



南京大学

设计模式重点回顾

- GoF设计模式
 - 名，目的
 - 问题 - 解决方案(典型实现)
 - 与其它模式的关系
 - 考虑例子!
- 分类
 - 创建型
 - 结构型
 - 行为型

OO设计模式

- 重点

- Singleton：多线程安全的方案；如何实现多例模式
- Prototype：深浅拷贝实现方式
- Adapter：类适配，对象适配
- Proxy：静态代理，动态代理以及其它几种常见的代理模式
- Command：命令对列，请求日志，撤销和恢复操作
- Iterator：内部类实现方式
- Observer：观察者模式与MVC之间的关系
- Template Method：反向控制结构
- Visitor：双分派

单例模式(Singleton)

- 单例模式的定义

单例模式：确保一个类**只有一个实例**，并提供一个**全局访问点**来访问这个唯一实例。

Singleton Pattern: Ensure a class has **only one instance**, and provide **a global point** of access to it.

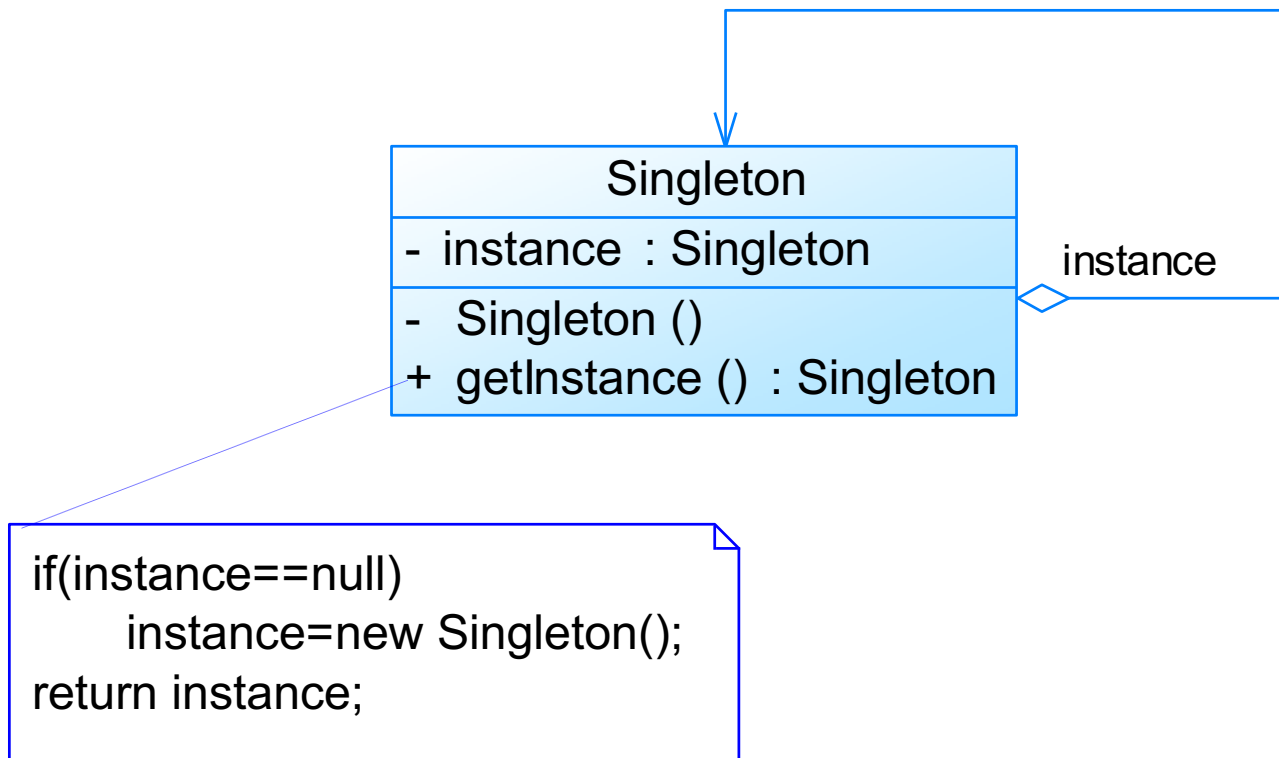
- 对象创建型模式

单例模式概述

- 单例模式的定义
 - 要点：
 - 某个类只能有一个实例
 - 必须自行创建这个实例
 - 必须自行向整个系统提供这个实例

单例模式的结构与实现

- 单例模式的结构



单例模式的结构与实现

- 单例模式的结构
 - 单例模式只包含一个单例角色：
 - Singleton（单例）

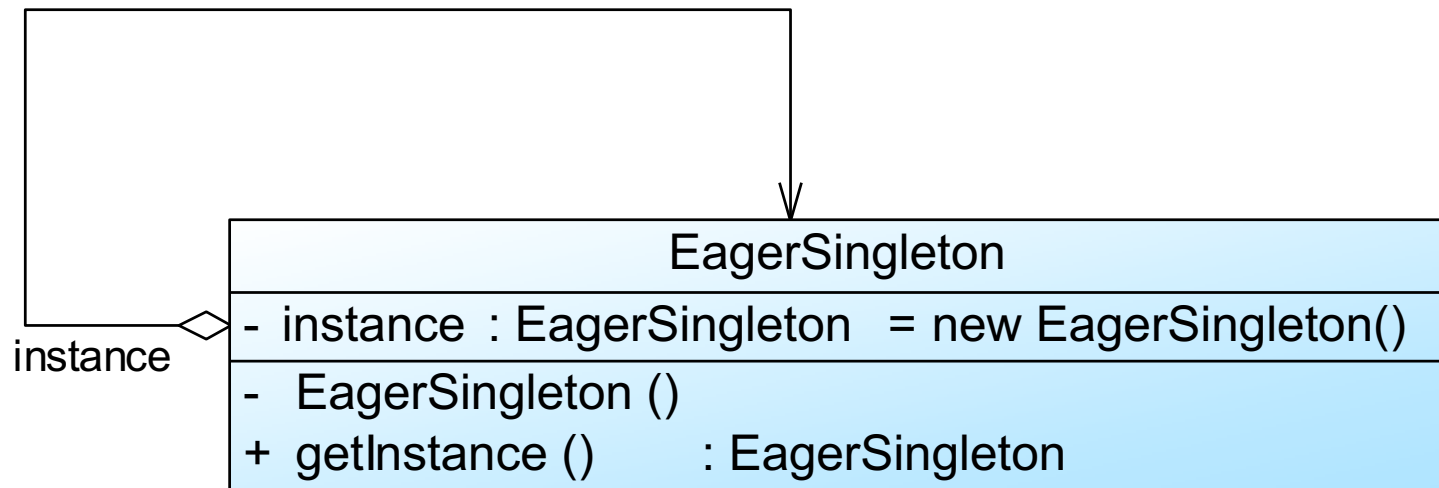
单例模式的结构与实现

- 单例模式的实现
 - 私有构造函数
 - 静态私有成员变量（自身类型）
 - 静态公有的工厂方法

```
public class Singleton {  
    private static Singleton instance=null;  
    //静态私有成员变量  
  
    //私有构造函数  
    private Singleton() {  
    }  
  
    //静态公有工厂方法，返回唯一实例  
    public static Singleton getInstance() {  
        if(instance==null)  
            instance=new Singleton();  
  
        return instance;  
    }  
}
```


饿汉式单例与懒汉式单例

- 饿汉式单例类
 - 饿汉式单例类(Eager Singleton)



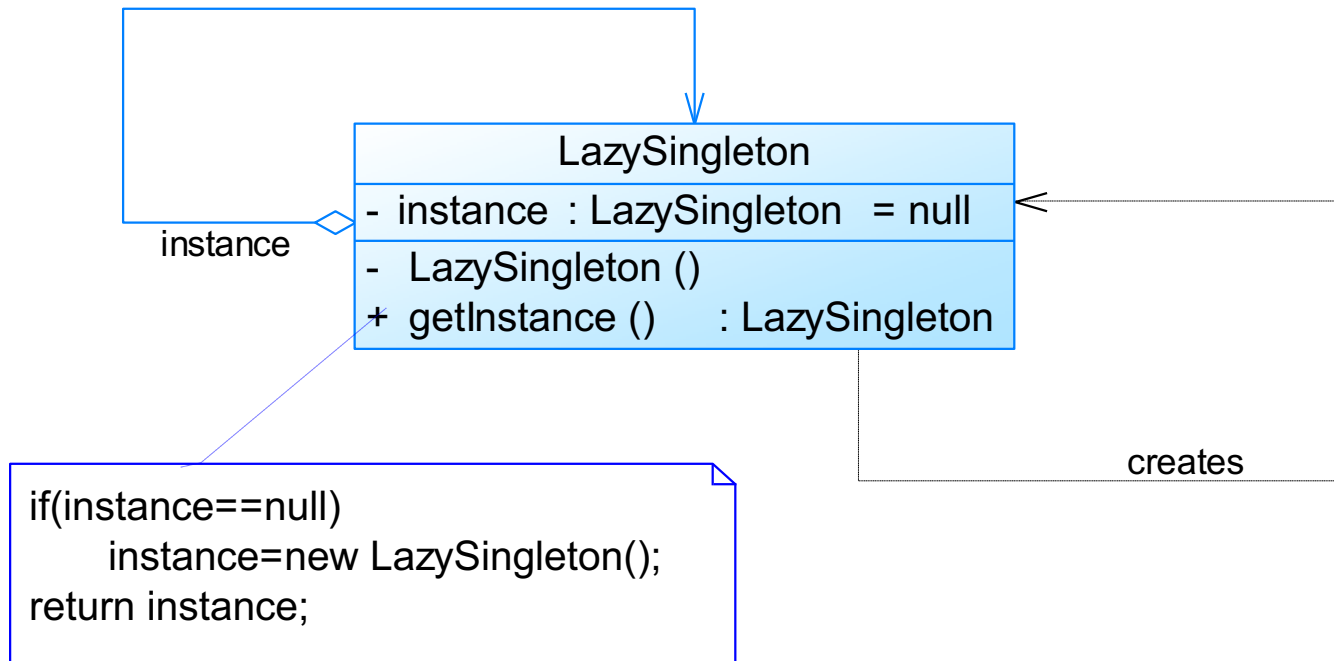
饿汉式单例与懒汉式单例

- 饿汉式单例类
 - 饿汉式单例类(Eager Singleton)

```
public class EagerSingleton {  
    private static final EagerSingleton instance = new EagerSingleton();  
    private EagerSingleton() { }  
  
    public static EagerSingleton getInstance() {  
        return instance;  
    }  
}
```

饿汉式单例与懒汉式单例

- 懒汉式单例类与双重检查锁定
 - 懒汉式单例类(Lazy Singleton)



饿汉式单例与懒汉式单例

- 懒汉式单例类与双重检查锁定

- 延迟加载

```
public class LazySingleton {  
    private static LazySingleton instance = null;  
  
    private LazySingleton() { }  
  
    public static LazySingleton getInstance() {  
        if (instance == null) {  
            instance = new LazySingleton();  
        }  
        return instance;  
    }  
}
```

多个线程同时访问将导致
创建多个单例对象！怎么办？




需要较长时间

饿汉式单例与懒汉式单例

- 懒汉式单例类与双重检查锁定
 - 延迟加载

锁方法

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



饿汉式单例与懒汉式单例

- 懒汉式单例类与双重检查锁定
 - 延迟加载

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

锁代码段



饿汉式单例与懒汉式单例

```
public class LazySingleton {  
    private volatile static LazySingleton instance = null;  
  
    private LazySingleton() { }  
  
    public static LazySingleton getInstance() {  
        //第一重判断  
        if (instance == null) {  
            //锁定代码块  
            synchronized (LazySingleton.class) {  
                //第二重判断  
                if (instance == null) {  
                    instance = new LazySingleton(); //创建单例实例  
                }  
            }  
        }  
        return instance;  
    }  
}
```

Double-Check Locking

双重检查锁定



饿汉式单例与懒汉式单例

- 饿汉式单例类与懒汉式单例类的比较
 - **饿汉式单例类**：无须考虑多个线程同时访问的问题；调用速度和反应时间优于懒汉式单例；资源利用效率不及懒汉式单例；系统加载时间可能会比较长
 - **懒汉式单例类**：实现了延迟加载；必须处理好多个线程同时访问的问题；需通过双重检查锁定等机制进行控制，将导致系统性能受到一定影响

饿汉式单例与懒汉式单例

- 使用静态内部类实现单例模式

- Java语言中最好的实现方式
- Initialization on Demand Holder (IoDH): 使用静态内部类(static inner class)

```
//Initialization on Demand Holder  
public class Singleton {  
    private Singleton() {  
    }  
}
```

//静态内部类

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

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

```
public static void main(String args[]) {  
    Singleton s1, s2;  
    s1 = Singleton.getInstance();  
    s2 = Singleton.getInstance();  
    System.out.println(s1==s2);  
}
```

原型模式(Prototype)

- 原型模式的定义

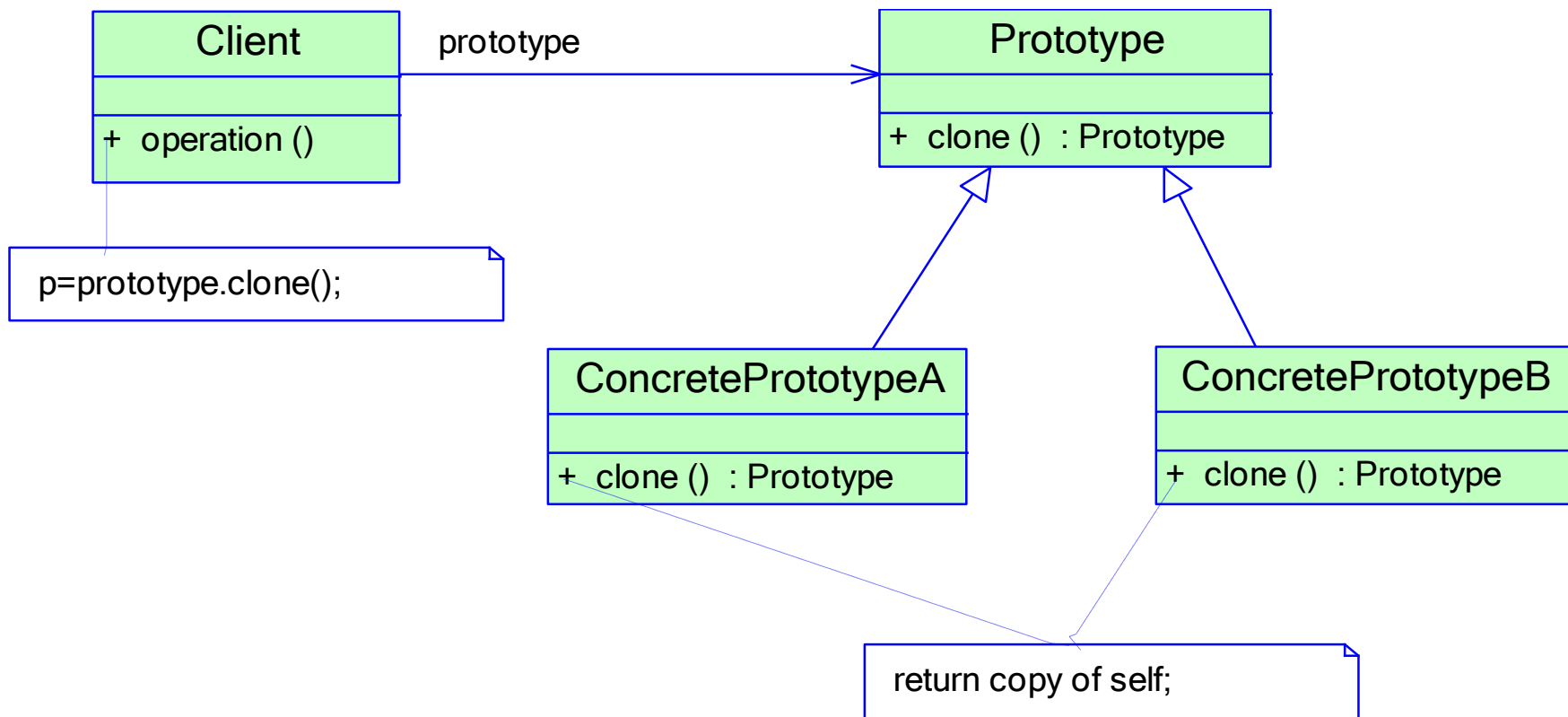
原型模式：使用原型实例指定待创建对象的类型，并且通过复制这个原型来创建新的对象。

Prototype Pattern: Specify the kinds of objects to create using a prototypical instance, and **create new objects by copying this prototype.**

- 对象创建型模式

原型模式的结构与实现

- 原型模式的结构



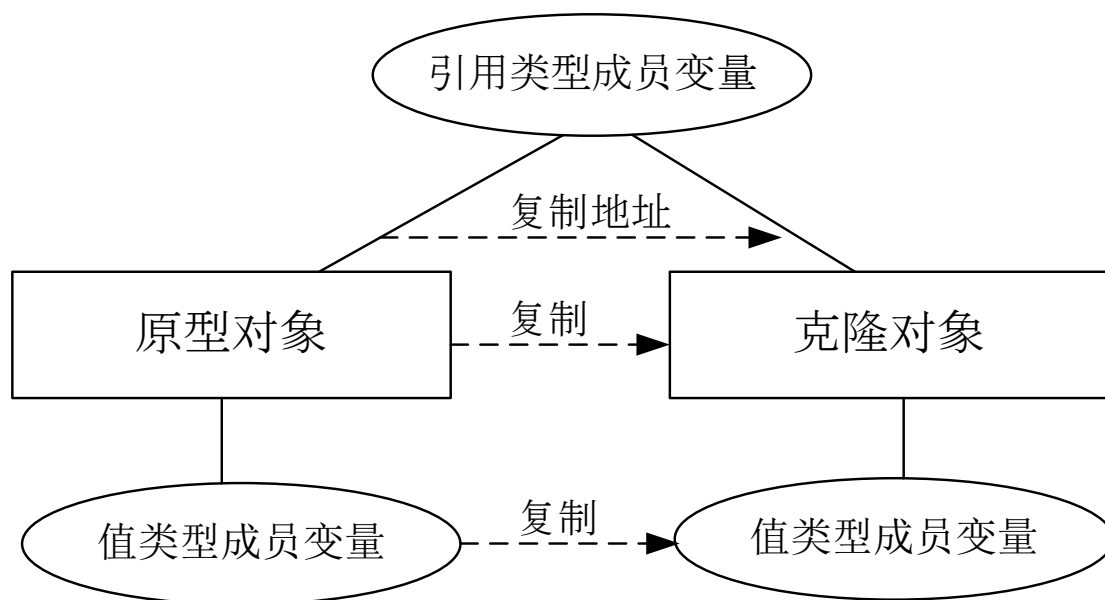
原型模式的结构与实现

- 原型模式的结构
 - 原型模式包含以下3个角色：
 - Prototype（抽象原型类）
 - ConcretePrototype（具体原型类）
 - Client（客户类）

原型模式的结构与实现

- 浅克隆与深克隆

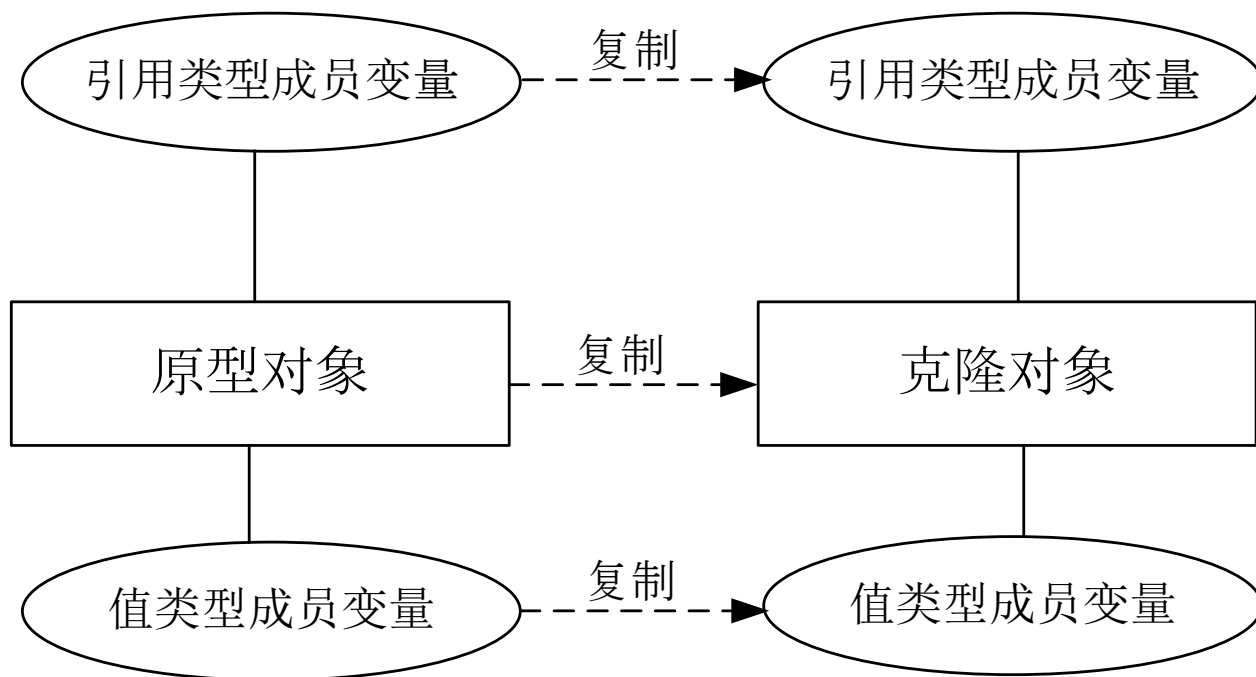
- **浅克隆(Shallow Clone)**：当原型对象被复制时，只复制它本身和其中包含的值类型的成员变量，而引用类型的成员变量并没有复制



原型模式的结构与实现

- 浅克隆与深克隆

- 深克隆(Deep Clone)：除了对象本身被复制外，对象所包含的所有成员变量也将被复制



原型模式的结构

- 原型模式的实现
 - 通用的克隆实现方法

```
public interface Prototype {  
    public Prototype clone();  
}  
  
public class ConcretePrototype implements Prototype {  
    private String attr;  
    public void setAttr(String attr) {  
        this.attr = attr;  
    }  
    public String getAttr() {  
        return this.attr;  
    }  
    //克隆方法  
    public Prototype clone() {  
        Prototype prototype = new ConcretePrototype();  
    //创建新对象  
        prototype.setAttr(this.attr);  
        return prototype;  
    }  
}
```

```
.....  
ConcretePrototype prototype = new ConcretePrototype();  
prototype.setAttr("Sunny");  
ConcretePrototype copy = (ConcretePrototype)prototype.clone();  
.....
```

原型模式的结构与实现

- 原型模式的实现
 - Java语言中的`clone()`方法和`Cloneable`接口
 - 在Java语言中，提供了一个`clone()`方法用于实现浅克隆，该方法使用起来很方便，直接调用`super.clone()`方法即可实现克隆

```
public class ConcretePrototype implements Cloneable {  
    .....  
    //Shallow Clone  
    public Prototype clone() {  
        Object object = null;  
        try {  
            object = super.clone();  
        }  
        catch (CloneNotSupportedException exception) {  
            System.err.println("Not support cloneable");  
        }  
        return (Prototype )object;  
    }  
    .....  
}
```

```
Prototype protptype = new ConcretePrototype();  
Prototype copy = protptype.clone();
```


- 通过Serializable接口如何实现深拷贝？

适配器模式概述

- 适配器模式的定义

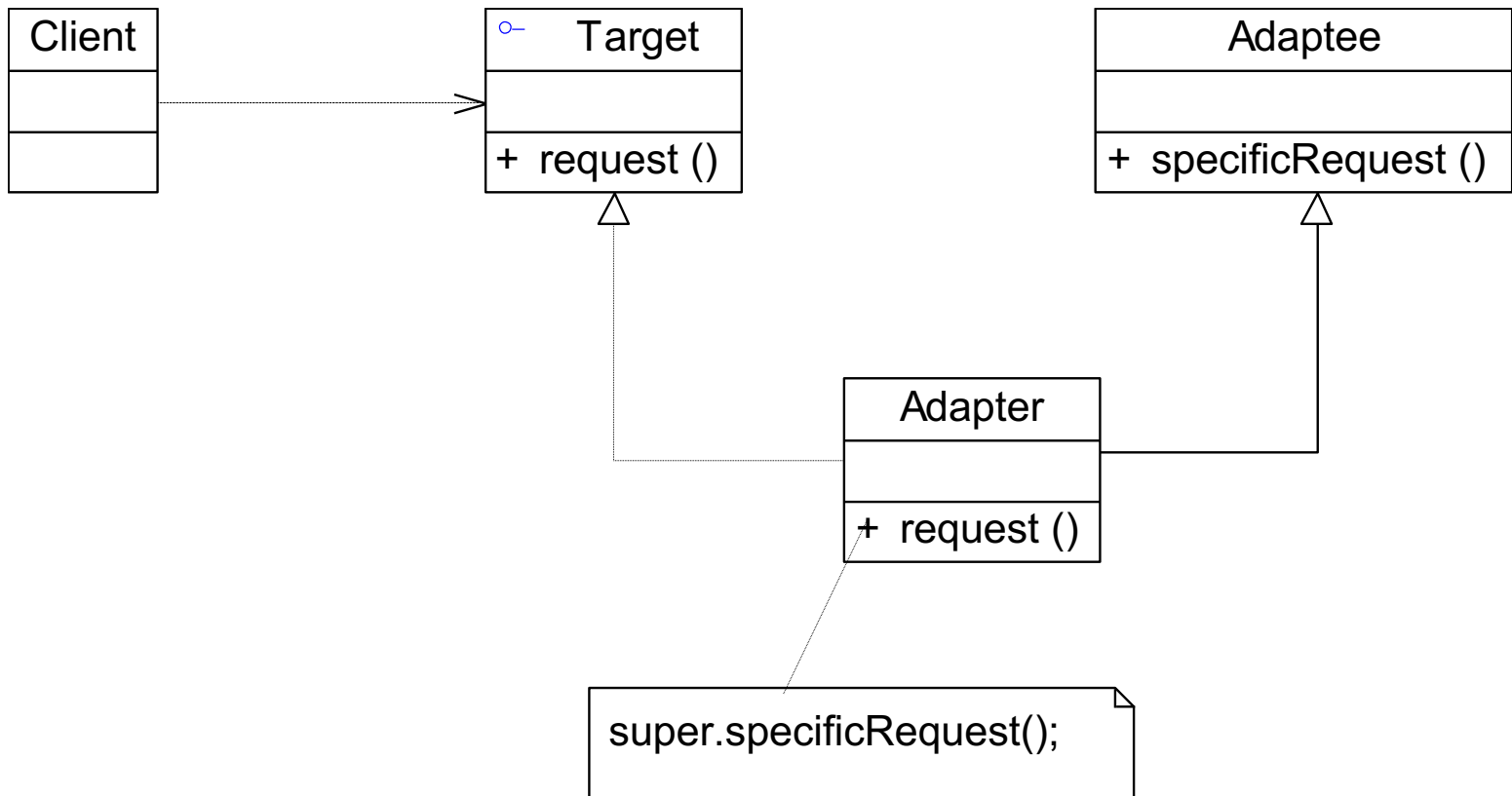
适配器模式： 将一个类的接口转换成客户希望的另一个接口。
适配器模式让那些接口不兼容的类可以一起工作。

Adapter Pattern: Convert the interface of a class into another interface clients expect. Adapter lets classes **work together** that couldn't otherwise because of **incompatible** interfaces.

- 对象结构型模式 / 类结构型模式

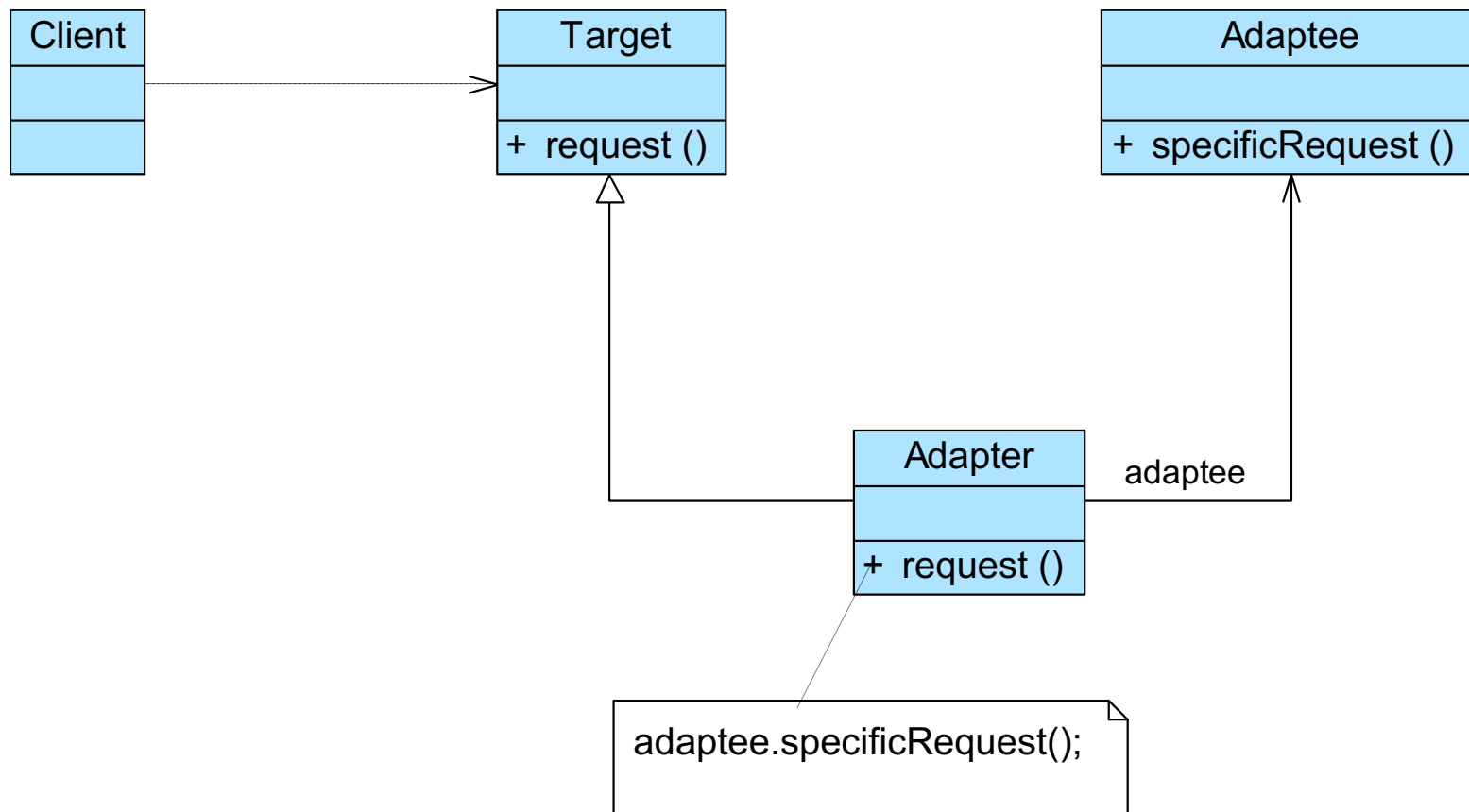
适配器模式的结构与实现

- 适配器模式的结构（类适配器）



适配器模式的结构与实现

- 适配器模式的结构（对象适配器）



适配器模式的结构与实现

- 适配器模式的结构
 - 适配器模式包含以下3个角色：
 - Target（目标抽象类）
 - Adapter（适配器类）
 - Adaptee（适配者类）

适配器模式的结构与实现

- 适配器模式的实现
 - 典型的类适配器代码：

```
public class Adapter extends Adaptee implements Target {  
    public void request() {  
        super.specificRequest();  
    }  
}
```

适配器模式的结构与实现

- 适配器模式的实现
 - 典型的对象适配器代码：

```
public class Adapter extends Target {  
    private Adaptee adaptee; //维持一个对适配者对象的引用  
  
    public Adapter(Adaptee adaptee) {  
        this.adaptee=adaptee;  
    }  
  
    public void request() {  
        adaptee.specificRequest(); //转发调用  
    }  
}
```

代理模式概述

- 代理模式的定义

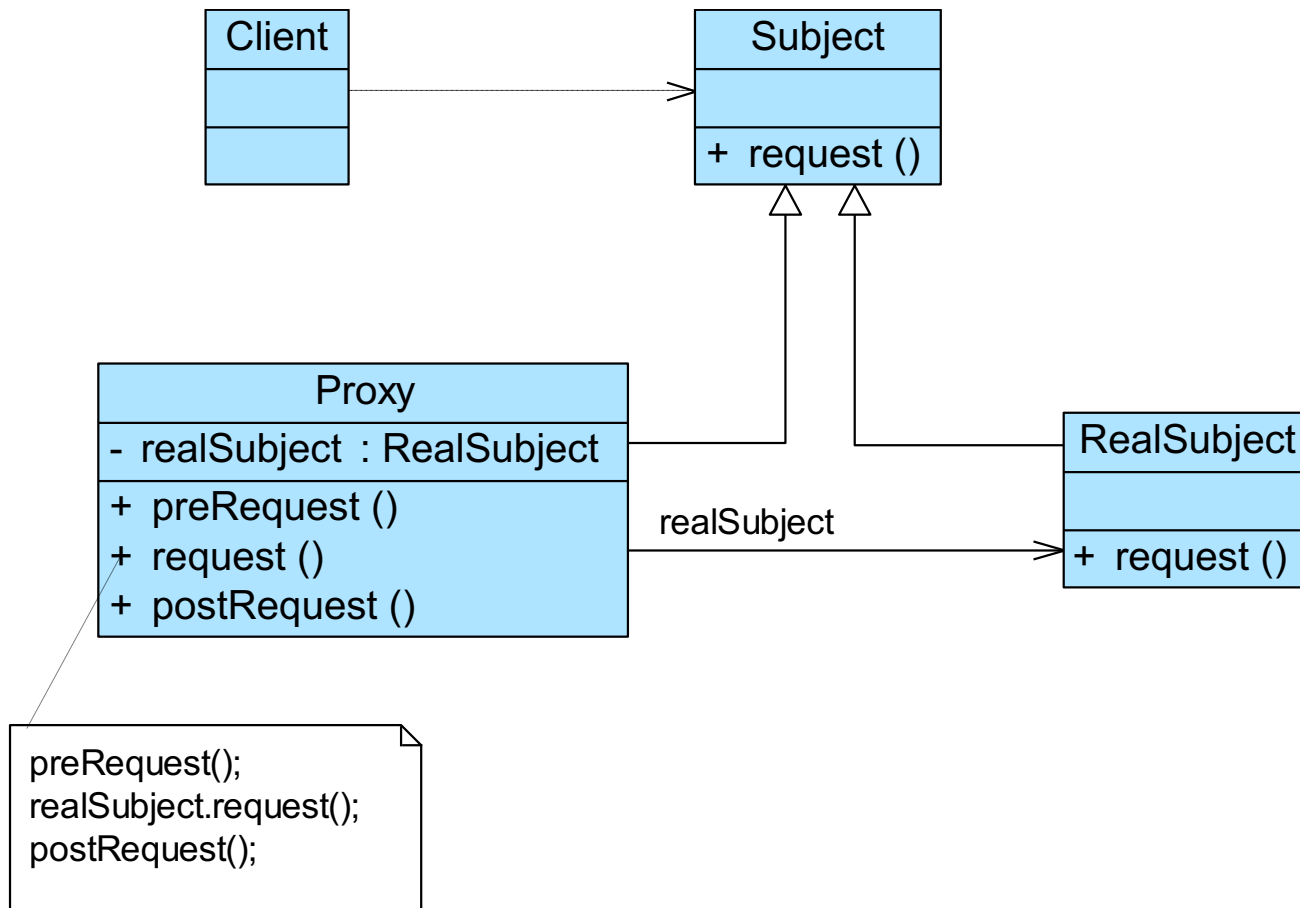
代理模式：给某一个对象提供一个代理或占位符，并由代理对象来控制对原对象的访问。

Proxy Pattern: Provide **a surrogate or placeholder** for another object to control access to it.

- 对象结构型模式

代理模式的结构与实现

- 代理模式的结构



代理模式的结构与实现

- 代理模式的结构
 - 代理模式包含以下3个角色：
 - Subject（抽象主题角色）
 - Proxy（代理主题角色）
 - RealSubject（真实主题角色）

代理模式的结构与实现

- 代理模式的实现
 - 抽象主题类典型代码：

```
public abstract class Subject {  
    public abstract void request();  
}
```

代理模式的结构与实现

- 代理模式的实现
 - 真实主题类典型代码：

```
public class RealSubject extends Subject{  
    public void request() {  
        //业务方法具体实现代码  
    }  
}
```

代理模式的结构与实现

- 代理模式的实现

```
public class Proxy extends Subject {  
    private RealSubject realSubject = new RealSubject(); //维持一个对  
真实主题对象的引用  
    public void preRequest() {  
        .....  
    }  
  
    public void request() {  
        preRequest();  
        realSubject.request(); //调用真实主题对象的方法  
        postRequest();  
    }  
  
    public void postRequest() {  
        .....  
    }  
}
```

代理模式的结构与实现

- 几种常见的代理模式
 - 远程代理(Remote Proxy)：为一个位于不同的地址空间的对象提供一个本地的代理对象，这个不同的地址空间可以在同一台主机中，也可以在另一台主机中，远程代理又称为大使(Ambassador)
 - 虚拟代理(Virtual Proxy)：如果需要创建一个资源消耗较大的对象，先创建一个消耗相对较小的对象来表示，真实对象只在需要时才会被真正创建

代理模式的结构与实现

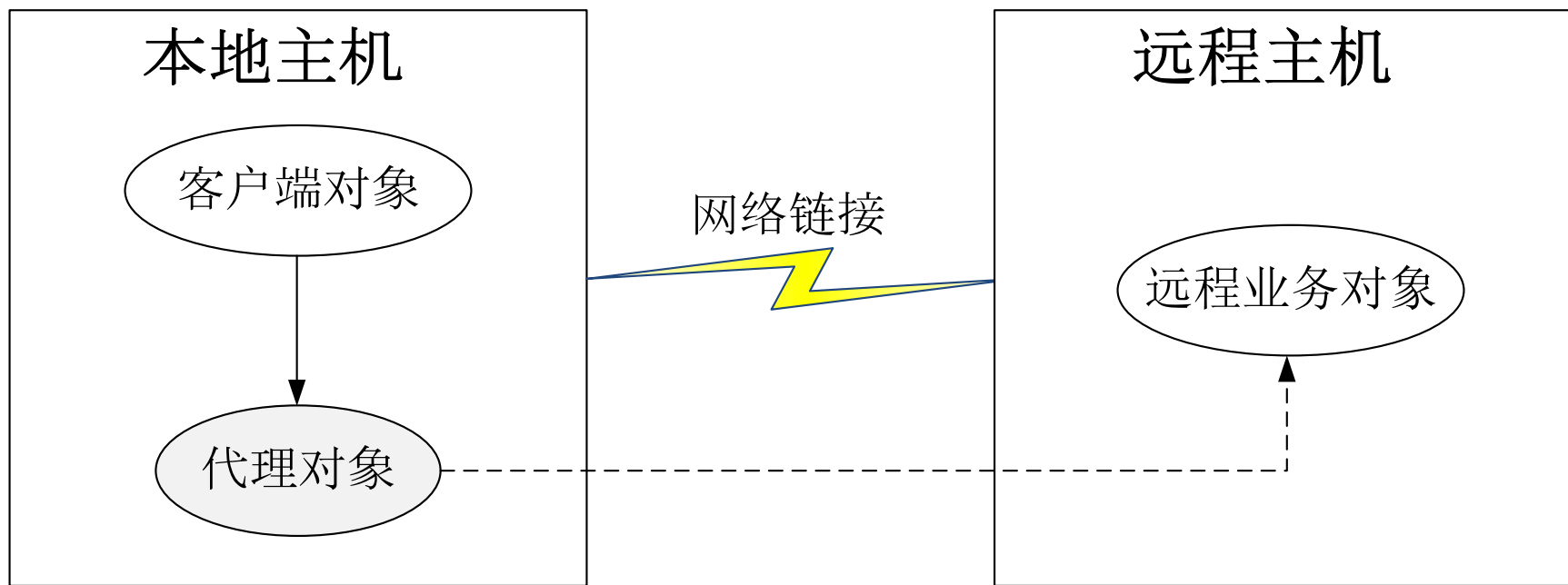
- 几种常见的代理模式
 - 保护代理(Protect Proxy)：控制对一个对象的访问，可以给不同的用户提供不同级别的使用权限
 - 缓冲代理(Cache Proxy)：为某一个目标操作的结果提供临时的存储空间，以便多个客户端可以共享这些结果
 - 智能引用代理(Smart Reference Proxy)：当一个对象被引用时，提供一些额外的操作，例如将对象被调用的次数记录下来等

远程代理

- 动机
 - 客户端程序可以访问在远程主机上的对象，远程主机可能具有更好的计算性能与处理速度，可以快速地响应并处理客户端的请求
 - 可以将网络的细节隐藏起来，使得客户端不必考虑网络的存在
 - 客户端完全可以认为被代理的远程业务对象是在本地而不是在远程，而远程代理对象承担了大部分的网络通信工作，并负责对远程业务方法的调用

远程代理

- 结构



Java RMI (Remote Method Invocation)

虚拟代理

- 动机

- 对于一些占用系统资源较多或者加载时间较长以给这些对象提供一个虚拟代理
- 在真实对象创建成功之前虚拟代理扮演真实对象而当真实对象创建之后，虚拟代理将用户的请求转发给真实对象
- 使用一个“虚假”的代理对象来代表真实对象对象来间接引用真实对象，可以在一定程度上提高性能

快捷方式



虚拟代理

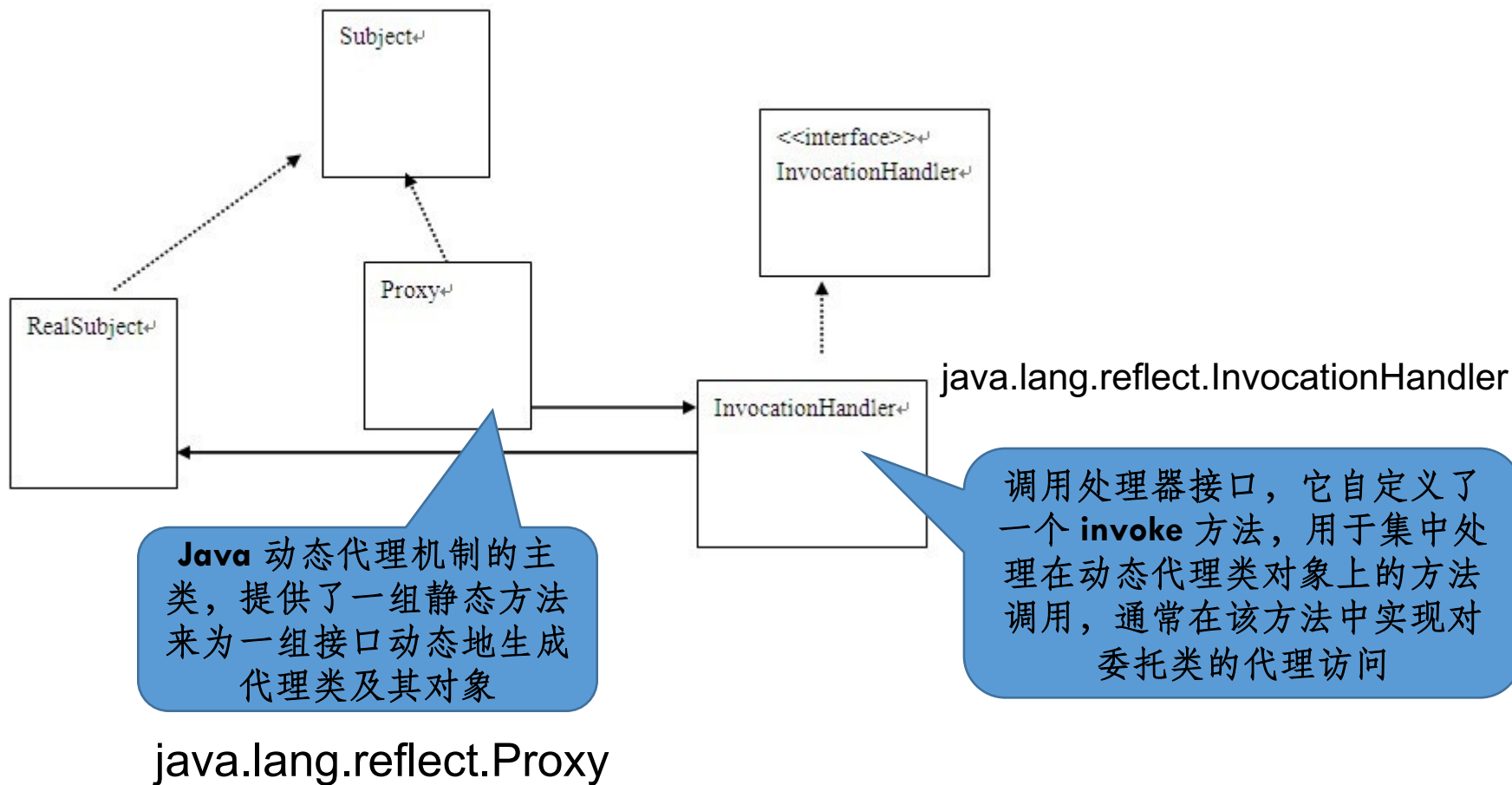
- 应用

- 由于对象本身的复杂性或者网络等原因导致一个对象需要较长的加载时间，此时可以用一个加载时间相对较短的代理对象来代表真实对象（结合多线程技术）
- 一个对象的加载十分耗费系统资源，让那些占用大量内存或处理起来非常复杂的对象推迟到使用它们的时候才创建，而在此之前用一个相对来说占用资源较少的代理对象来代表真实对象，再通过代理对象来引用真实对象（用时间换取空间）

Java动态代理

- 动态代理(Dynamic Proxy)可以让系统在运行时根据实际需要来动态创建代理类，让同一个代理类能够代理多个不同的真实主题类而且可以代理不同的方法
- Java语言提供了对动态代理的支持，Java语言实现动态代理时需要用到位于java.lang.reflect包中的一些类

- Java Dynamic Proxy



Java动态代理

- Proxy类

- `public static Class<?> getProxyClass(ClassLoader loader, Class<?>... interfaces)`：该方法用于返回一个Class类型的代理类，在参数中需要提供类加载器并需要指定代理的接口数组（与真实主题类的接口列表一致）
- `public static Object newProxyInstance(ClassLoader loader, Class<?>[] interfaces, InvocationHandler h)`：该方法用于返回一个动态创建的代理类的实例，方法中第一个参数loader表示代理类的类加载器，第二个参数interfaces表示代理类所实现的接口列表（与真实主题类的接口列表一致），第三个参数h表示所指派的调用处理程序类

Java动态代理

- InvocationHandler接口

- InvocationHandler接口是代理处理程序类的实现接口，该接口作为代理实例的调用处理者的公共父类，每一个代理类的实例都可以提供一个相关的具体调用处理者（InvocationHandler接口的子类）
- `public Object invoke(Object proxy, Method method, Object[] args)`：该方法用于处理对代理类实例的方法调用并返回相应的结果，当一个代理实例中的业务方法被调用时将自动调用该方法。
invoke()方法包含三个参数，其中第一个参数proxy表示代理类的实例，第二个参数method表示需要代理的方法，第三个参数args表示代理方法的参数数组

- 简化的动态代理对象创建过程：

```
// InvocationHandlerImpl 实现了 InvocationHandler 接口，并能实现方法调用从  
//代理类到委托类的分派转发
```

```
InvocationHandler handler = new InvocationHandlerImpl(..);
```

```
// 通过 Proxy 直接创建动态代理类实例
```

```
Interface proxy = (Interface)Proxy.newProxyInstance( classLoader,  
    new Class[] { Interface.class }, handler );
```


Java动态代理

- 动态代理类需要在运行时指定所代理真实主题类的接口，客户端在调用动态代理对象的方法时，调用请求会将请求自动转发给InvocationHandler对象的invoke()方法，由invoke()方法来实现对请求的统一处理。

命令模式概述

- 命令模式的定义

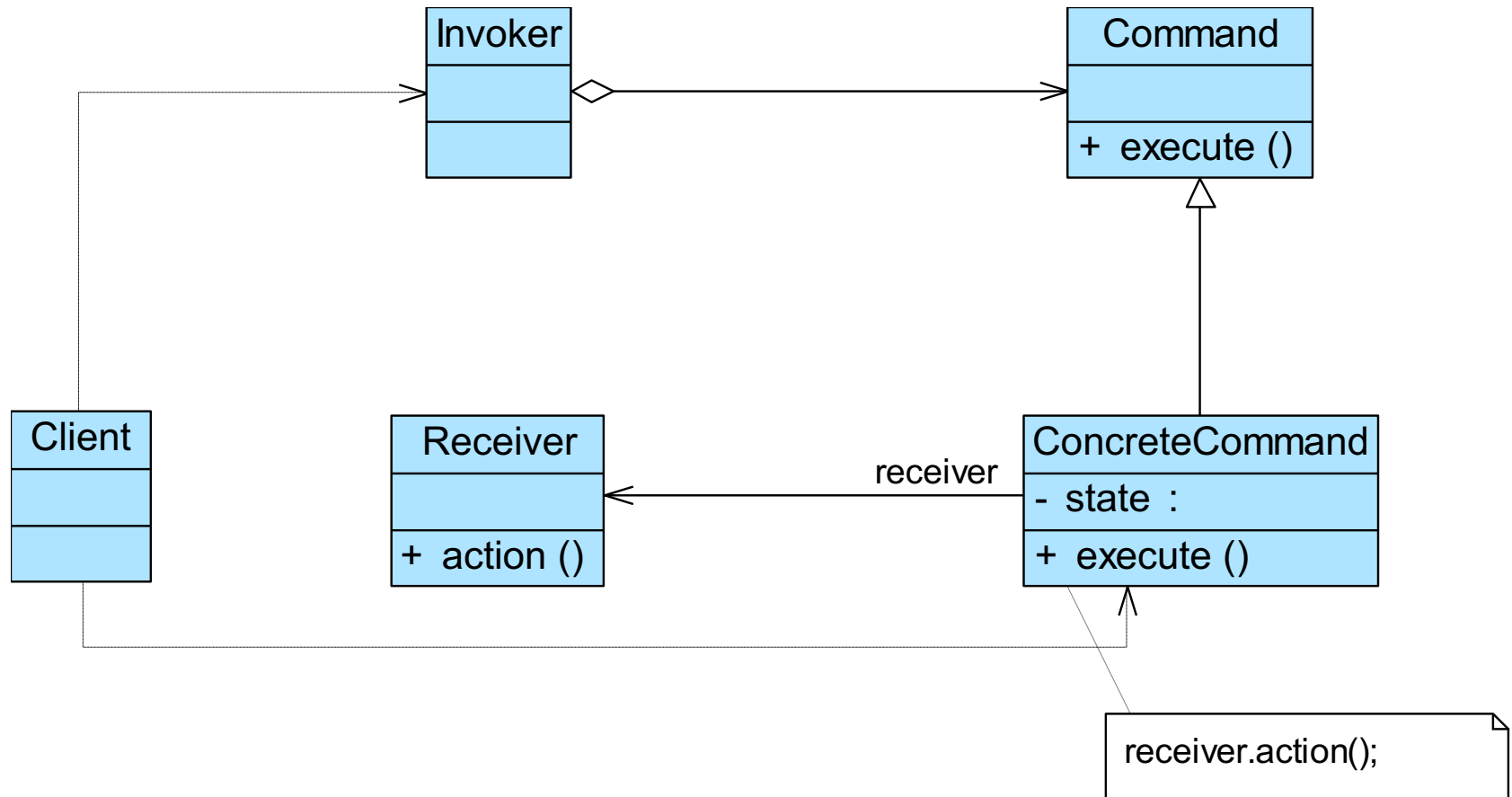
命令模式：将一个请求封装为一个对象，从而可以用不同的请求对客户进行参数化，对请求排队或者记录请求日志，以及支持可撤销的操作。

Command Pattern: Encapsulate a request as an object, thereby letting you **parameterize clients** with different requests, **queue or log requests**, and **support undoable operations**.

- 对象行为型模式

命令模式的结构与实现

- 命令模式的结构



命令模式的结构与实现

- 命令模式的结构
 - 命令模式包含以下4个角色：
 - Command（抽象命令类）
 - ConcreteCommand（具体命令类）
 - Invoker（调用者）
 - Receiver（接收者）

命令模式的结构与实现

- 命令模式的实现
 - 命令模式的本质是对请求进行封装
 - 一个请求对应于一个命令，将发出命令的责任和执行命令的责任分开
 - 命令模式允许请求的一方和接收的一方独立开来，使得请求的一方不必知道接收请求的一方的接口，更不必知道请求如何被接收、操作是否被执行、何时被执行，以及是怎么被执行的

命令模式的结构与实现

- 命令模式的实现
 - 典型的抽象命令类代码：

```
public abstract class Command {  
    public abstract void execute();  
}
```

命令模式的结构与实现

```
public class Invoker {  
    private Command command;  
  
    //构造注入  
    public Invoker(Command command) {  
        this.command = command;  
    }  
  
    //设值注入  
    public void setCommand(Command command) {  
        this.command = command;  
    }  
  
    //业务方法，用于调用命令类的execute()方法  
    public void call() {  
        command.execute();  
    }  
}
```

命令模式的结构与实现

- 命令模式的实现
 - 典型的**具体命令类**代码：

```
public class ConcreteCommand extends Command {  
    private Receiver receiver; //维持一个对请求接收者对象的引用  
  
    public void execute() {  
        receiver.action(); //调用请求接收者的业务处理方法action()  
    }  
}
```


命令模式的结构与实现

- 命令模式的实现
 - 典型的请求接收者类代码：

```
public class Receiver {  
    public void action() {  
        //具体操作  
    }  
}
```

实现命令队列

- 动机
 - 当一个请求发送者发送一个请求时，有不止一个请求接收者产生响应，这些请求接收者将逐个执行业务方法，完成对请求的处理
 - 增加一个CommandQueue类，由该类负责存储多个命令对象，而不同的命令对象可以对应不同的请求接收者
 - 批处理

实现命令队列

```
import java.util.*;

public class Invoker {
    //维持一个CommandQueue对象的引用
    private CommandQueue commandQueue;

    //构造注入
    public Invoker(CommandQueue commandQueue) {
        this.commandQueue = commandQueue;
    }

    //设值注入
    public void setCommandQueue (CommandQueue commandQueue) {
        this.commandQueue = commandQueue;
    }

    //调用CommandQueue类的execute()方法
    public void call() {
        commandQueue.execute();
    }
}
```

记录请求日志

- 动机
 - 将请求的历史记录保存下来，通常以日志文件(Log File)的形式永久存储在计算机中
 - 为系统提供一种恢复机制
 - 可以用于实现批处理
 - 防止因为断电或者系统重启等原因造成请求丢失，而且可以避免重新发送全部请求时造成某些命令的重复执行

记录请求日志

- 实现
 - 将发送请求的命令对象通过序列化写到日志文件中
 - 命令类必须实现接口 `Serializable`



实现撤销操作

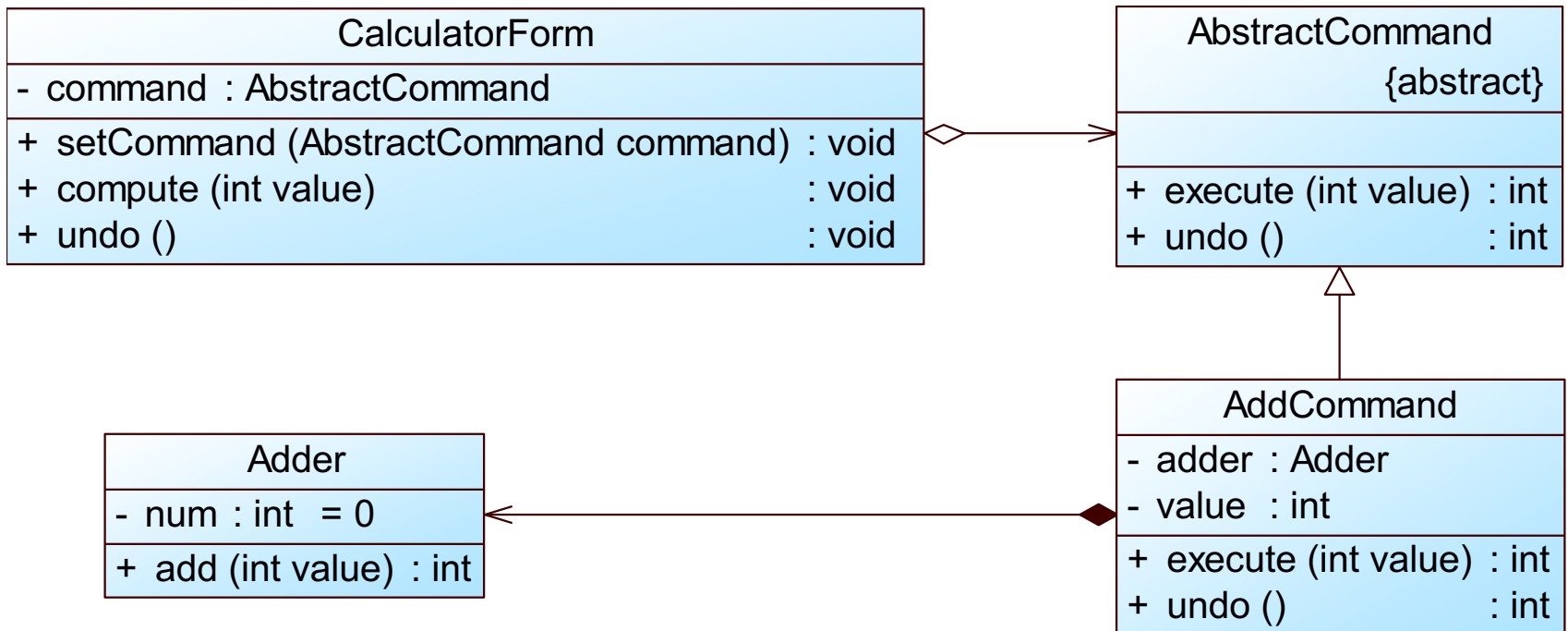
- 实例

- 可以通过对命令类进行修改使得系统支持撤销(Undo)操作和恢复(Redo)操作

设计一个简易计算器，该计算器可以实现简单的数学运算，还可以对运算实施撤销操作。

实现撤销操作

- 结构



简易计算器结构图

实现撤销操作

- 实现
 - 加法类：Adder（请求接收者）
 - 抽象命令类：AbstractCommand
 - 加法命令类：AddCommand（具体命令类）
 - 计算器界面类：CalculatorForm（请求发送者）
 - 客户端测试类：Client

演示.....

Code (designpatterns.command.calculator)

迭代器模式概述

- 迭代器模式的定义

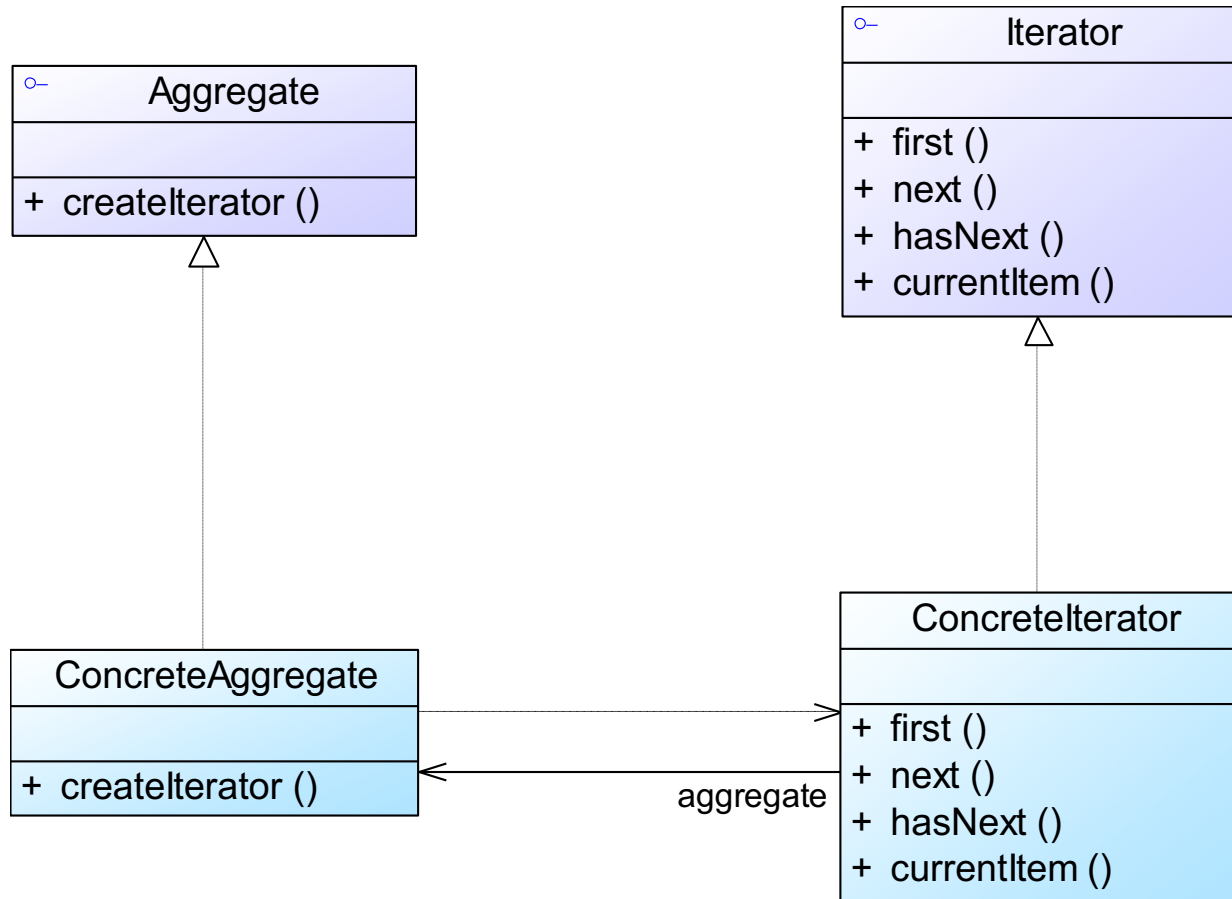
迭代器模式：提供一种方法顺序访问一个聚合对象中各个元素，且不用暴露该对象的内部表示。

Iterator Pattern: Provide a way to **access the elements of an aggregate object** sequentially **without exposing its underlying representation**.

- 对象行为型模式

迭代器模式的结构与实现

- 迭代器模式的结构



迭代器模式的结构与实现

- 迭代器模式的结构
 - 迭代器模式包含以下4个角色：
 - Iterator（抽象迭代器）
 - ConcreteIterator（具体迭代器）
 - Aggregate（抽象聚合类）
 - ConcreteAggregate（具体聚合类）

迭代器模式的结构与实现

- 迭代器模式的实现
 - 典型的抽象迭代器代码：

```
public interface Iterator {  
    public void first();           //将游标指向第一个元素  
    public void next();          //将游标指向下一个元素  
    public boolean hasNext();    //判断是否存在下一个元素  
    public Object currentItem(); //获取游标指向的当前元素  
}
```

迭代器模式的结构与实现

- 迭代器模式的实现

```
public class ConcreteIterator implements Iterator {  
    private ConcreteAggregate objects; //维持一个对具体聚合对象的引用，以便于访问存储在聚合对象中的数据  
    private int cursor; //定义一个游标，用于记录当前访问位置  
    public ConcreteIterator(ConcreteAggregate objects) {  
        this.objects=objects;  
    }  
  
    public void first() { ..... }  
  
    public void next() { ..... }  
  
    public boolean hasNext() { ..... }  
  
    public Object currentItem() { ..... }  
}
```

迭代器模式的结构与实现

- 迭代器模式的实现
 - 典型的抽象聚合类代码：

```
public interface Aggregate {  
    Iterator createIterator();  
}
```

迭代器模式的结构与实现

- 迭代器模式的实现
 - 典型的**具体聚合类**代码：

```
public class ConcreteAggregate implements Aggregate {  
    .....  
    public Iterator createIterator() {  
        return new ConcreteIterator(this);  
    }  
    .....  
}
```

使用内部类实现迭代器

- 实现
 - JDK中的AbstractList

```
package java.util;
.....
public abstract class AbstractList<E> extends AbstractCollection<E> implements
List<E> {
    .....
    private class Itr implements Iterator<E> {
        int cursor = 0;
        .....
    }
    .....
}
```


使用内部类实现迭代器

- 实现

- JDK中的AbstractList

//使用内部类实现的商品数据类

```
public class ProductList extends AbstractObjectList {  
    public ProductList(List products) {  
        super(products);  
    }  
}
```

```
public AbstractIterator createIterator() {  
    return new ProductIterator();  
}
```

//商品迭代器：具体迭代器，内部类实现

```
private class ProductIterator implements AbstractIterator {  
    private int cursor1;  
    private int cursor2;  
    //省略其他代码  
}  
}
```

观察者模式概述

- 观察者模式的定义

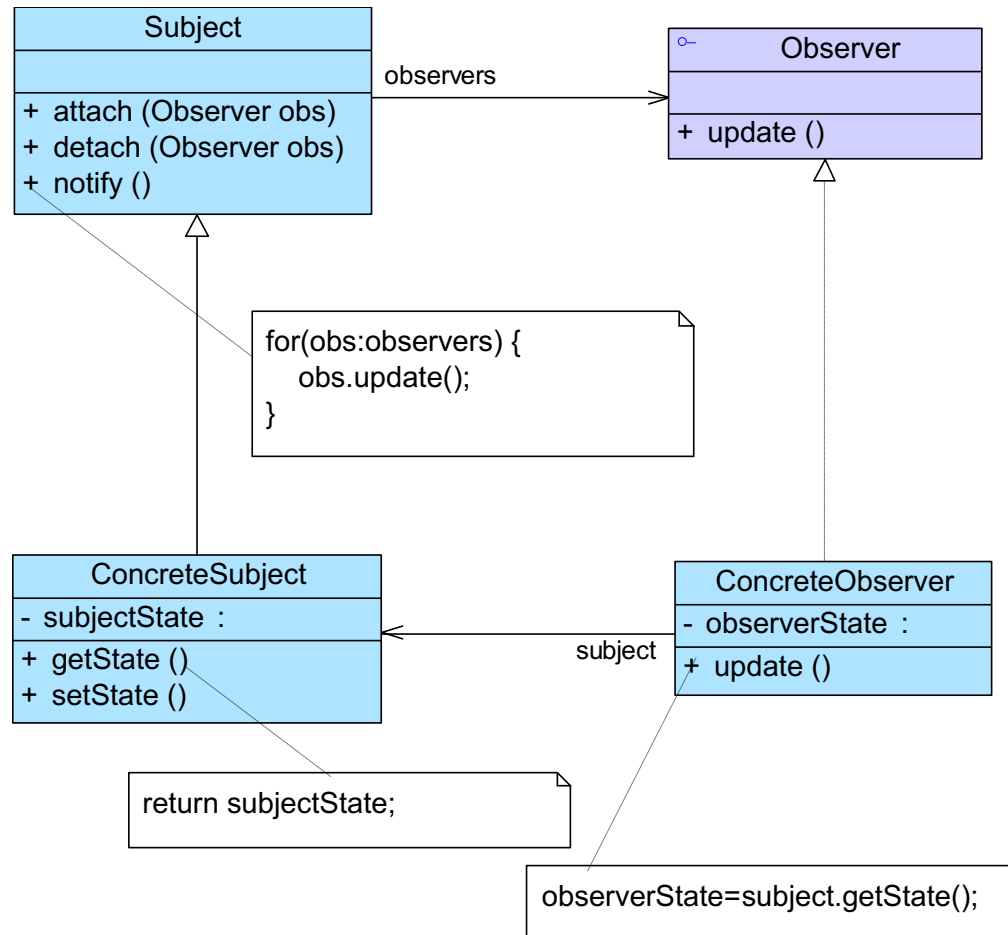
观察者模式：定义对象之间的一种**一对多依赖关系**，使得每当一个**对象状态发生改变**时，其相关依赖对象**都得到通知并被自动更新**。

Observer Pattern: Define a **one-to-many dependency** between objects so that when **one object changes state**, all its dependents are **notified and updated automatically**.

- **对象行为型**模式

观察者模式的结构与实现

- 观察者模式的结构



观察者模式的结构与实现

- 观察者模式的结构
 - 观察者模式包含以下4个角色：
 - Subject（目标）
 - ConcreteSubject（具体目标）
 - Observer（观察者）
 - ConcreteObserver（具体观察者）

观察者模式的结构与实现

```
import java.util.*;

public abstract class Subject {
    //定义一个观察者集合用于存储所有观察者对象
    protected ArrayList<Observer> observers = new ArrayList();

    //注册方法，用于向观察者集合中增加一个观察者
    public void attach(Observer observer) {
        observers.add(observer);
    }

    //注销方法，用于在观察者集合中删除一个观察者
    public void detach(Observer observer) {
        observers.remove(observer);
    }

    //声明抽象通知方法
    public abstract void notify();
}
```

观察者模式的结构与实现

- 观察者模式的实现
 - 典型的**具体目标类**代码：

```
public class ConcreteSubject extends Subject {  
    //实现通知方法  
    public void notify() {  
        //遍历观察者集合，调用每一个观察者的响应方法  
        for(Object obs:observers) {  
            ((Observer)obs).update();  
        }  
    }  
}
```

观察者模式的结构与实现

- 观察者模式的实现
 - 典型的抽象观察者代码：

```
public interface Observer {  
    //声明响应方法  
    public void update();  
}
```

观察者模式的结构与实现

- 观察者模式的实现
 - 典型的**具体**观察者代码：

```
public class ConcreteObserver implements Observer {  
    //实现响应方法  
    public void update() {  
        //具体响应代码  
    }  
}
```


观察者模式的结构与实现

- 观察者模式的实现

- 说明：

- 有时候在具体观察者类ConcreteObserver中需要使用到具体目标类ConcreteSubject中的状态（属性），会存在关联或依赖关系
 - 如果在具体层之间具有关联关系，系统的扩展性将受到一定的影响，增加新的具体目标类有时候需要修改原有观察者的代码，在一定程度上违背了开闭原则，但是如果原有观察者类无须关联新增的具体目标，则系统扩展性不受影响

观察者模式的结构与实现

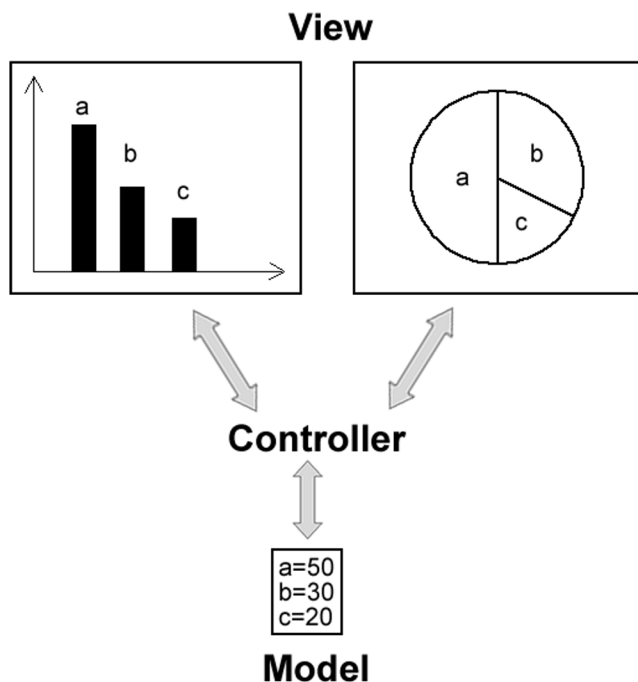
- 观察者模式的实现
 - 典型的客户端代码片段：

```
.....  
Subject subject = new ConcreteSubject();  
Observer observer = new ConcreteObserver();  
subject.attach(observer); //注册观察者  
subject.notify();  
.....
```

观察者模式与MVC

- MVC(Model-View-Controller)架构

- 模型(Model) , 视图(View)和控制器(Controller)
- 模型可对应于观察者模式中的观察目标 , 而视图对应于观察者 , 控制器可充当两者之间的中介者
- 当模型层的数据发生改变时 , 视图层将自动改变其显示内容



MVC结构示意图

模板方法模式概述

- 模板方法模式的定义

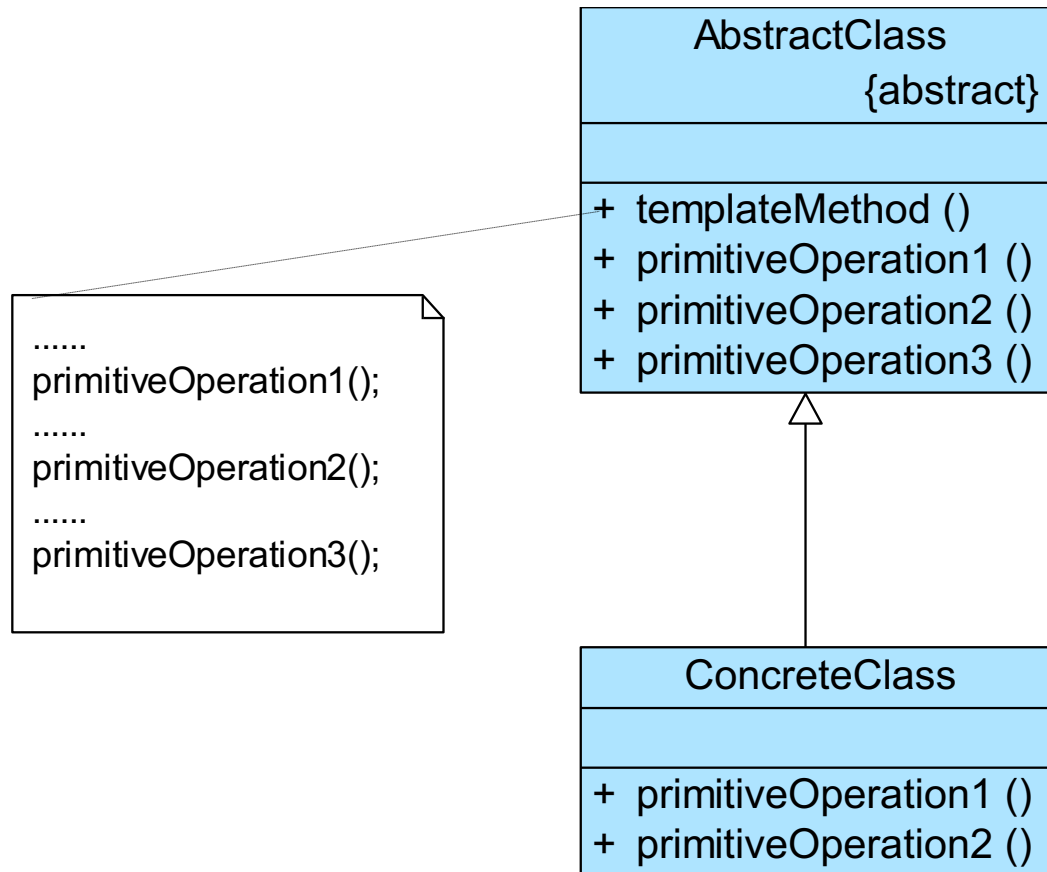
模板方法模式：定义一个操作中**算法的框架**，而**将一些步骤延迟到子类中**。模板方法模式使得子类不改变一个算法的结构即可**重定义**该算法的**某些特定步骤**。

Template Method Pattern: Define **the skeleton of an algorithm** in an operation, **deferring some steps to subclasses**. Template Method lets subclasses **redefine certain steps** of an algorithm without changing the algorithm's structure.

- 类行为型模式

模板方法模式的结构与实现

- 模板方法模式的结构



模板方法模式的结构与实现

- 模板方法模式的结构
 - 模板方法模式包含以下两个角色：
 - AbstractClass（抽象类）
 - ConcreteClass（具体子类）

模板方法模式的结构与实现

- 模板方法模式的实现
 - 模板方法 (Template Method)
 - 基本方法 (Primitive Method)
 - 抽象方法 (Abstract Method)
 - 具体方法 (Concrete Method)
 - 钩子方法 (Hook Method)

模板方法模式的结构与实现

- 模板方法模式的实现

.....

//模板方法

```
public void template() {
```

```
    open();
```

```
    display();
```

//通过钩子方法来确定某一步骤是否执行

```
    if(isPrint()) {
```

```
        print();
```

```
    }
```

```
}
```

//钩子方法

```
public boolean isPrint() {
```

```
    return true;
```

```
}
```

.....

模板方法模式的结构与实现

```
public abstract class AbstractClass {
```

//模板方法

```
public void templateMethod() {
```

```
    primitiveOperation1();
```

```
    primitiveOperation2();
```

```
    primitiveOperation3();
```

```
}
```

//基本方法—具体方法

```
public void primitiveOperation1() {
```

```
    //实现代码
```

```
}
```

//基本方法—抽象方法

```
public abstract void primitiveOperation2();
```

//基本方法—钩子方法

```
public void primitiveOperation3()
```

```
{ }
```

```
}
```

模板方法模式的结构与实现

- 模板方法模式的实现
 - 具体子类典型代码：

```
public class ConcreteClass extends AbstractClass {  
    public void primitiveOperation2() {  
        //实现代码  
    }  
  
    public void primitiveOperation3() {  
        //实现代码  
    }  
}
```

钩子方法的使用

- 实例

某软件公司要为销售管理系统提供一个数据图表显示功能，该功能的实现包括以下几个步骤：

- (1) 从数据源获取数据。
- (2) 将数据转换为XML格式。
- (3) 以某种图表方式显示XML格式的数据。

该功能支持多种数据源和多种图表显示方式，但所有的图表显示操作都基于XML格式的数据，因此**可能需要对数据进行转换，如果从数据源获取的数据已经是XML数据，则无须转换。**

钩子方法的使用

- 结构



数据图表显示功能结构图

```
//designpatterns.templateMethod.hookMethod.DataViewer.java
package designpatterns.templateMethod.hookMethod;

public abstract class DataView {
    //抽象方法：获取数据
    public abstract void getData();

    //具体方法：转换数据
    public void convertData() {
        System.out.println("将数据转换为XML格式。");
    }

    //抽象方法：显示数据
    public abstract void displayData();

    //钩子方法：判断是否为XML格式的数据
    public boolean isNotXMLData() {
        return true;
    }

    //模板方法
    public void process() {
        getData();
        //如果不是XML格式的数据则进行数据转换
        if (isNotXMLData()) {
            convertData();
        }
        displayData();
    }
}
```

```
//designpatterns.templateMethod.hookMethod.XMLDataViewer.java
package designpatterns.templateMethod.hookMethod;

public class XMLDataViewer extends DataView {
    //实现父类方法：获取数据
    public void getData() {
        System.out.println("从XML文件中获取数据。");
    }

    //实现父类方法：显示数据，默认以柱状图方式显示，可结合桥接模式来改进
    public void displayData() {
        System.out.println("以柱状图显示数据。");
    }

    //覆盖父类的钩子方法
    public boolean isNotXMLData() {
        return false;
    }
}
```

反向控制结构

- 上述例子中，可实现一种反向控制结构，通过子类覆盖父类的钩子方法来决定某一特定步骤是否需要执行

访问者模式概述

- 访问者模式的定义

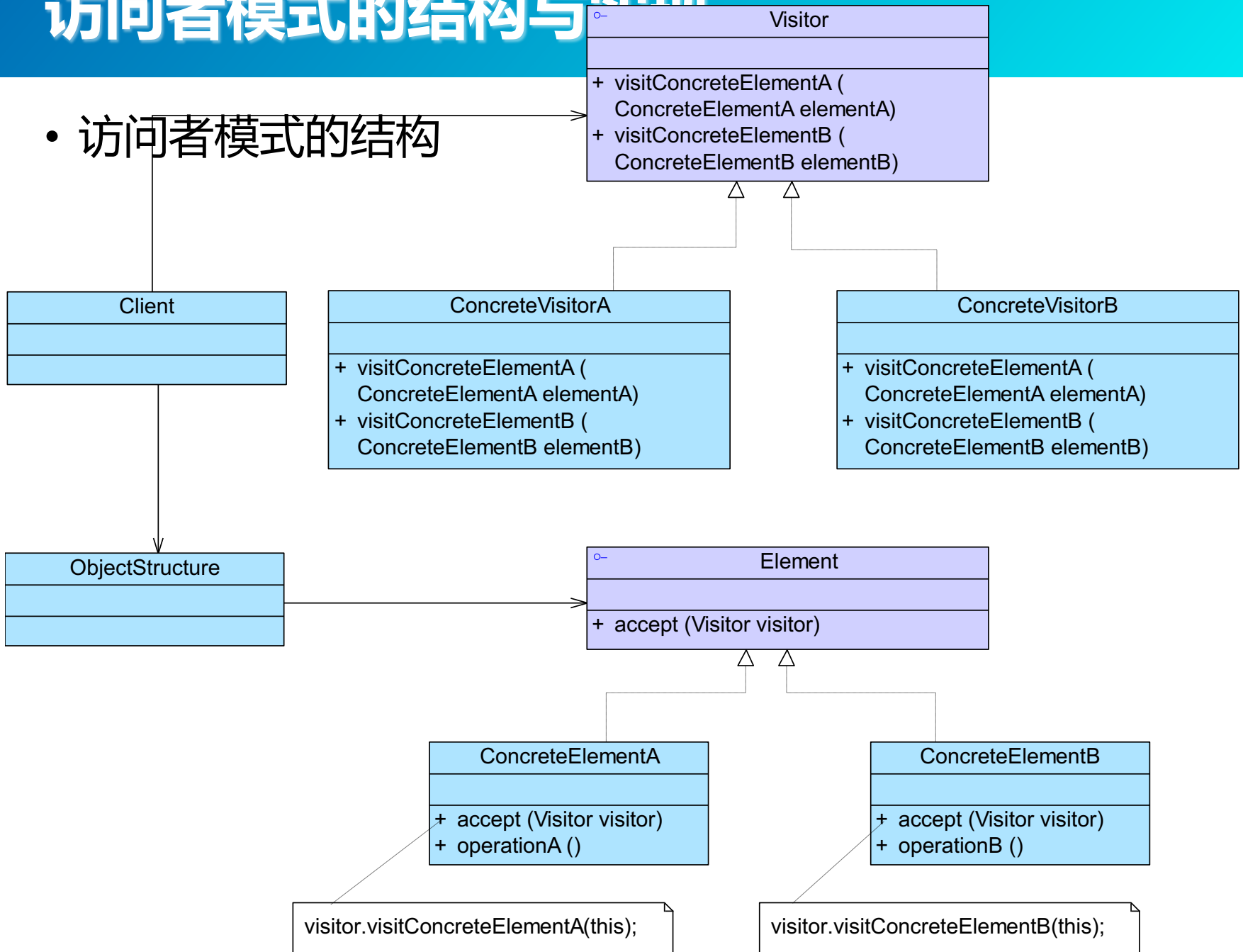
访问者模式：表示一个作用于某对象结构中的各个元素的操作。访问者模式让你可以在不改变各元素的类的前提下定义作用于这些元素的新操作。

Visitor Pattern: Represent an operation to be performed on the elements of an **object structure**. Visitor lets you **define a new operation without changing the classes of the elements** on which it operates.

- 对象行为型模式

访问者模式的结构与实现

• 访问者模式的结构



访问者模式的结构与实现

- 访问者模式的结构
 - 访问者模式包含以下5个角色：
 - Visitor（抽象访问者）
 - ConcreteVisitor（具体访问者）
 - Element（抽象元素）
 - ConcreteElement（具体元素）
 - ObjectStructure（对象结构）

访问者模式的结构与实现

- 访问者模式的实现
 - 典型的抽象访问者类代码：

```
public abstract class Visitor {  
    public abstract void visit(ConcreteElementA elementA);  
    public abstract void visit(ConcreteElementB elementB);  
  
    public void visit(ConcreteElementC elementC) {  
        //元素ConcreteElementC操作代码  
    }  
}
```

访问者模式的结构与实现

- 访问者模式的实现
 - 典型的**具体访问者类**代码：

```
public class ConcreteVisitor extends Visitor {  
    public void visit(ConcreteElementA elementA) {  
        //元素ConcreteElementA操作代码  
    }  
  
    public void visit(ConcreteElementB elementB) {  
        //元素ConcreteElementB操作代码  
    }  
}
```

访问者模式的结构与实现

- 访问者模式的实现
 - 典型的抽象元素类代码：

```
public interface Element {  
    public void accept(Visitor visitor);  
}
```

访问者模式的结构与实现

- 访问者模式的实现
 - 典型的**具体元素类**代码：

```
public class ConcreteElementA implements Element {  
    public void accept(Visitor visitor) {  
        visitor.visit(this);  
    }  
  
    public void operationA() {  
        //业务方法  
    }  
}
```

访问者模式的结构与实现

- 访问者模式的实现

- 双重分派

ConcreteElementA.accept(Visitor visitor)

- (1) 调用具体元素类的 **accept(Visitor visitor)** 方法，并将 Visitor 子类对象作为其参数

- (2) 在具体元素类 **accept(Visitor visitor)** 方法内部调用传入的

ConcreteVisitorA.visit(ConcreteElementA elementA)
<ConcreteVisitorA.visit(this)>

ConcreteElementA)，将当前具体元素类对象(this)作为参数，例如
visitor.visit(this)

- (3) 执行 Visitor 对象的 **visit()** 方法，在其中还可以调用具体元素对象的业务方法

ConcreteElementA.operationA()

```
import java.util.*;
```

```
public class ObjectStructure
```

```
{
```

```
    private ArrayList<Element> list = new ArrayList<Element>(); //定义一个集合用于存储元素对象
```

```
    //接受访问者的访问操作
```

```
    public void accept(Visitor visitor) {
```

```
        Iterator i=list.iterator();
```

```
        while(i.hasNext()) {
```

```
            ((Element)i.next()).accept(visitor); //遍历访问集合中的每一个元素
```

```
        }
```

```
    }
```

```
    public void addElement(Element element) {
```

```
        list.add(element);
```

```
    }
```

```
    public void removeElement(Element element) {
```

```
        list.remove(element);
```

```
    }
```

```
}
```


- 时间：2021年6月22日8:00-10:00
- 地点：XX
- 方式：闭卷