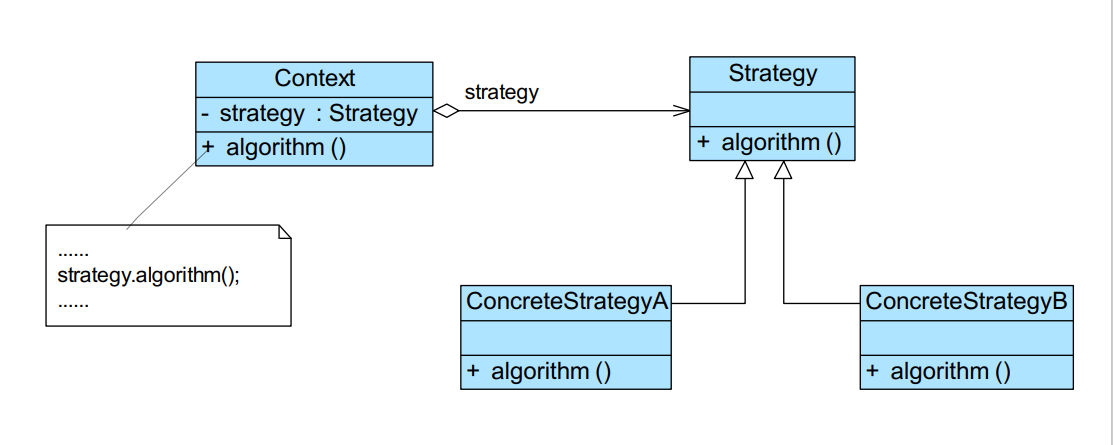
本次作业部分参考了霍静老师本学期“面向对象设计方法”课程的内容。

选择GoF设计模式中的策略模式。该模式定义一系列算法，将每一个算法封装起来，并让它们可以相互替换。这样一来，算法就可以独立于使用它的客户变化了。

策略模式的一般结构如下图，其中包含3个角色：Context（环境类）、Strategy（抽象策略类）和ConcreteStrategy（具体策略类）。典型的环境类中往往维持一个对抽象策略类的引用，可以由外界注入具体的策略对象，并在自己的方法中调用具体策略类（可在运行时指定类型，比如在Java中通过配置文件和反射机制实现）的算法。



该模式的优点是提供了对开闭原则的完美支持，用户可以不修改原有系统而选择算法或行为，也可以灵活地增加新的算法或行为。这样避免了多重条件选择语句，是一种方便的算法复用机制，用户不需要知道与算法相关的内部细节，还能提高算法的保密性与安全性。缺点在于客户端必须知道所有的策略类，自行决定使用哪一个，而且无法同时使用多个。适用环境是系统需要动态地在几种算法中选择一种，避免使用难以维护的多重条件选择语句。

现假设某软件公司为某电影院开发了一套影院售票系统，在该系统中需要为不同类型的用户提供不同的电影票打折方式，具体打折方案如下：

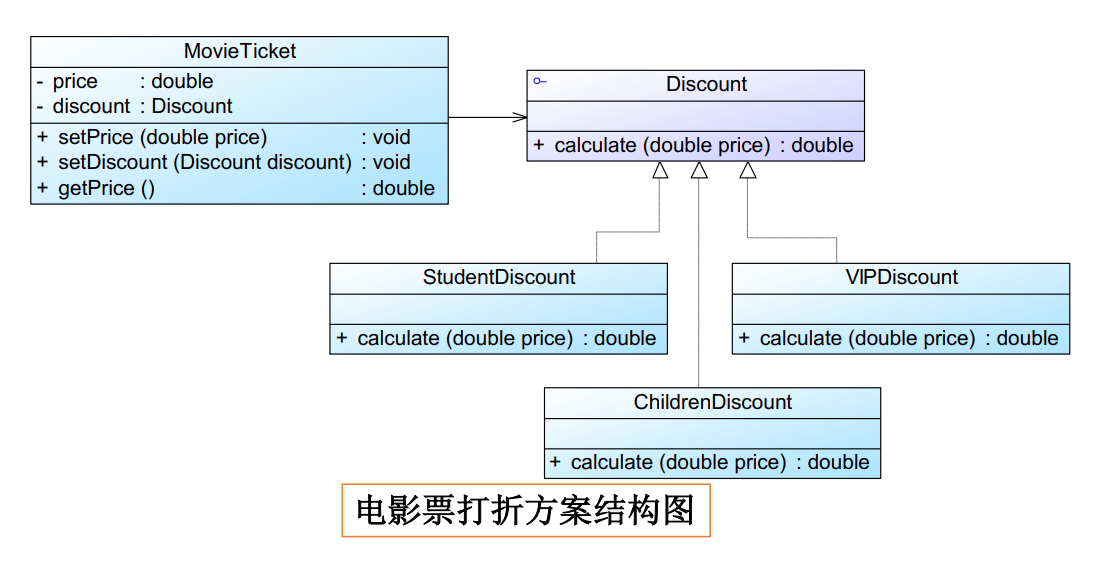
(1) 学生凭学生证可享受票价8折优惠。

(2) 10周岁及以下儿童每张票减免10元的（原始票价需大于等于20元）。

(3) VIP用户除享受半价优惠外还可积分，累计到一定额度换取奖品。

该系统在将来可能还要根据需要引入新的打折方式。

使用策略模式设计该影院售票系统的打折方案。首先考虑面向对象设计方法：



interface Discount {

    calculate(price : number) : number;

}

class ChildrenDiscount implements Discount {

    private discount : number = 10;

    public calculate(price : number) : number {

        console.log("儿童票：");

        if (price >= 20)

            return price - this.discount;

        else

            return price;

    }

}

class StudentDiscount implements Discount {

    private discount : number = 0.8;

    public calculate(price : number) : number {

        console.log("学生票：");

        return price \* this.discount;

    }

}

class VIPDiscount implements Discount {

    private discount : number = 0.5;

    public calculate(price : number) : number {

        console.log("VIP票：增加积分！");

        return price \* this.discount;

    }

}

class MovieTicket {

    private price : number;

    private discount : Discount;

    constructor(price : number, discount : Discount) {

        this.price = price;

        this.discount = discount;

    }

    public getPrice() : number {

        return this.discount.calculate(this.price);

    }

}

function main() {

    const originalPrice : number = 60.0;

    const discount : Discount = new StudentDiscount();

    const mt : MovieTicket = new MovieTicket(originalPrice, discount);

    console.log("原始价为：" + originalPrice);

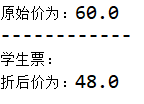
    console.log("------------");

    const currentPrice = mt.getPrice();

    console.log("折后价为：" + currentPrice);

}

测试代码中设置原始票价为60元，折扣计算方法为学生票，则输出结果为：



创建MovieTicket类时传入原始票价和Discount对象，调用getPrice方法即可对不同票价的电影采用不同的折扣计算方法来计算折后价。如果需要更换打折方式，只需在用户代码中更改注入MovieTicket的Discount对象（可使用反射机制），如果需要增加新的打折方式，则为Discount添加新的子类即可，不需要改动现有业务代码，符合开闭原则。

考虑用函数式写法对其进行改造。由于特意选择了策略模式，即，该模式本身的目的就是定义一系列可以相互替换的算法，抽象策略类实际上提供了一类算法的模板接口，由具体策略类对它进行继承，实现不同的具体算法。可以看到每个策略类都只有一个成员函数，就是实现这个算法的函数。那么很容易想到，在支持函数式的语言中，可以把这个函数单独拿出来作为“策略”注入环境类，就不需要真的把策略定义成类，而且创建真的对象了。如果在策略类中定义了一些成员变量，目的是供算法使用，改成函数内部的局部变量即可。此外，因为这个例子里的MovieTicket类也只有getPrice一个方法接口，不妨也直接改成函数，当然如果环境类除了要用到策略进行计算之外，还有很多其他的方法，就不方便这样修改了。

type discount = (price : number) => number;

function childrenDiscount (price : number) : number {

    const discount : number = 10;

    console.log("儿童票：");

    if (price >= 20)

        return price - discount;

    else

        return price;

}

function studentDiscount (price : number) : number {

    const discount : number = 0.8;

    console.log("学生票：");

    return price \* discount;

}

function vipDiscount (price : number) : number {

    const discount : number = 0.5;

    console.log("VIP票：增加积分！");

    return price \* discount;

}

function movieTicket (price: number, discount : discount){

    return discount(price);

}

function main() {

    const originalPrice : number = 60.0;

    console.log("原始价为：" + originalPrice);

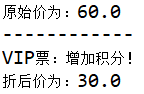
    console.log("------------");

    const currentPrice = movieTicket(originalPrice, vipDiscount);

    console.log("折后价为：" + currentPrice);

}

测试代码中设置原始票价为60元，折扣计算方法为VIP票，则输出结果为：



调用movieTicket函数时传入不同的原始票价和discount函数，即可对不同票价的电影采用不同的折扣计算折后价。更换打折方式只需在用户代码中更改调用movieTicket传入的discount参数（也可使用反射机制），增加新的打折方式则添加类型为discount的函数即可，同样不需要改动现有业务代码，同样符合开闭原则，支持复用和扩展。

两者的区别主要在于函数式写法中的折扣方法和票价计算都是函数，可以直接使用，不需要创建任何对象，而面向对象写法为了调用某类的成员函数总是要创建和维持它的对象。可以理解为建模粒度和抽象聚焦点不同，函数式写法聚焦于票价计算的处理逻辑，将它拆成“已知基础票价根据一定策略返回折后价”和“已知基础票价和策略计算折后价”两个层次，都是注重输入、输出及转化过程的方法；面向对象写法则更多将电影票和策略视为有各种属性和方法的对象，如折扣力度是折扣策略的属性，原始票价和折扣策略又是电影票属性，类和对象基于这些属性提供方法接口与外界交互。总的来说，这个例子改写成函数式写法之所以合适，是因为它本身就关注策略方法，而且方法比较单一，一个接口仅声明了一个方法，完全可以用函数类型来替代，使得实现更加简洁。如果一个接口声明多个方法，需要处理多个函数，则面向对象提供的继承、组合（聚合）特性更实用。比如，假设MovieTicket类除getPrice还有sell、refund、chooseSeat等方法，就不宜改成函数；VIPDiscount类假如除了实现父类接口的calculate方法，还提供savePoint为VIP用户计算积分，改成函数时需要把这些方法单独拆开，也会造成程序的冗杂和混乱，增大管理难度。