责任链模式（Chain of Responsibility Pattern）

责任链模式为请求创建了一个接收者对象的链。这种模式给予请求的类型，对请求的发送者和接收者进行解耦。这种类型的设计模式属于行为型模式。 在这种模式中，通常每个接收者都包含对另一个接收者的引用。如果一个对象不能处理该请求，那么它会把相同的请求传给下一个接收者，依此类推。

意图

避免请求发送者与接收者耦合在一起，让多个对象都有可能接收请求，将这些对象连接成一条链，并且沿着这条链传递请求，直到有对象处理它为止。

主要解决

职责链上的处理者负责处理请求，客户只需要将请求发送到职责链上即可，无须关心请求的处理细节和请求的传递，所以职责链将请求的发送者和请求的处理者解耦了。

何时使用

在处理消息时对消息进行多重过滤。

如何解决

拦截的类都实现统一接口。

关键代码

Handler 里面聚合它自己，在 HanleRequest 里判断是否合适，如果没达到条件则向下传递，向谁传递之前 set 进去。

应用实例

https://www.cnblogs.com/lizo/p/7503862.html

1、红楼梦中的"击鼓传花"。

2、JS 中的事件冒泡。

3、JAVA WEB 中 Apache Tomcat 对 Encoding 的处理，Struts2 的拦截器，jsp servlet 的 Filter。

优点

1、降低耦合度。它将请求的发送者和接收者解耦。

2、简化了对象。使得对象不需要知道链的结构。

3、增强给对象指派职责的灵活性。通过改变链内的成员或者调动它们的次序，允许动态地新增或者删除责任。

4、增加新的请求处理类很方便。

缺点

1、不能保证请求一定被接收。

2、系统性能将受到一定影响，而且在进行代码调试时不太方便，可能会造成循环调用。

3、可能不容易观察运行时的特征，有碍于除错。

使用场景

1、有多个对象可以处理同一个请求，具体哪个对象处理该请求由运行时刻自动确定。

2、在不明确指定接收者的情况下，向多个对象中的一个提交一个请求。

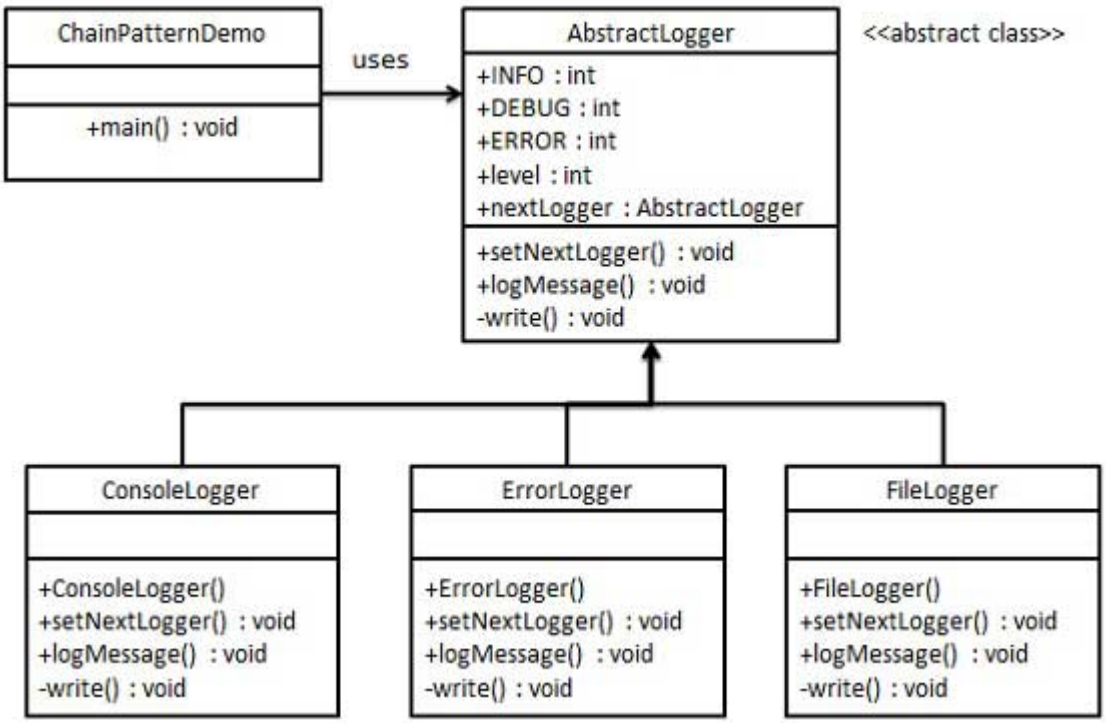
3、可动态指定一组对象处理请求。

注意事项

在 JAVA WEB 中遇到很多应用。

示例实现

我们创建抽象类 AbstractLogger，带有详细的日志记录级别。然后我们创建三种类型的记录器，都扩展了 AbstractLogger。每个记录器消息的级别是否属于自己的级别，如果是则相应地打印出来，否则将不打印并把消息传给下一个记录器。



其中抽象的log记录器类AsbtractLogger拥有下一个记录器的引用：

```
public abstract class AbstractLogger {
    public static int INFO = 1;
    public static int DEBUG = 2;
    public static int ERROR = 3;

    protected int level;
    protected AbstractLogger nextLogger;

    public void setNextLogger(AbstractLogger nextLogger) {
        this.nextLogger = nextLogger;
    }
    public void logMessage(int level, String msg) {
        if(this.level <= level) {
            write(msg);
        }
        if(nextLogger != null) {
            nextLogger.logMessage(level, msg);
        }
    }
    abstract protected void write(String msg);
}
```

其他过滤器的实现可以根据异常等级、写入方式等自定义行为。

命令模式（Command Pattern）

命令模式是一种数据驱动的设计模式，它属于行为型模式。请求以命令的形式包裹在对象中，并传给调用对象。调用对象寻找可以处理该命令的合适的对象，并把该命令传给相应的对象，该对象执行命令。

意图

将一个请求封装成一个对象，从而使您可以用不同的请求对客户进行参数化。

主要解决

在软件系统中，行为请求者与行为实现者通常是一种紧耦合的关系，但某些场合，比如需要对行为进行记录、撤销或重做、事务等处理时，这种无法抵御变化的紧耦合的设计就不太合适。
例如，我们平常在编写菜单时会写出这样的逻辑：

```
switch(command) {
case open:
    menu.changecolor()
    menu.toggle()
…
}
```

此时，调用者是知道菜单类内的成员的，他们成耦合的关系，如果要修改open的逻辑则必须直接在这块switch中进行修改，非常不利于扩展，所以引入了命令模式，将这块open逻辑打包传给menu，再由menu回调执行。
解耦调用操作的对象和具有执行该操作所需信息的对象。

何时使用

在某些场合，比如要对行为进行"记录、撤销/重做、事务"等处理，这种无法抵御变化的紧耦合是不合适的。在这种情况下，如何将"行为请求者"与"行为实现者"解耦？将一组行为抽象为对象，可以实现二者之间的松耦合。

如何解决

通过调用者调用接受者执行命令，顺序：调用者→接受者→命令。

关键代码

定义三个角色
1、received 真正的命令执行对象
2、Command
3、invoker 使用命令对象的入口

应用实例

struts 1 中的 action 核心控制器 ActionServlet 只有一个，相当于 Invoker，而模型层的类会随着不同的应用有不同的模型类，相当于具体的 Command。

优点

1、降低了系统耦合度。
2、新的命令可以很容易添加到系统中去。

缺点

使用命令模式可能会导致某些系统有过多的具体命令类。

使用场景

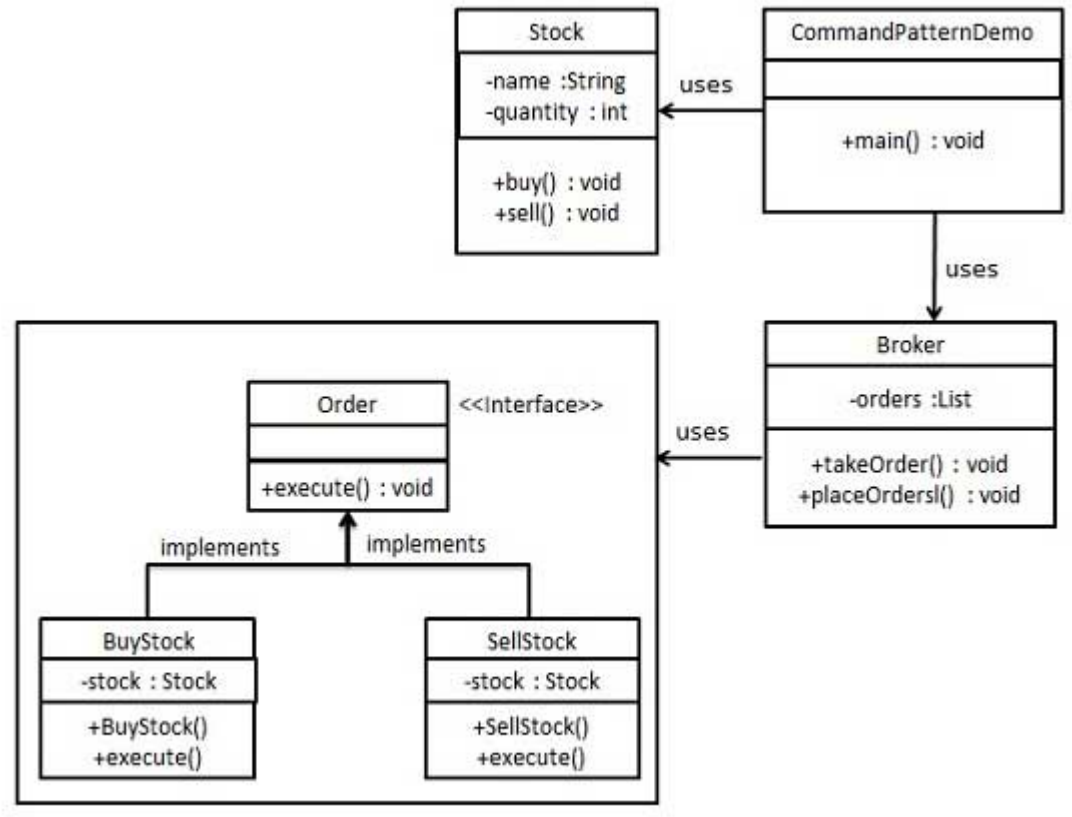
认为是命令的地方都可以使用命令模式，
比如：1、GUI 中每一个按钮都是一条命令。
2、模拟 CMD。

注意事项

系统需要支持命令的撤销(Undo)操作和恢复(Redo)操作，也可以考虑使用命令模式，见命令模式的扩展。

示例实现

我们首先创建作为命令的接口 Order，然后创建作为请求的 Stock 类。实体命令类 BuyStock 和 SellStock，实现了 Order 接口，将执行实际的命令处理。创建作为调用对象的类 Broker，它接受订单并能下订单。
Broker 对象使用命令模式，基于命令的类型确定哪个对象执行哪个命令。CommandPatternDemo，我们的演示类使用 Broker 类来演示命令模式。



Broker相当于一个指令执行器，可以对Stock执行指定的Order。

```
1 public class Broker {
```

```
    private List<Order> orderList = new ArrayList<Order>();
    public void takeOrder(Order order) {
        orderList.add(order);
    }
    public void placeOrders() {
        for(Order order : orderList) {
            order.execute();
        }
        orderList.clear();
    }
}
```

解释器模式（Interpreter Pattern）

解释器模式提供了评估语言的语法或表达式的方式，它属于行为型模式。这种模式实现了一个表达式接口，该接口解释一个特定的上下文。这种模式被用在 SQL 解析、符号处理引擎等。

意图

给定一个语言，定义它的文法表示，并定义一个解释器，这个解释器使用该标识来解释语言中的句子。

主要解决

对于一些固定文法构建一个解释句子的解释器。

何时使用

如果一种特定类型的问题发生的频率足够高，那么可能就值得将该问题的各个实例表述为一个简单语言中的句子。这样就可以构建一个解释器，该解释器通过解释这些句子来解决该问题。

如何解决

构建语法树，定义终结符与非终结符。

关键代码

构件环境类，包含解释器之外的一些全局信息，一般是 HashMap。

应用实例

编译器、运算表达式计算。

优点

1、可扩展性比较好，灵活。
2、增加了新的解释表达式的方式。
3、易于实现简单文法。

缺点

1、可利用场景比较少。
2、对于复杂的文法比较难维护。
3、解释器模式会引起类膨胀。
4、解释器模式采用递归调用方法。

使用场景

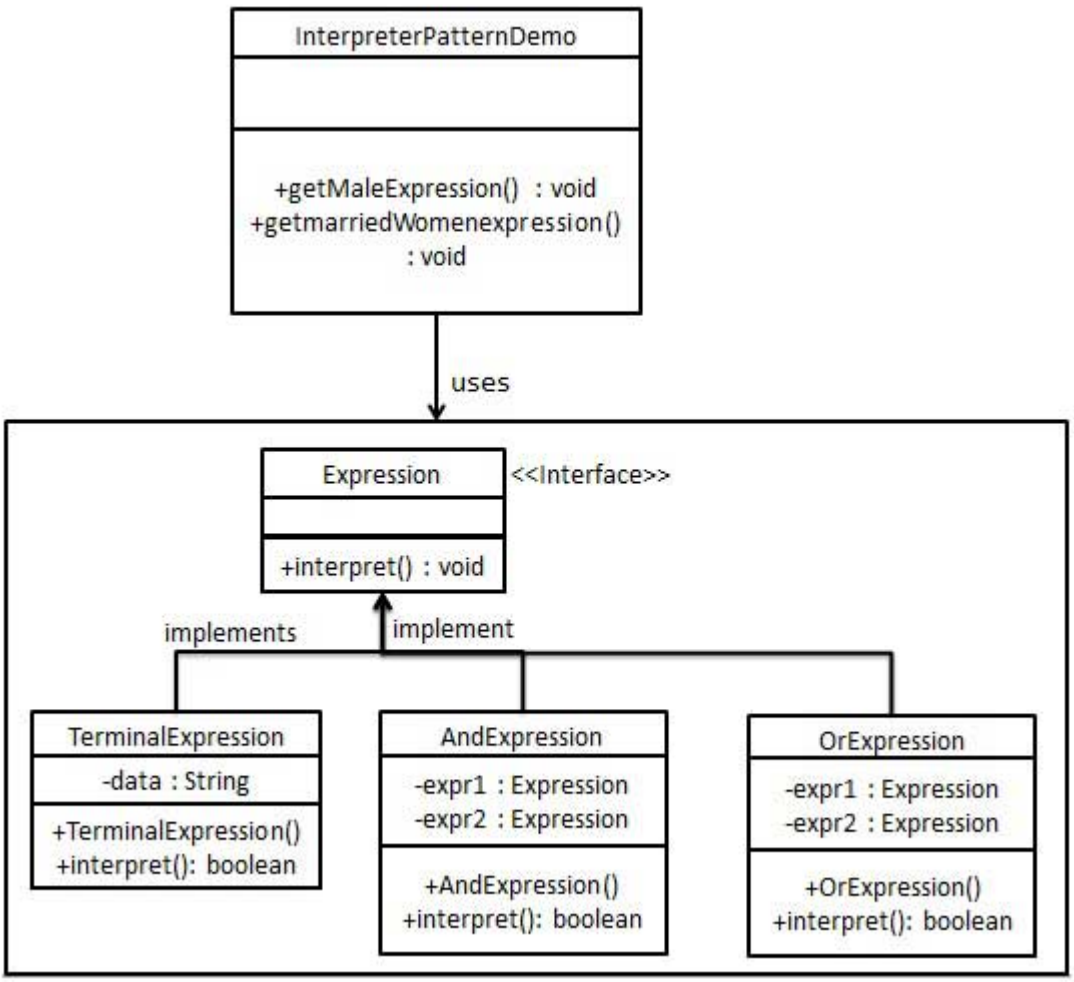
1、可以将一个需要解释执行的语言中的句子表示为一个抽象语法树。
2、一些重复出现的问题可以用一种简单的语言来进行表达。
3、一个简单语法需要解释的场景。

注意事项

可利用场景比较少，JAVA 中如果碰到可以用 expression4J 代替。

示例实现

我们将创建一个接口 Expression 和实现了 Expression 接口的实体类。定义作为上下文中主要解释器的 TerminalExpression 类。其他的类 OrExpression、AndExpression 用于创建组合式表达式。
我们的演示类使用 Expression 类创建规则和演示表达式的解析。



迭代器模式（Iterator Pattern）

是 Java 和 .Net 编程环境中非常常用的设计模式。这种模式用于顺序访问集合对象的元素，不需要知道集合对象的底层表示。
迭代器模式属于行为型模式。

意图

提供一种方法顺序访问一个聚合对象中各个元素, 而又无须暴露该对象的内部表示。

主要解决

不同的方式来遍历整个聚合对象。

何时使用

遍历一个聚合对象。

如何解决

把在元素之间游走的责任交给迭代器，而不是聚合对象。

关键代码

定义接口：hasNext, next。 应用实例：JAVA 中的 iterator。

优点

1、它支持以不同的方式遍历一个聚合对象。
2、迭代器简化了聚合类。
3、在同一个聚合上可以有多个遍历。
4、在迭代器模式中，增加新的聚合类和迭代器类都很方便，无须修改原有代码。

缺点

由于迭代器模式将存储数据和遍历数据的职责分离，增加新的聚合类需要对应增加新的迭代器类，类的个数成对增加，这在一定程度上增加了系统的复杂性。

使用场景

1、访问一个聚合对象的内容而无须暴露它的内部表示。
2、需要为聚合对象提供多种遍历方式。
3、为遍历不同的聚合结构提供一个统一的接口。

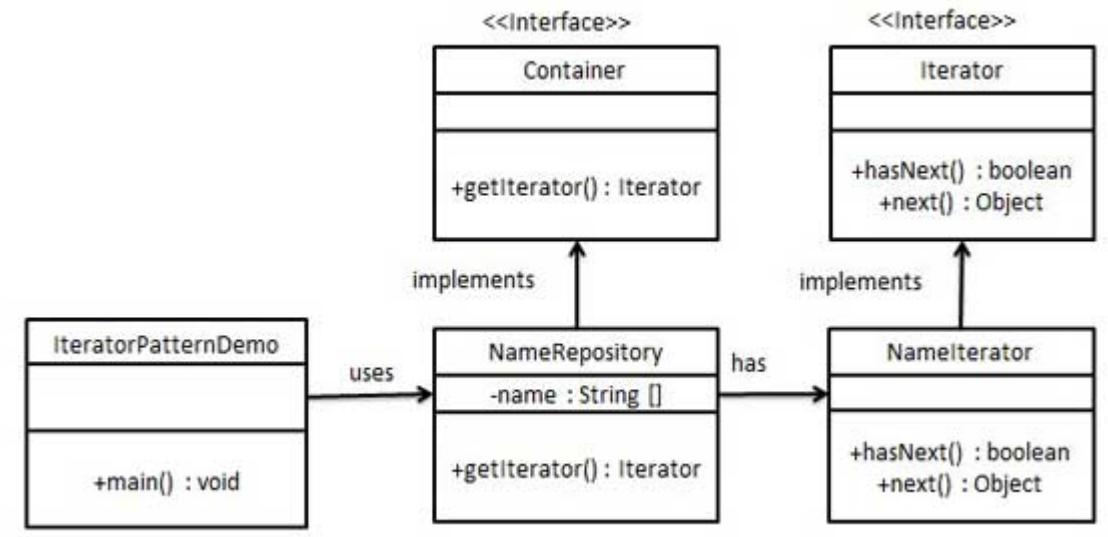
注意事项

迭代器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，这样既可以做到不暴露集合的内部结构，又可让外部代码透明地访问集合内部的数据。

示例实现

我们将创建一个叙述导航方法的 Iterator 接口和一个返回迭代器的 Container 接口。实现了 Container 接口的实体类将负责实现 Iterator 接口。

我们的演示类使用实体类 NamesRepository 来打印 NamesRepository 中存储为集合的 Names。



NameRepository的逻辑非常简单，就是遍历一个字符串数组：

```
public class NameRepository implements Container {
    public String names[] = {"Robert", "John", "Julie", "Lora"};
    @Override
    public Iterator getIterator() {
        return new NameIterator();
    }
    private class NameIterator implements Iterator {
        int index;
        @Override
        public boolean hasNext() {
            if(index < names.length) {
                return true;
            }
        }
        @Override
        public Object next() {
            if(this.hasNext()) {
                return names[index++];
            }
            return null;
        }
    }
}
```

中介者模式（Mediator Pattern）

中介者模式可以用来降低多个对象和类之间的通信复杂性。
中介者模式属于行为型模式。中介者模式**包装了一系列模块之间的相互作用方式，解耦了多个类之间的调用关系**，使它们不直接相互引用，而是以一种较松散的方式耦合。
在有依赖关系的模块之间添加一个中介，一个模块通过委托中介来实现对其他模块的调用。
当这些模块中的某些之间的相互作用发生改变时，不会立即影响到其他的一些之间的相互作用，从而保证这些相互作用可以彼此独立地变化。

为什么使用中介者模式

1. 设计阶段类间的依赖关系过于复杂。面向对象设计中倾向于将类的职责尽量单一化（SRP），这有可能导致系统内对象过多，职责与相互依赖关系变得复杂。
2. 如果类之间存在依赖关系，一个类修改其他关联的类也得随之修改，不利于功能的扩展和维护。
3. 使用中介者模式解耦，对象之间可以互不知道，只需关心和中介类的关联（DP），具体对象与对象之间的调度由中介类负责。

意图

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

主要解决

对象与对象之间存在大量的关联关系，这样势必会导致系统的结构变得很复杂，同时若一个对象发生改变，我们也需要跟踪与之相关联的对象，同时做出相应的处理。

何时使用

多个类相互耦合，形成了网状结构。

如何解决

将上述网状结构分离为星型结构。

关键代码

对象 Colleague 之间的通信封装到一个类中单独处理。

应用实例

1. 中国加入 WTO 之前是各个国家相互贸易，结构复杂，现在是各个国家通过 WTO 来互相贸易。
2. 机场调度系统。
3. MVC 框架，其中C（控制器）就是 M（模型）和 V（视图）的中介者。
4. Spring容器（ioc依赖注入）
   不是由A来主动地实例化一个B，而是由Spring容器来将对象赋值给调用者的成员变量。这样实现了通过Spring容器作为中介来管理对象之间的依赖。

优点

1、降低了类的复杂度，将一对多转化成了一对一。
2、各个类之间的解耦。
3、符合迪米特原则。

缺点

中介者会庞大，变得复杂难以维护。

使用场景

1、系统中对象之间存在比较复杂的引用关系，导致它们之间的依赖关系结构混乱而且难以复用该对象。
2、想通过一个中间类来封装多个类中的行为，而又不想生成太多的子类。

注意事项

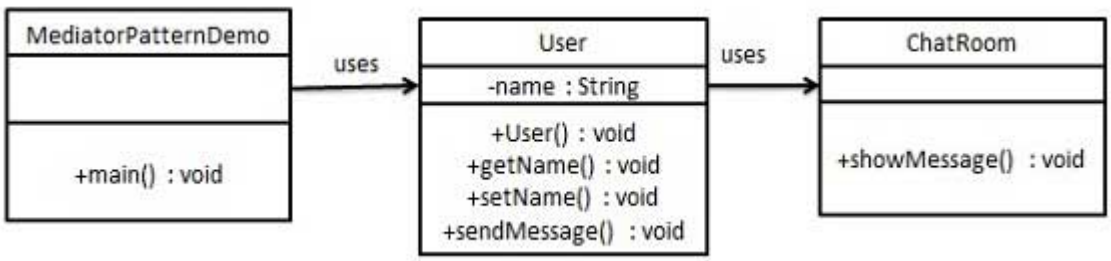
不应当在职责混乱的时候使用（中介者模式不能用于优化系统）。

和其他模式的区别

外观模式（facade） 使用一个外观对象控制对一组子系统的访问
委托给中间对象（外观）调用子系统，主要用于简化外界对系统的访问。
代理模式（proxy） 使用一个代理对象控制对一个对象的访问
由发起方通过中间对象（代理）调用接收方，主要用于控制访问或负载均衡。
观察者模式（observer） 当一个对象（被观察者/发布者）发生改变时，其他依赖该对象的对象（观察者/订阅者）都会改变
中间对象（被观察者）是主动方，由它发起对其它所有对象的修改，主要用于保持系统内对象的一致性，而不至于使得系统耦合过高。
中介者模式（mediator） 由中介者代为管理一组对象之间的访问
中间对象（中介者）是被动方，由另外依赖的对象来发起对其他对象的调用，然后交给中介处理，主要用于消除系统内复杂的依赖关系。

示例实现

我们通过聊天室实例来演示中介者模式。实例中，多个用户可以向聊天室发送消息，聊天室向所有的用户显示消息。我们将创建两个类 ChatRoom 和 User。User 对象使用 ChatRoom 方法来分享他们的消息。
我们的演示类使用 User 对象来显示他们之间的通信。



如上所示，客户端不会直接操作ChatRoom，而是调用User的发送消息接口：

```
public class User {
    private String name;
```

```
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public User(String name) {
        this.name = name;
    }
    public void sendMessage(String message) {
        ChatRoom.showMessage(this.message);
    }
}
```

设计

1. 忽略抽象中介
   系统较小时，忽略中介的扩展
2. 中介+观察者模式
   使用在这样的场景：一个成员对象发生状态变化时需要告知其他成员对象，这时中介者作为观察者，在成员对象发生状态变化后，由中介者将变化告知其他对象；
3. 多中介
   系统较复杂，每个中介负责一部分对象之间的通信，减少中介者的复杂性；
4. 通信接口
   设计通信接口时，考虑通信需要识别发送方和接收方，我们可以将发送方或者接收方作为参数传递进对应send方法；

问题

1. 如果系统已经很大很复杂了，可以使用中介者模式优化吗?
   中介者模式不能用于优化一个依赖混乱的系统，它不能代替人对系统进行责任划分，所以系统设计阶段的工作要做的充分。
2. 依赖很复杂，中介类变得很大
   考虑界面组件慢慢添加到组件库中，它们之间的通信使用一个中介类处理，这些组件间的交互势必使得中介类变得累赘，所以要做好系统的模块划分。

备忘录模式（Memento Pattern）

备忘录模式保存一个对象的某个状态，以便在适当的时候恢复对象。备忘录模式属于行为型模式。

意图

在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态。

主要解决

所谓备忘录模式就是在不破坏封装的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态，这样可以在以后将对象恢复到原先保存的状态。

何时使用

很多时候我们总是需要记录一个对象的内部状态，这样做的目的就是为了允许用户取消不确定或者错误的操作，能够恢复到他原先的状态，使得他有"后悔药"可吃。

如何解决

通过一个备忘录类专门存储对象状态。

关键代码

客户不与备忘录类耦合，与备忘录管理类耦合。

应用实例

1、后悔药。
2、打游戏时的存档。
3、Windows 里的 ctri + z。
4、IE 中的后退。
5、数据库的事务管理。

优点

1、给用户提供了一种可以恢复状态的机制，可以使用户能够比较方便地回到某个历史的状态。
2、实现了信息的封装，使得用户不需要关心状态的保存细节。

缺点

消耗资源。如果类的成员变量过多，势必会占用比较大的资源，而且每一次保存都会消耗一定的内存。

使用场景

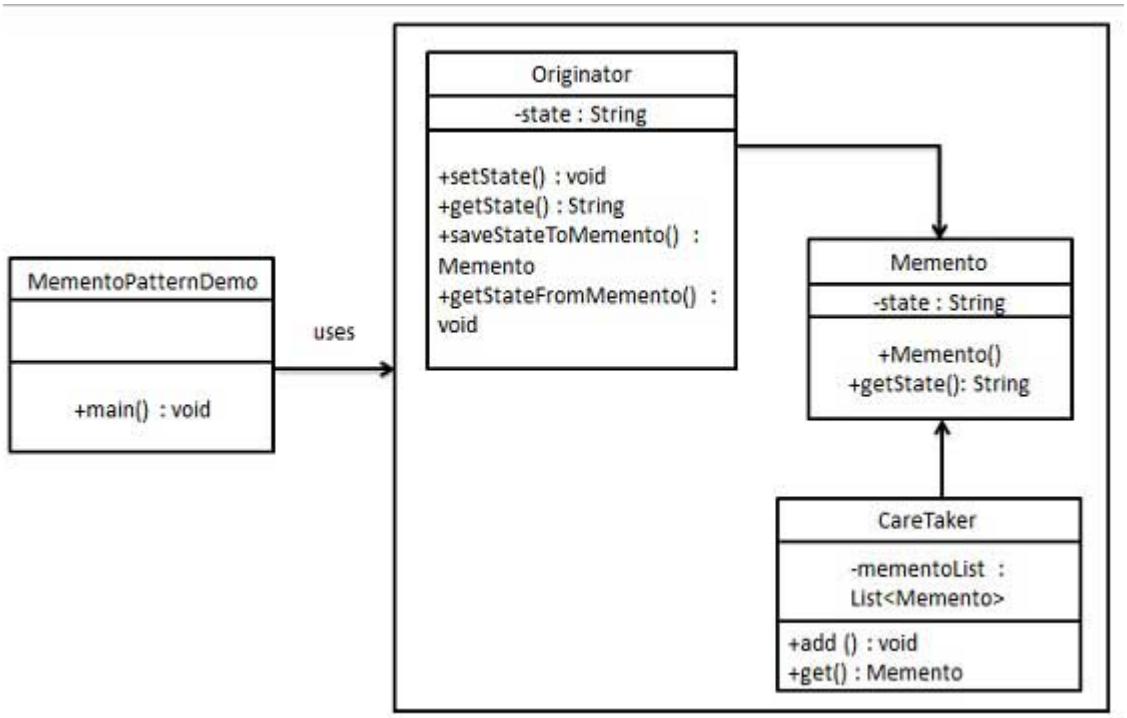
1、需要保存/恢复数据的相关状态场景。
2、提供一个可回滚的操作。

注意事项

1、为了符合迪米特原则，还要增加一个管理备忘录的类。
2、为了节约内存，可使用原型模式+备忘录模式。

示例实现

备忘录模式使用三个类 Memento、Originator 和 CareTaker。Memento 包含了要被恢复的对象的状态。Originator 创建并在 Memento 对象中存储状态。Caretaker 对象负责从 Memento 中恢复对象的状态。MementoPatternDemo，我们的演示类使用 CareTaker 和 Originator 对象来显示对象的状态恢复。



这里CareTaker使用一个List来“持久化”Memento对象：

```
public class CareTaker {
    private List<Memento> mementoList = new ArrayList<Memento>();
    public void add(Memento state) {
        mementoList.add(state);
    }
    public Memento get(int index) {
        return mementoList.get(index);
    }
}
```

观察者模式（Observer Pattern）

当对象间存在一对多关系时，则使用观察者模式。比如，当一个对象被修改时，则会自动通知它的依赖对象。观察者模式属于行为型模式。

意图

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

主要解决

一个对象状态改变给其他对象通知的问题，而且要考虑到易用和低耦合，保证高度的协作。

何时使用

一个对象（目标对象）的状态发生改变，所有的依赖对象（观察者对象）都将得到通知，进行广播通知。

如何解决

使用面向对象技术，可以将这种依赖关系弱化。

关键代码

在抽象类里有一个 ArrayList 存放观察者们。

应用实例

1、拍卖的时候，拍卖师观察最高标价，然后通知给其他竞价者竞价。
2、西游记里面悟空请求菩萨降服红孩儿，菩萨洒了一地水招来一个老乌龟，这个乌龟就是观察者，他观察菩萨洒水这个动作。

优点

1、观察者和被观察者是抽象耦合的。
2、建立一套触发机制。

缺点

1、如果一个被观察者对象有很多的直接和间接的观察者的话，将所有的观察者都通知到会花费很多时间。
2、如果在观察者和观察目标之间有循环依赖的话，观察目标会触发它们之间进行循环调用，可能导致系统崩溃。
3、观察者模式没有相应的机制让观察者知道所观察的目标对象是怎么发生变化的，而仅仅只是知道观察目标发生了变化。

使用场景

1、有多个子类共有的方法，且逻辑相同。
2、重要的、复杂的方法，可以考虑作为模板方法。

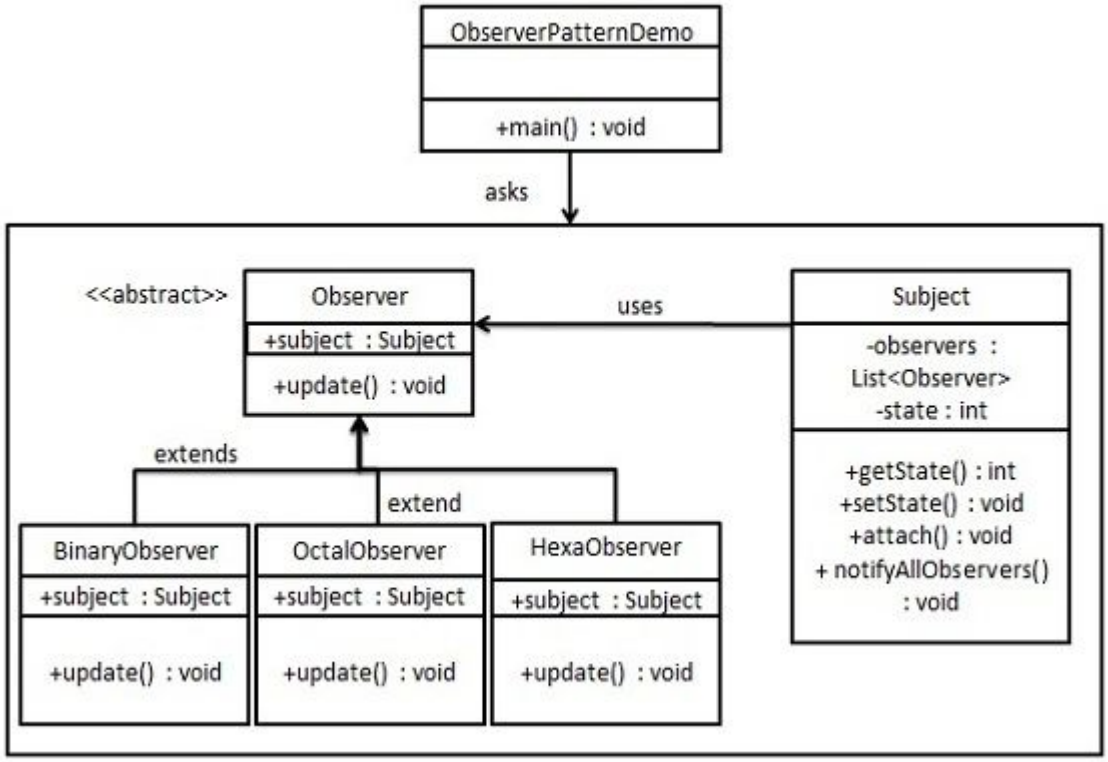
注意事项

1. JAVA 中已经有了对观察者模式的支持类（Observer和Observable）。
2. 避免循环引用。
3. 如果顺序执行，某一观察者错误会导致系统卡壳，一般采用异步方式。
4. 观察者并非一成不变。
   并不是每天都是所有人回家吃午饭，只有提前告知老妈要回家吃午饭的人才会接到电话。因为观察者的可变性，需要被观察者维护一个列表。
5. 观察者模式可以便捷的完成目标。
   不需要观察者不停的轮询查看事件变化，也不需要被观察者多次询问观察者意愿。只需要观察者提前加入或离开列表，便可以由被观察者准确的进行事件通知。 说起来，微信订阅号便是观察者模式的一种实现。感兴趣的人订阅公众号，在公众号有新的文章时推送给所有订阅人。

示例实现

观察者模式使用三个类 Subject、Observer 和 Client。Subject 对象带有绑定观察者到 Client 对象和从 Client 对象解绑观察者的方法。我们创建 Subject 类、Observer 抽象类和扩展了抽象类 Observer 的实体类。

我们的演示类使用 Subject 和实体类对象来演示观察者模式。



如上所述，被观察者（Subject）需要使用一个列表来保存观察者：

```
public class Subject {
    private List<Observer> observers = new ArrayList<Observer>();
    private int state;
    public int getState() {
        return state;
    }
    // 状态改变后需要通知所有观察者
    public void setState(int state) {
        this.state = state;
        notifyAllObservers();
    }
    public void attach(Observer observer) {
        observers.add(observer);
    }
    public void notifyAllObservers() {
        for(Observer observer : observers) {
            observer.update();
        }
    }
}
```

应用实例

需求：启动 N 个线程执行不同的或并发的任务，当服务接收到停止暂停、启动或停止命令时，任务状态变更，需要进行所有任务的调度。
被观察者：任务状态
事件：任务状态变更命令
观察者：多线程
实现要点：

1. 定义Event接口及其实现类：

```
public interface Event {
}
public class TaskStatusEvent implements Event{
    private Type type;
    private String taskId;
    public TaskStatusEvent(String taskId,Type type){
        this.taskId=taskId;
        this.type=type;
    }
    public Type getType() {
        return type;
    }
    public String getTaskId(){
        return this.taskId;
    }
    public static enum Type{
        OK,SYNTIME,TASK_START,TASK_STOP,VU_PLUS,VU_MINUS
    }
}
```

1. 被观察者是每个实现了 Watcher 接口的 runnable 实例：

```
public interface Watcher {
    public void onDoSomeThing(Event event);
}
public class TaskProcess implements Runnable, Watcher {
    @Override
    public void onDoSomeThing(Event event) {
        TaskStatusEvent taskStatusEvent = (TaskStatusEvent) event;
        switch(taskStatusEvent.getType){
            case "TASK_STOP":...break;
            case "TASK_START":....break;
        }
```

```
    }
}
```

1. 观察者

```
public interface Subject {
    public void addWatcher(Watcher watcher);
    public void doSomeThing();
}
public class MultiThreadSubject implements Subject {
    private static ArrayList<TaskProcess> watchers = new ArrayList<>();
    TaskStatusEvent event = null;
    public TaskStatusEvent getEvent() {
        return event;
    }
    public void setEvent(TaskStatusEvent event) {
        this.event = event;
    }
    @Override
    public void addWatcher(TaskProcess watcher) {
        watchers.add(watcher);
    }
    @Override
    public void doSomeThing() {
        // TODO: 当服务接收到停止暂停、启动或停止命令时，任务状态变更，需要进行所有任务的调度

    }
}
```

状态模式（State Pattern）

在状态模式中，类的行为是基于它的状态改变的。这种类型的设计模式属于行为型模式。
在状态模式中，我们创建表示各种状态的对象和一个行为随着状态对象改变而改变的 context 对象。

意图

一个对象在内部状态改变时可以改变行为，并且其必须在**运行时**根据状态实时改变其行为，对象看起来好像修改了它的类。

主要解决

对象的行为依赖于它的状态（属性），并且可以根据它的状态改变而改变它的相关行为。

何时使用

代码中包含大量与对象状态有关的条件语句。

如何解决

将各种具体的状态类抽象出来。

关键代码

通常命令模式的接口中只有一个方法。而状态模式的接口中有一个或者多个方法。而且，状态模式的实现类的方法，一般返回值， 或者是改变实例变量的值。也就是说，状态模式一般和对象的状态有关。实现类的方法有不同的功能，覆盖接口中的方法。状态模式和命令模式一样，也可以用于消 除 if...else 等条件选择语句。

应用实例

1、打篮球的时候运动员可以有正常状态、不正常状态和超常状态。
2、曾侯乙编钟中，'钟是抽象接口','钟A'等是具体状态，'曾侯乙编钟'是具体环境（Context）。
3、浏览器可以根据当前的网络情况来决定停止加载一部分图片。

优点

1、封装了转换规则。
2、枚举可能的状态，在枚举状态之前需要确定状态种类。
3、将所有与某个状态有关的行为放到一个类中，并且可以方便地增加新的状态，只需要改变对象状态即可改变对象的行为。
4、允许状态转换逻辑与状态对象合成一体，而不是某一个巨大的条件语句块。
5、可以让多个环境对象共享一个状态对象，从而减少系统中对象的个数。

缺点

1、状态模式的使用必然会增加系统类和对象的个数。
2、状态模式的结构与实现都较为复杂，如果使用不当将导致程序结构和代码的混乱。
3、状态模式对"开闭原则"的支持并不太好，对于可以切换状态的状态模式，增加新的状态类需要修改那些负责状态转换的源代码，否则无法切换到新增状态，而且修改某个状态类的行为也需修改对应类的源代码。

使用场景

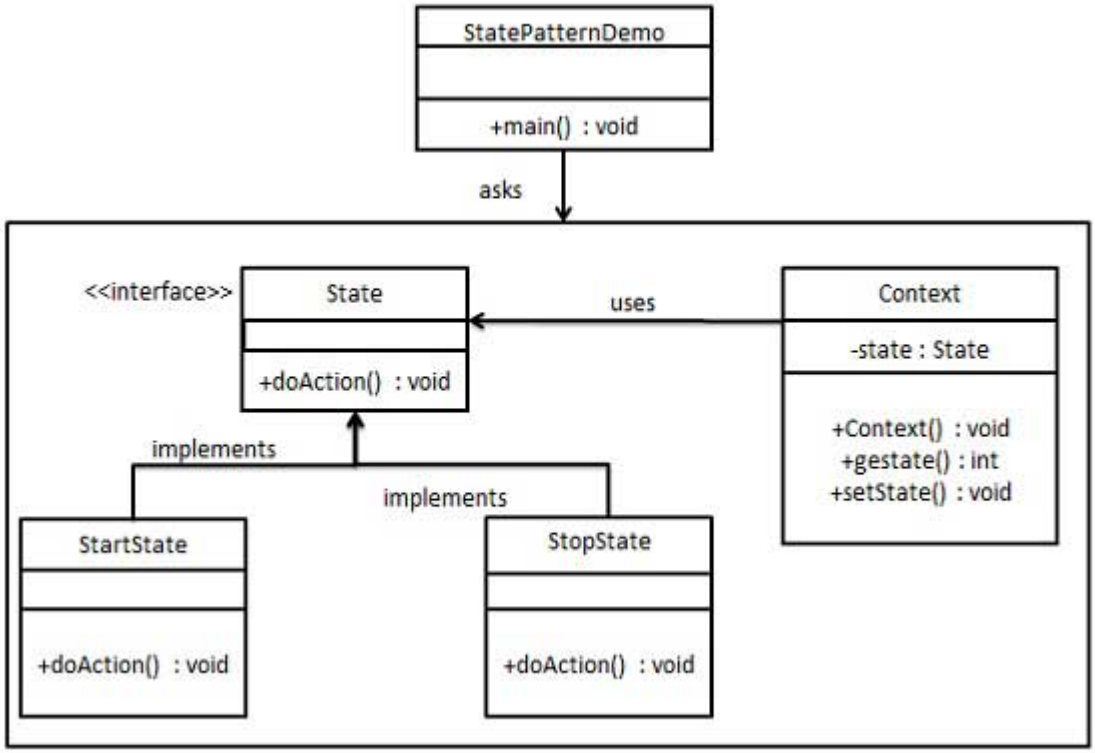
1、行为在**运行时刻**随状态改变而改变的场景。
2、作为条件、分支语句的代替者。

注意事项

在行为受状态约束的时候使用状态模式，而且状态不超过 5 个。

示例实现

我们将创建一个 State 接口和实现了 State 接口的实体状态类。Context 是一个带有某个状态的类。
我们的演示类使用 Context 和状态对象来演示 Context 在状态改变时的行为变化。



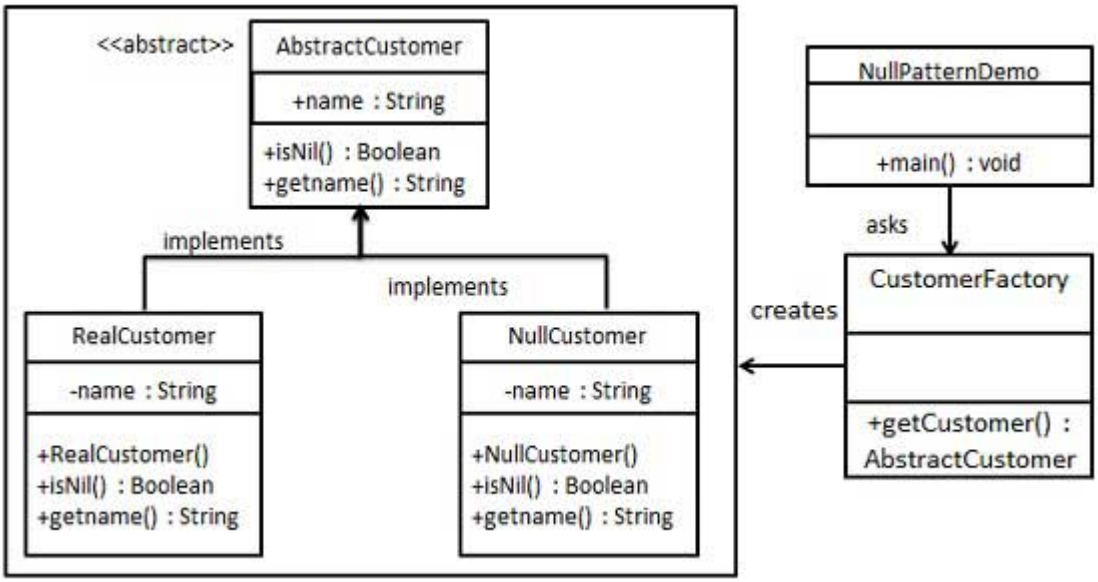
当Context获取到新状态后它对外会表现为新状态的行为，在新状态下运行的途中由可能会获得新的状态，从而形成状态链。

空对象模式（Null Object Pattern）

在空对象模式中，一个空对象取代 NULL 对象实例的检查。Null 对象不是检查空值，而是反应一个不做任何动作的关系。这样的 Null 对象也可以在数据不可用的时候提供默认的行为。
在空对象模式中，我们创建一个指定各种要执行的操作的抽象类和扩展该类的实体类，还创建一个未对该类做任何实现的空对象类，该空对象类将无缝地使用在需要检查空值的地方。

示例实现

我们将创建一个定义操作（在这里，是客户的名称）的 AbstractCustomer 抽象类，和扩展了 AbstractCustomer 类的实体类。工厂类 CustomerFactory 基于客户传递的名字来返回 RealCustomer 或 NullCustomer 对象。
我们的演示类使用 CustomerFactory 来演示空对象模式的用法。



NullCustomer提供对AbstractCustomer的默认实现，特别是对空对象检查方法isNil()的实现：

```
public class NullCustomer extends AbstractCustomer {
    @Override
    public String getName() {
        return "Not Available in Customer Database";
    }
    @Override
    public boolean isNil() {
        return true;
    }
}
```

CustomerFactory在实例化顾客实例的时候可以判断若该顾客不存在则直接返回一个空对象：

```
public class CustomerFactory {
    public static final String[] names = {"Rob", "Joe", "Julie"};
    public static AbstractCustomer getCustomer(String name) {
        for(int i = 0; i < names.length; i++) {
            if(names[i].equalsIgnoreCase(name)) {
                return new RealCustomer(name);
            }
        }
        return new NullCustomer();
    }
}
```

策略模式（Strategy Pattern）

在策略模式中，一个类的行为或其算法可以在运行时更改。这种类型的设计模式属于行为型模式。
在策略模式中，我们创建表示各种策略的对象和一个行为随着策略对象改变而改变的 context 对象。策略对象改变 context 对象的执行算法。

意图

定义一系列的算法,把它们一个个封装起来, 并且使它们可相互替换。

主要解决

在有多种算法相似的情况下，使用 if...else 所带来的复杂和难以维护。

何时使用

一个系统有许多许多类，而区分它们的只是他们直接的行为。

如何解决

将这些算法封装成一个一个的类，任意地替换。

关键代码

实现同一个接口。

应用实例

1、诸葛亮的锦囊妙计，每一个锦囊就是一个策略。
2、旅行的出游方式，选择骑自行车、坐汽车，每一种旅行方式都是一个策略。
3、JAVA AWT 中的 LayoutManager。

优点

1、算法可以自由切换。
2、避免使用多重条件判断。
3、扩展性良好。

缺点

1、策略类会增多。
2、所有策略类都需要对外暴露。

使用场景

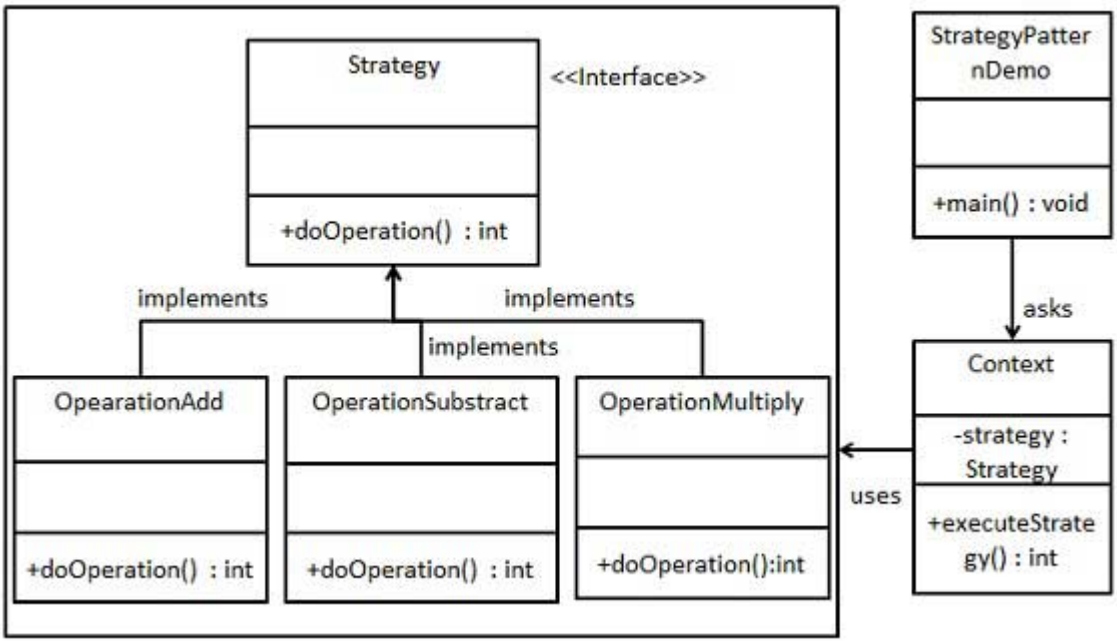
1、如果在一个系统里面有许多类，它们之间的区别仅在于它们的行为，那么使用策略模式可以动态地让一个对象在许多行为中选择一种行为。
2、一个系统需要动态地在几种算法中选择一种。
3、如果一个对象有很多的行为，如果不用恰当的模式，这些行为就只好使用多重的条件选择语句来实现。

注意事项

如果一个系统的策略多于四个，就需要考虑使用混合模式，解决策略类膨胀的问题。

示例实现

我们将创建一个定义活动的 Strategy 接口和实现了 Strategy 接口的实体策略类。Context 是一个使用了某种策略的类。
StrategyPatternDemo，我们的演示类使用 Context 和策略对象来演示 Context 在它所配置或使用的策略改变时的行为变化。



由用户指定Context聚合的Strategy：

```
public class Context {
    private Strategy strategy;
    public Context(Strategy strategy) {
        this.strategy = strategy;
    }
    public int executeStrategy(int num1, int num2) {
        return strategy.doOperation(num1, num2);
    }
}
```

相似模式

策略模式的类图与状态模式的非常相似，但是它们有着很大的不同：

1. 意图不同，策略模式封装一系列平行且复杂多变的实现方式，而状态模式实现把对象的内在状态的变化封装起来，用外部行为来表现出来；
2. 决定哪一个具体实现类的主体不同，策略模式是由客户端指定的，状态模式是由Context指定的；

模板模式（Template Pattern）

在模板模式中，一个抽象类公开定义了执行它的方法的方式/模板。它的子类可以按需要重写方法实现，但调用将以抽象类中定义的方式进行。这种类型的设计模式属于行为型模式。

意图

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

主要解决

一些方法通用，却在每一个子类都重新写了这一方法。

何时使用

有一些通用的方法。

如何解决

将这些通用算法抽象出来。

关键代码

在抽象类实现，其他步骤在子类实现。

应用实例

1、在造房子的时候，地基、走线、水管都一样，只有在建筑的后期才有加壁橱加栅栏等差异。
2、西游记里面菩萨定好的 81 难，这就是一个顶层的逻辑骨架。
3、Spirng 中对 Hibernate 的支持，将一些已经定好的方法封装起来，比如开启事务、获取 Session、关闭 Session 等，程序员不重复写那些已经规范好的代码，直接丢一个实体就可以保存。

优点

1、封装不变部分，扩展可变部分。
2、提取公共代码，便于维护。
3、行为由父类控制，子类实现。

缺点

每一个不同的实现都需要一个子类来实现，导致类的个数增加，使得系统更加庞大。

使用场景

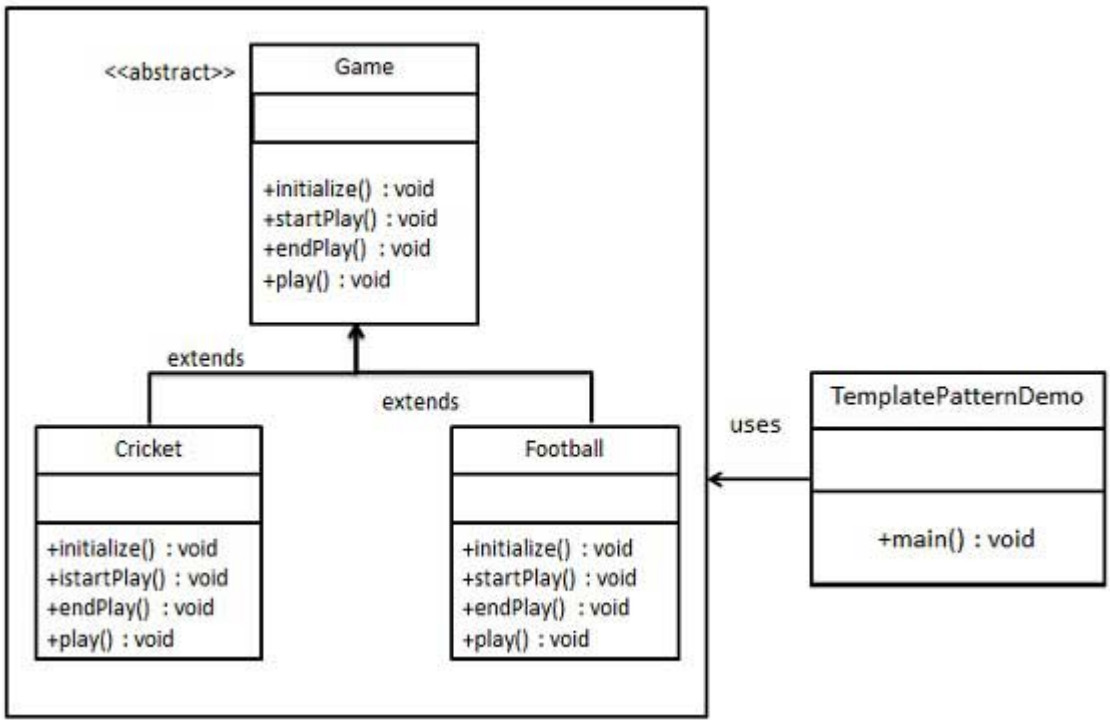
1、有多个子类共有的方法，且逻辑相同。
2、重要的、复杂的方法，可以考虑作为模板方法。

注意事项

为防止恶意操作，一般模板方法都加上 final 关键词。

示例实现

我们将创建一个定义操作的 Game 抽象类，其中，模板方法设置为 final，这样它就不会被重写。Cricket 和 Football 是扩展了 Game 的实体类，它们重写了抽象类的方法。
我们的演示类使用 Game 来演示模板模式的用法。



注意对play()方法使用final修饰，play()中调用到了initialize()、startPlay()、endPlay()这三个方法，具体代码就不给出了。

访问者模式（Visitor Pattern）

在访问者模式中，我们使用了一个访问者类，它改变了元素类的执行算法。通过这种方式，元素的执行算法可以随着访问者改变而改变。这种类型的设计模式属于行为型模式。根据模式，元素对象已接受访问者对象，这样访问者对象就可以处理元素对象上的操作。

意图

主要将数据结构与数据操作分离，可以对某个容器中各元素定义不同的操作，而不需要改变该元素的类。。

主要解决

稳定的数据结构和易变的操作耦合问题。

何时使用

需要对一个对象结构中的对象进行很多不同的并且不相关的操作，而需要避免让这些操作"污染"这些对象的类，使用访问者模式将这些封装到类中。

如何解决

在被访问的类里面加一个对外提供接待访问者的接口。

关键代码

在数据基础类里面有一个方法接受访问者，将自身引用传入访问者。

应用实例

您在朋友家做客，您是访问者，朋友接受您的访问，您通过朋友的描述，然后对朋友的描述做出一个判断，这就是访问者模式。

优点

1、符合单一职责原则。
2、优秀的扩展性。
3、灵活性。

缺点

1、具体元素对访问者公布细节，违反了迪米特原则。
2、具体元素变更比较困难。
3、违反了依赖倒置原则，依赖了具体类，没有依赖抽象。

使用场景

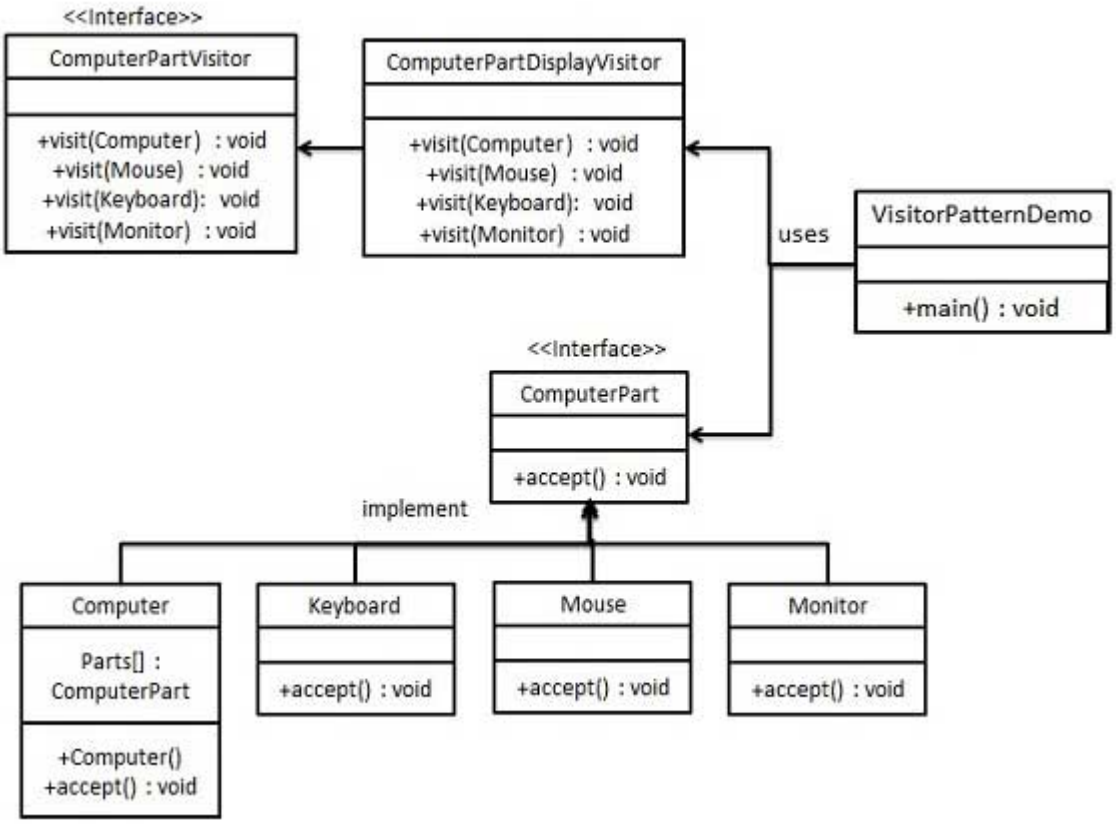
1、对象结构中对象对应的类很少改变，但经常需要在此对象结构上定义新的操作。
2、需要对一个对象结构中的对象进行很多不同的并且不相关的操作，而需要避免让这些操作"污染"这些对象的类，也不希望在增加新操作时修改这些类。

注意事项

访问者可以对功能进行统一，可以做报表、UI、拦截器与过滤器。

示例实现

我们将创建一个定义接受操作的 ComputerPart 接口。Keyboard、Mouse、Monitor 和 Computer 是实现了 ComputerPart 接口的实体类。我们将定义另一个接口 ComputerPartVisitor，它定义了访问者类的操作。Computer 使用实体访问者来执行相应的动作。
我们的演示类使用 Computer、ComputerPartVisitor 类来演示访问者模式的用法。



ComputerPart提供了一个接口accept(ComputerPartVisitor)来接受访问者的操作：

```
public class Keyboard implements ComputerPart {
    @Override
    public void accept(ComputerPartVisitor computerPartVisitor) {
        computerPartVisitor.visit(this);
    }
}
...
public class Computer implements ComputerPart {
    ComputerPart[] parts;
    public Computer() {
        parts = new ComputerPart[] {new Mouse(), new Keyboard(), new Monitor()};
    }
    @Override
    public void accept(ComputerPartVisitor computerPartVisitor) {
        for(int i = 0; i < parts.length; i++) {
            parts[i].accept(computerPartVisitor);
        }
        computerPartVisitor.visit(this);
    }
}
```

ComputerPartDisplayVisitor分别实现对四种ComputerPart的访问操作。