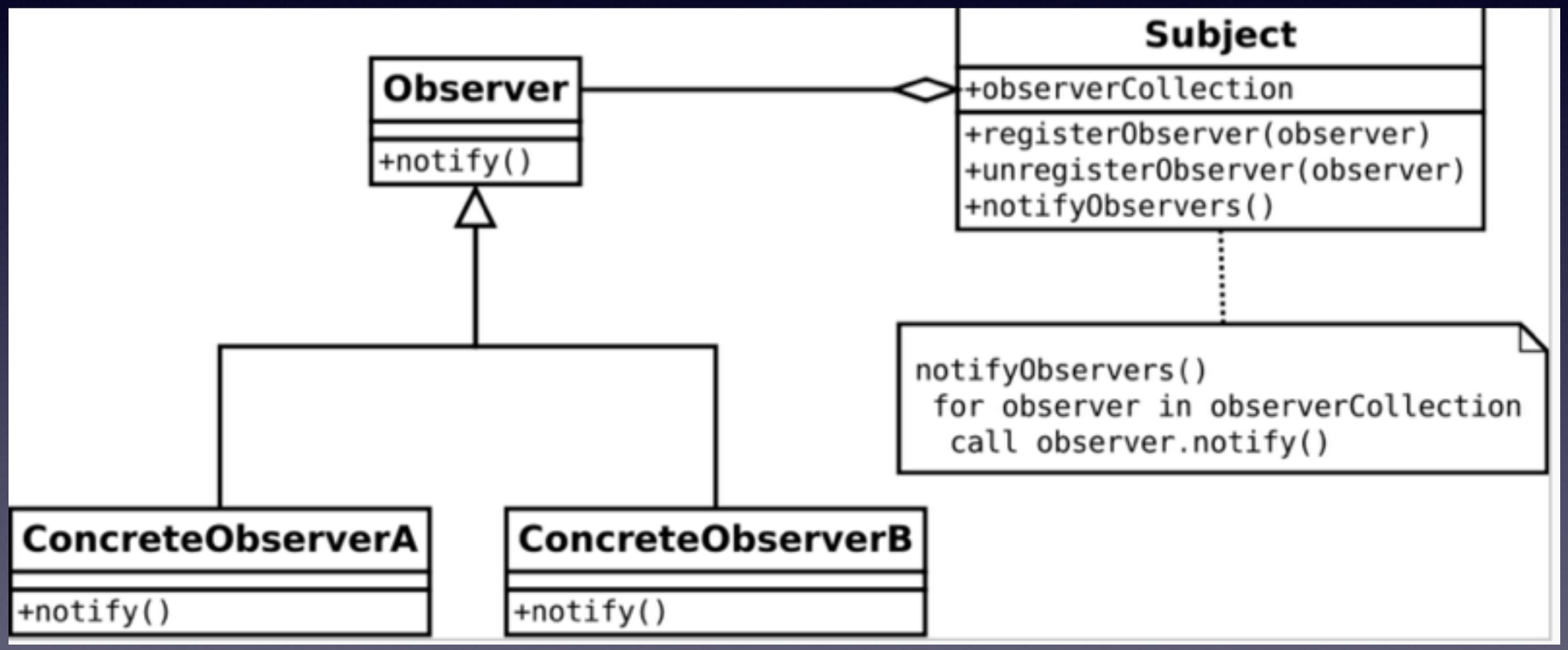
```
class Originator {
    private String state;
    // The class could also contain additional data that
    // state saved in the memento..

    public void set(String state) {
        System.out.println("Originator: Setting state to "
        this.state = state;
    }

    public Memento saveToMemento() {
        System.out.println("Originator: Savir
        return new Memento(this.state);
    }

    public void restoreFromMemento(Memento me
        this.state = memento.getSavedState();
        System.out.println("Originator: State
    }

    public static class Memento {
        private final String state;

        public Memento(String stateToSave) {
            state = stateToSave;
        }

        private String getSavedState() {
            return state;
        }
    }
}
```

```
class Caretaker {
    public static void main(String[] args) {
        List<Originator.Memento> savedStates = new ArrayList<O

        Originator originator = new Originator();
        originator.set("State1");
        originator.set("State2");
        savedStates.add(originator.saveToMemento());
        originator.set("State3");
        // We can request multiple mementos, and choose which
        savedStates.add(originator.saveToMemento());
        originator.set("State4");

        originator.restoreFromMemento(savedStates.get(1));
    }
}
```

Memento

在不破坏对象的封装性的情况下能够保存对象的状态

Observer


Subject
+observerCollection
+registerObserver(observer)
+unregisterObserver(observer)
+notifyObservers()
Observer
+notify()
notifyObservers()
for observer in observerCollection
call observer.notify()
ConcreteObserverA
+notify()
ConcreteObserverB
+notify()

Observer

```
class EventSource extends Observable implements Runnable {
    public void run() {
        while (true) {
            String response = new Scanner(System.in).next();
            setChanged();
            notifyObservers(response);
        }
    }
}
```

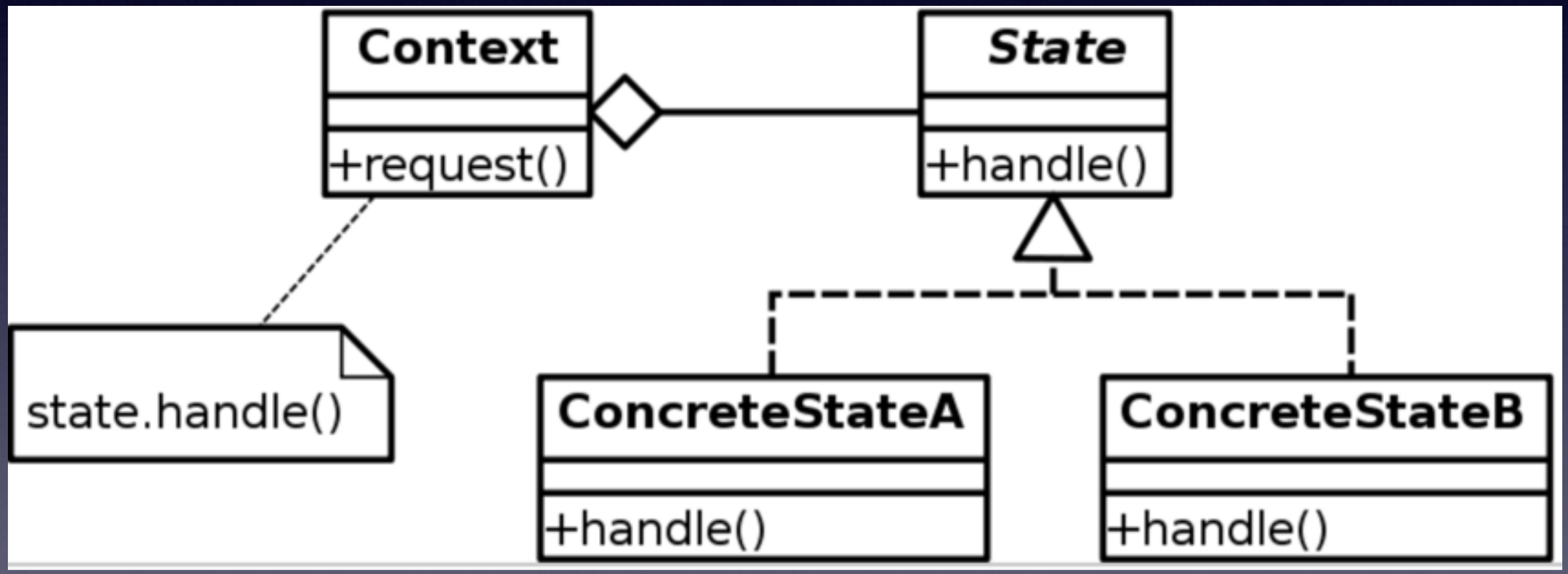
```
public static void main(String[] args) {
    System.out.println("Enter Text: ");
    EventSource eventSource = new EventSource();

    eventSource.addObserver(new Observer() {
        public void update(Observable obj, Object arg) {
            System.out.println("Received response: " + arg);
        }
    });

    new Thread(eventSource).start();
}
```

Observer

使消息的发送与接收解耦，即使响应与发送存在依赖；

发送方不必了解有多少接收放以及他们是谁

State


Context
+request()
State
+handle()
state.handle()
ConcreteStateA
+handle()
ConcreteStateB
+handle()

State

```
interface Statelike {

    void writeName(StateContext context, String name);

}

class StateLowerCase implements Statelike {

    @Override
    public void writeName(final StateContext context, final
        System.out.println(name.toLowerCase());
        context.setState(new StateMultipleUpperCase());
    }

}

class StateMultipleUpperCase implements Statelike {
    /** Counter local to this state */
    private int count = 0;

    @Override
    public void writeName(final StateContext context, final
        System.out.println(name.toUpperCase());
        /* Change state after StateMultipleUpperCase's write
        if(++count > 1) {
            context.setState(new StateLowerCase());
        }
    }
}
```

```
class StateContext {
    private Statelike myState;
    StateContext() {
        setState(new StateLowerCase());
    }

    /**
     * Setter method for the state.
     * Normally only called by classes imple
     * @param newState the new state of this
     */
    void setState(final Statelike newState)
        myState = newState;
    }

    public void writeName(final String name)
        myState.writeName(this, name);
    }
```

State

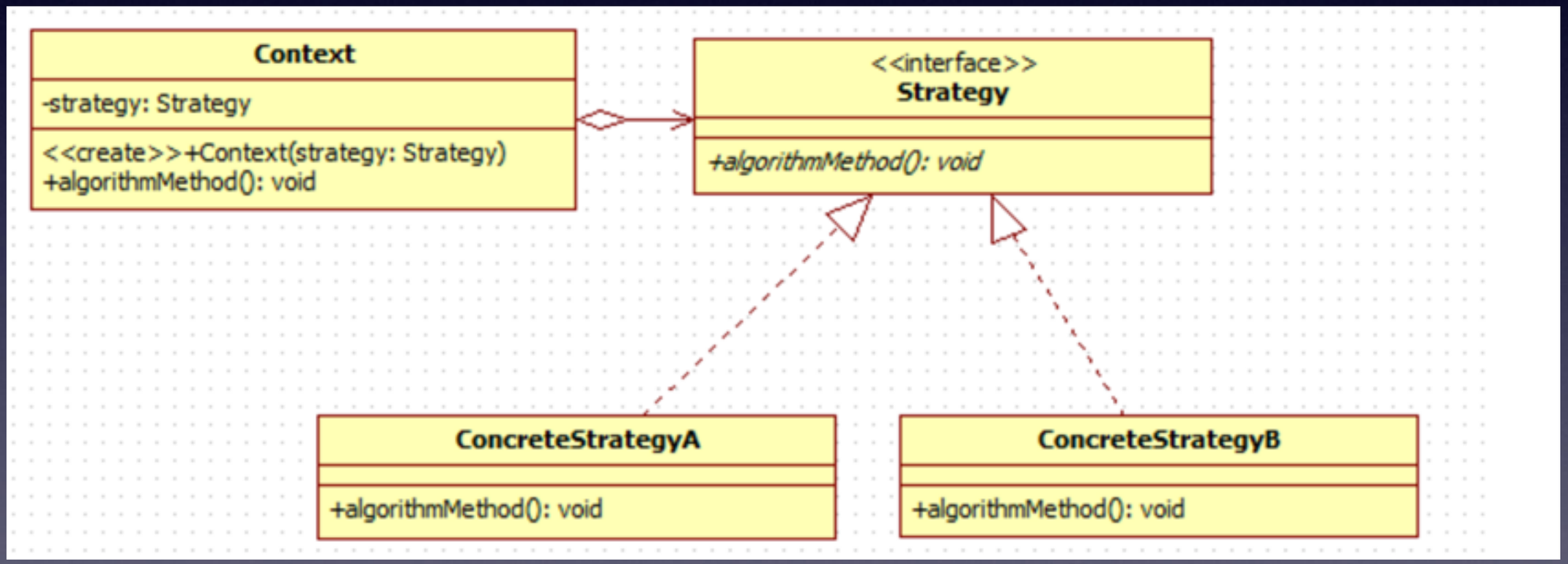
```
public class DemoOfClientState {
    public static void main(String[] args) {
        final StateContext sc = new StateContext();

        sc.writeName("Monday");
        sc.writeName("Tuesday");
        sc.writeName("Wednesday");
        sc.writeName("Thursday");
        sc.writeName("Friday");
        sc.writeName("Saturday");
        sc.writeName("Sunday");
    }
}
```

```
monday
TUESDAY
WEDNESDAY
thursday
FRIDAY
SATURDAY
sunday
```

State

在含有大量分之语句，且分支有状态有关的情景下，
通常可以用一个或多个枚举量或类来表示，
将对象的状态用这个变来关联到相应的 Action，
对象的状态便作为一个对象独立存在不依赖于其他对象

Strategy

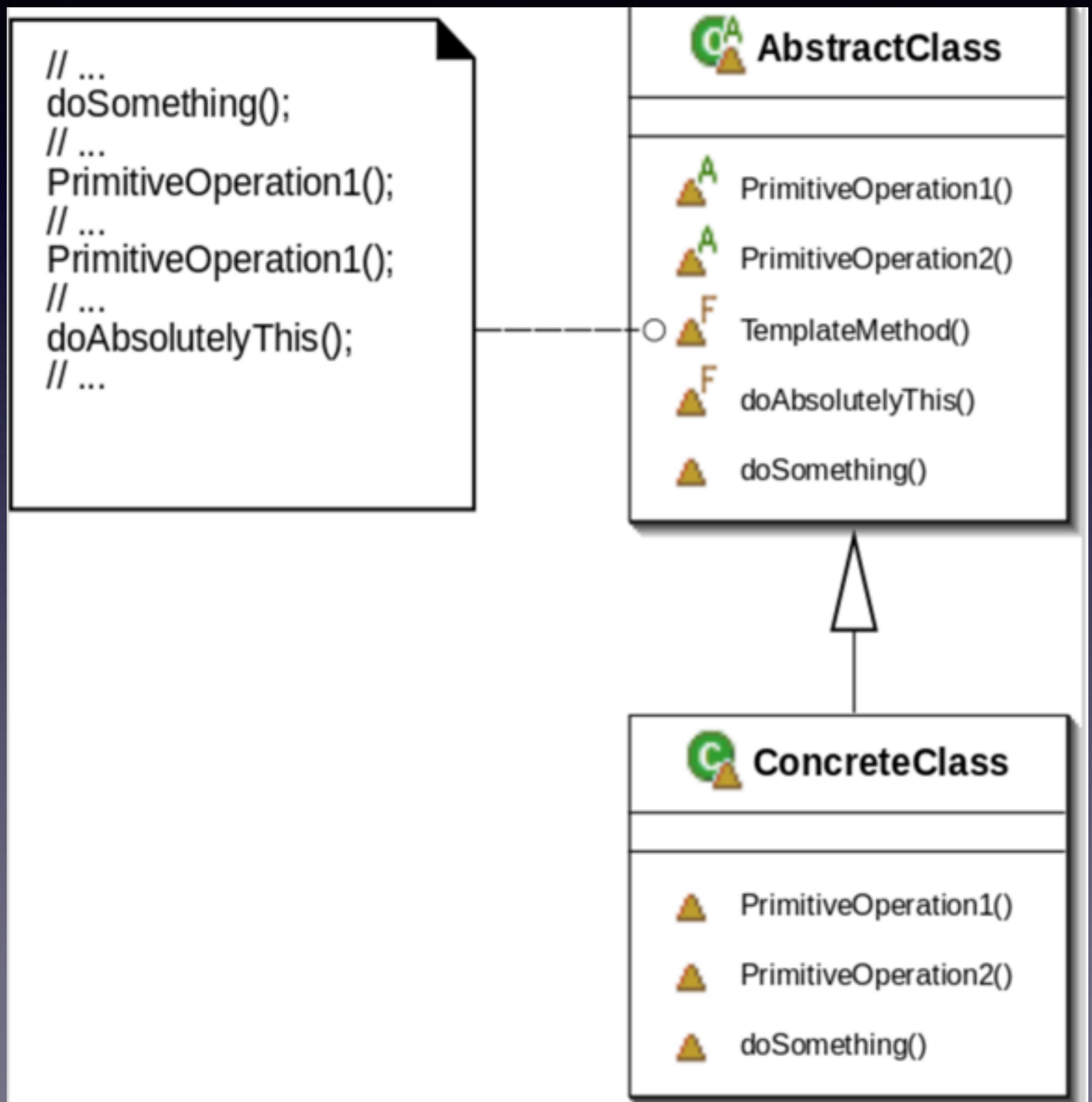

Context
-strategy: Strategy
<<create>>+Context(strategy: Strategy)
+algorithmMethod(): void
<<interface>>
Strategy
+algorithmMethod(): void
ConcreteStrategyA
+algorithmMethod(): void
ConcreteStrategyB
+algorithmMethod(): void

Strategy

```php
interface TravelStrategy{
    public function travelAlgorithm();
}


/**
 * 具体策略类(ConcreteStrategy)1：乘坐飞机
 */
class AirPlanelStrategy implements TravelStrategy
    public function travelAlgorithm(){
        echo "travel by AirPlain", "<BR>\r\n";
    }
}


/**
 * 具体策略类(ConcreteStrategy)2：乘坐火车
 */
class TrainStrategy implements TravelStrategy {
    public function travelAlgorithm(){
        echo "travel by Train", "<BR>\r\n";
    }
}

/**
 * 具体策略类(ConcreteStrategy)3：骑自行车
 */
class BicycleStrategy implements TravelStrategy {
    public function travelAlgorithm(){
        echo "travel by Bicycle", "<BR>\r\n";
    }
}
```

```php
class PersonContext{
    private $_strategy = null;

    public function __construct(TravelStrategy $travel){
        $this->_strategy = $travel;
    }
    /**
    * 旅行
    */
    public function setTravelStrategy(TravelStrategy $tra
        $this->_strategy = $travel;
    }
    /**
    * 旅行
    */
    public function travel(){
        return $this->_strategy ->travelAlgorithm();
    }
}

// 乘坐火车旅行
$person = new PersonContext(new TrainStrategy());
$person->travel();

// 改骑自行车
$person->setTravelStrategy(new BicycleStrategy());
$person->travel();
```

Template Method

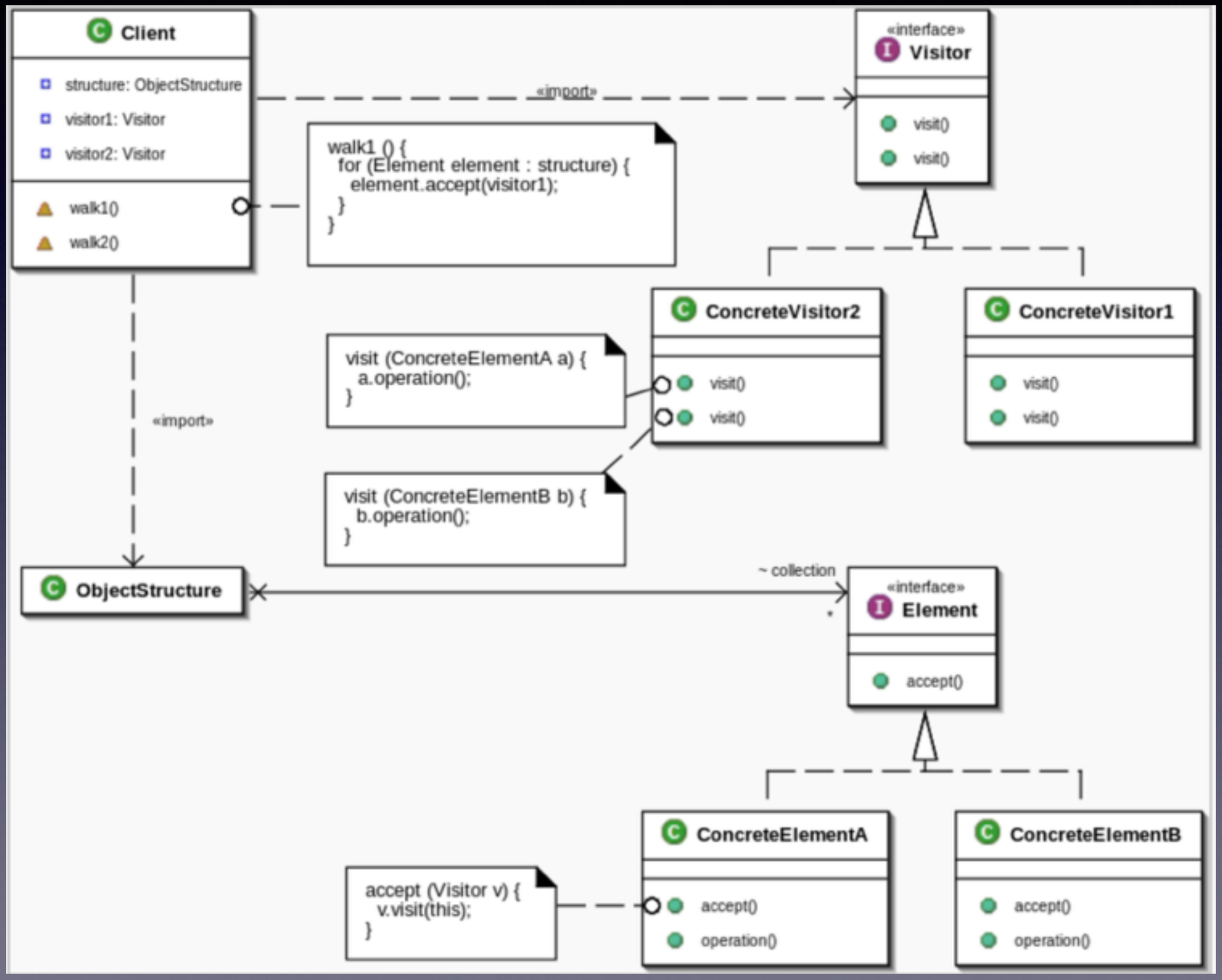


Template Method

使用多态特性将算法中不变的部分封装，扩展可变部分，提取公共代码，便于维护；

通常用到抽象类或泛型等，是代码不易读

Visitor


Client
structure: ObjectStructure
visitor1: Visitor
visitor2: Visitor
walk1()
walk2()
«import»
walk1 () {
for (Element element : structure) {
element.accept(visitor1);
}
}
«interface»
Visitor
visit()
visit()
ConcreteVisitor2
visit()
visit()
ConcreteVisitor1
visit()
visit()
visit (ConcreteElementA a) {
a.operation();
}
visit (ConcreteElementB b) {
b.operation();
}
«import»
ObjectStructure
~ collection
«interface»
Element
accept()
ConcreteElementA
accept()
operation()
ConcreteElementB
accept()
operation()
accept (Visitor v) {
v.visit(this);
}

Visitor

```
interface Visitor {
    void visit(Wheel wheel);
    void visit(Engine engine);
    void visit(Body body);
    void visit(Car car);
}

class Wheel {
    private String name;
    Wheel(String name) {
        this.name = name;
    }
    String getName() {
        return this.name;
    }
    void accept(Visitor visitor) {
        visitor.visit(this);
    }
}

class Engine {
    void accept(Visitor visitor) {
        visitor.visit(this);
    }
}

class Body {
    void accept(Visitor visitor) {
        visitor.visit(this);
    }
}
```

```
class Car {
    private Engine  engine = new Engine();
    private Body    body   = new Body();
    private Wheel[] wheels
        = { new Wheel("front left"), new Wheel("front right"),
            new Wheel("back left") , new Wheel("back right")  };
    void accept(Visitor visitor) {
        visitor.visit(this);
        engine.accept(visitor);
        body.accept(visitor);
        for (int i = 0; i < wheels.length; ++ i)
            wheels[i].accept(visitor);
    }
}

class PrintVisitor implements Visitor {
    public void visit(Wheel wheel) {
        System.out.println("Visiting " + wheel.getName()
                           + " wheel");
    }
    public void visit(Engine engine) {
        System.out.println("Visiting engine");
    }
    public void visit(Body body) {
        System.out.println("Visiting body");
    }
    public void visit(Car car) {
        System.out.println("Visiting car");
    }
}
```

```
public class VisitorDemo {
    static public void main(String[] args) {
        Car car = new Car();
        Visitor visitor = new PrintVisitor();
        car.accept(visitor);
    }
}
```

Visitor

通常用于对象结构中包含很多对象并含有不同的接口，
如果想要实施一些依赖与具体类的操作而又不想
「污染」这些类可以使用访问者模式