IN2013, Week 5 – Design Patterns

Model Answers

Dr Peter T. Popov

25th October 2018

Scenario

Consider an extension of the BAPPERS case study as follows. Some of the regular customers may be given a discount from their orders (Jobs). BIPL have decided on two discount plans:

- *Fixed* discount plan. The customer gets a discount as a % of the order, e.g. 3-5%. The particular rate of discount may vary between different customers and is decided by the Office Manager.
- *Flexible* discount plan. Each task from the list of predefined Tasks (TaskDescriptions) may be given a rate of discount, typically in the range of 5-10%. The discount rates may differ between tasks. Some tasks may not NOT be given a discount. The rates of discount per task may also differ between customers. Consider the following two possibilities:
    - A "default" flexible discount plan, which defines a "global" flexible discount plan. This plan can be shared by many customers (:CustomerAccount[1] objects).
    - A customized flexible discount plan, which is uniquely defined for a customer (:CustomerAccount).

A refined class diagram is provided in the Appendix, in which the options for a discount plan are modelled (see the classes in the red box on the right). The extension of the model is also available as a VP project.

Question 1. Discuss the provided model and answer the following questions:

- Does the proposed model allow for discount plans to be shared between customers (i.e. multiple customers to use the same discount plan, i.e. to save the plan once in the system)?
- How easy is it to change the discount plan of a customer from say – no discount plan to a fixed discount plan of say from a 5% fixed discount to a 10% fixed discount plan? What about changing from a fixed to a flexible discount?
- Compare how the functionality of classes before the introduction of discounts (e.g. the model developed in Week3/4) will be affected. Will we need to change some of the classes and if so – how.
- In particular compare the impact of introducing discounts on the calculation of price for a *:Job*.

Model answer

The two discount plans have a composition relationship with CustomerAccount, which is exclusive (i.e. an object :CustormeAccount will own exclusively an object :FixedDiscountPlan and :FlexibleDiscountPlan), hence no sharing of discount plans between :CustomerAccount objects is possible with the model as is.

Changing compositions to aggregations, in principle, will allow for sharing the same discount plan among multiple :CustomerAccount objects.

[1] A:Class is the standard UML notation for objects, which refers to an object A of type Class.

Note also, the XOR constraint between the two composition relationships. The idea here is to show that we want to rule out the possibility of having two discount plans links to a particular :CustomerAccount. In other words, :CustomerAccount will always have a single link to either an :FixedDiscountPlan or an :FlexibleDiscountPlan, but never two separate links.

Changing the discount plan from say fixed to flexible is relatively easy: such a change will require us to remove the link between :CustomerAccount to a :FixedDiscountPlan and add a link to :FixedDiscountPlan.

Advanced stuff:

Note that in the model I use association roles – "discount" and "flexiDiscount"in the association of CustomerAccount class with FixedDiscountPlan and FlexibleDiscountPlan classes, respectively. These roles will force a good UML tool (like VP) when generating code for the CustomerAccount class to represent the associations by adding to the attributes of CustomerAccount two attributes, as foolows:

- *discount:FlexibleAccount*
- *flexiDiscount:FlexiblePlan*

These attributes will contain references :FixedDiscountPlan or :FlexibleDiscountPlan objects, respectively. The attributes may also take values null in case there is no link between :CustomerAccount and either :FixedDiscountPlan and :FlexibleDiscountPlan, respectively, which is possible given multiplicity 0..1 at the discount plans end. Changing the plan is effectively limited to assigning new values to the two attributes – discount and flexiDiscount.

There are no visible changes to the classes before the introduction of the discount plans.

A potential problem with the model is that price calculation is now only defined in discount classes. Clearly, if there is no discount, Job will still need somehow to calculate its price.

It is unclear how the Job price was meant to be calculated in the original model. My intention when constructing the Job class was that whenever saveJob() operation is invoked, the price for the job would be calculated. This decision seems reasonable as it is only when saveJob() is called that the :Job is "fixed" and no further changes (i.e. adding/removing tasks) will be possible. Clearly we need to ***reconcile*** *somehow the price calculation without a discount and the price calculation in a discount. In other words, introducing discounts will impact the Job class with possibly significant changes of its code.*

Question 2 (Decorator):

Consider the ***fixed discount plan*** and discuss whether the Decorator design pattern may be used to implement a fixed discount plan. Concentrate on how the price of a Job will differ between no discount and a fixed discount. The key insight here is that the fixed discount may be seen as a "decoration" (i.e. adding a bit extra calculation) of the full price calculation.

Revise the part of the class model, which captures the fixed discount, with the use of the Decorator design pattern. Reflect on:

- the changes of the classes Job and AccountCustomer, which became necessary in order to use the decorator pattern.
- whether the new model allows us to switch between no discount, and fixed discount with different discount rates (e.g. from 5% to 10%)

Hint: We need to define an interface, e.g. IPrice, with an operation, which allows us to calculate the price of a :Job. Let's define this operation of the interface as calculatePrice(j:Job):float. IPrice may have a number of different implementations – one for no discount calculation, e.g. FullPriceCalculator, and a few - to calculate the price with different discounts, e.g.

FixedDiscountCalculator. The essential bit of the model is to recognize that FixedDiscountCalculator::calculatePrice(j:Job):float relies on FullPriceCalculator::calculatePrice(j:Job):float, i.e. invoke it to get the full price and then amends it by applying the due discount.

Model answer

The model with a fixed discount is presented below

I introduced a new model FixedModelDesign, in which the Decorator pattern was applied to model the fixed discount. This was necessary so that I can retain the original BAPPERS model largely unaffected by the change.

A diagram which captures the affected classes is shown below. The other classes, shown in the assignment, and their relationships are not affected. Check the VP project of the model answer for a better view of the changes.

The model answer is a straightforward application of the Decorator pattern (check the pattern itself, e.g. using my lecture notes).

We define an interface IPrice with two implementation classes – FullPriceCalculator and FixedDiscountCalculator. The latter uses the former to get the full price and then applies the due discount, the discount rate is captured as an attribute of FixedDiscountCalculator.

The AccountCustomer class has now been extended with a new attribute:

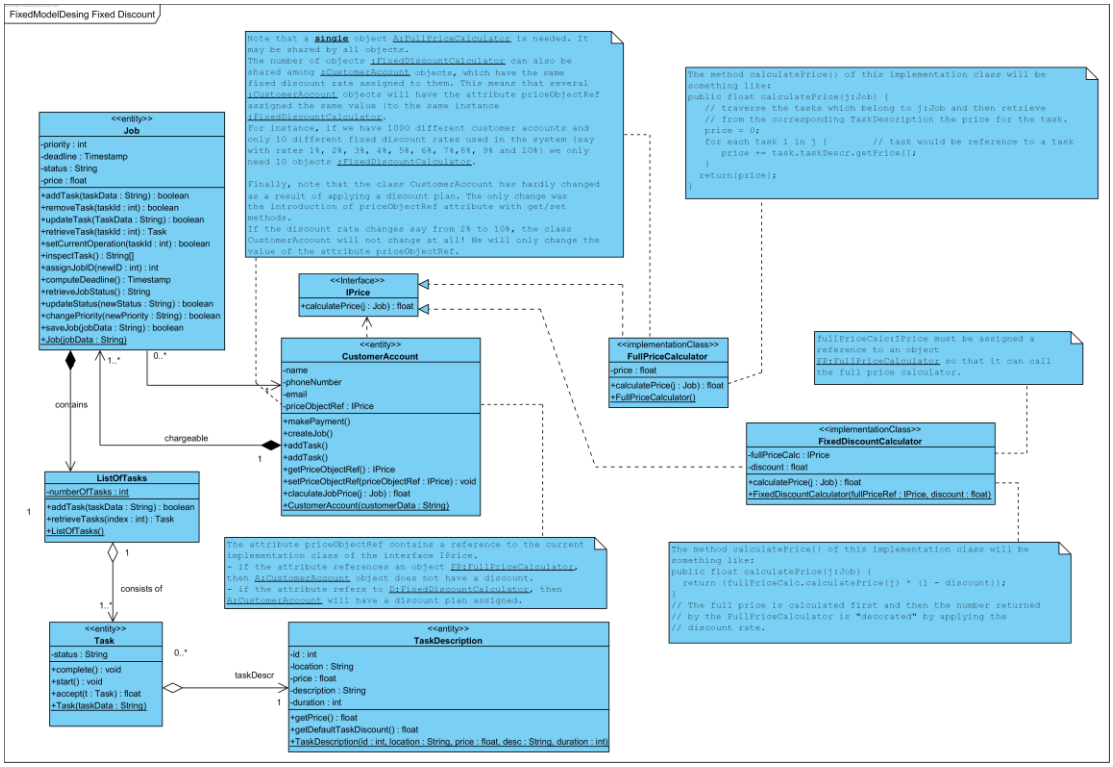
priceObjectRef:IPrice and getter/setter. This attribute is mere "implementation" of the dependency relationship between AccountCustomer and IPrice. This attribute will be assigned a reference to the respective implementation class, which represents the current discount position of :CustomerAcccount object, which may have either a fixed discount (in this case priceObjectRef will be assigned a reference to :FixedDiscountCalulator) or no discount (in this case priceObjectRef will be assigned a reference to FullPriceCalculator).

Under the current model :Job price *calculations are called by a :CustomerAccount object, which has dependency on IPrice interface. This implies that Job should be able to invoke when necessary the price calculation.*

A new public method was added to the CustomerAccount class: claculatePrice(j:Job):float. Whenever :Job invokes this method of :CustomerAccount it will have to pass itself (this) as a parameter.

An alternative arrangement would be for :Job to use IPrice directly. However, in this case :Job will have to retrieve priceObjectRef via the public method getPriceObjectRef() defined in the CustomerAccount class.

The trick, of course, is what we keep in the attribute priceObjectRef. If it refers to an instance of the implementation class FullPriceCalculator, then the full price will be computed. If, instead, priceObjectRef refers to an instance of the implementation class FixedDiscountCalculator then the discounted price will be computed. Note that FixedDiscountCalculator contains an attribute fullPriceCalc:IPrice. This is meant to store a reference to an instance of FullPriceCalculator (which in turn is an implementation class IPrice interface) so that when calculatePrice(j:Job):float of FixedDiscountCalculator is called it first gets the full price calling its own method fullPriceCalc.calculatePrice(j:Job), which returns the full price of the job. This full price is then "decorated" by applying the due discount rate.



Question 2 (Visitor):

Consider now the ***flexible discount plan***.

a) Let's work with the ***default*** discount plan first. The default plan is one for which the discount rates for individual tasks are captured in the class TaskDescription by adding an additional attribute defaultDiscount:float. All :CustomerAccount objects assigned a default flexible discount will share the default discount plan, i.e. in price calculation of the jobs from these customers the defaultDiscount rates will apply to the respective tasks.

b) Extend the model to allow for different (possibly) unique flexible discount plans to be assigned to each :CustomerAccount object (as is done in the provided extension of the model – class FlexibleDiscountPlan, ListOfDiscounts and TaskDiscount).

Discuss the implications of the model of flexible discount plan for the classes Job, Task, TaskDescription and CustomerAccount. Has anything changed? If so – what?

How easy is it now to change the discount plan assigned to an object :CustomerAccount?

Hint: In all job price calculations :Job will traverse the tasks that "belong" to :Job. The job price is the sum of prices for the individual :Task objects included in the :Job object.

Consider defining a Visitor interface, e.g. IVisitor, with an operation computeTaskPrice(t:Task):float. This interface will have different implementation classes, e.g. NoDiscountVisitor, DefaultFlexiDiscount (dealing with default flexible discount) and CustomisedFlexiDiscount (allowing for customized flexible discount).

Consider also defining an interface ITask with a single operation accept(v:IVisitor):float, which will be implemented by the Class Task (the class is already defined in the class model, but may need to be revised to make it an implementation class of the ITask interface).

These two interfaces are sufficient for us to compute the Job price using the Visitor pattern. The class Job will have an operation, e.g. calculatePrice(), which will traverse the individual :Task objects (i.e. access them in a loop), which belong to the particular :Job.

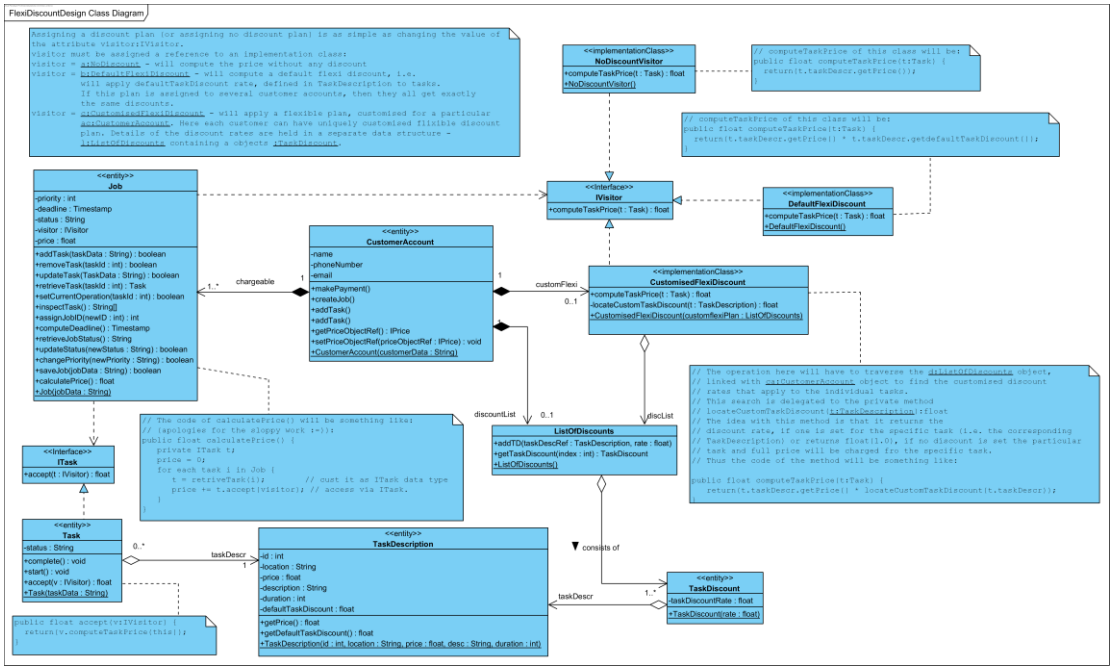
Let us arrange this loop according to the Visitor pattern:

- *For each :Task calculatePrice() should call accept(v:IVisitor) via ITask interface, passing to it as a parameter a suitable reference to an instance of an implementation class of the interface IVisitor.*
- *accept(), will then "accept the visitor" and will invoke v.computeTaskPrice(this) passing a reference to itself (this) so that the Visitor (v) can access the :Task data using its public methods to compute the contribution of the task to the :Job price. This contribution will vary depending on the implementation class – from no discount to possible some discount – default or unique.*

The loop will be completed and the contribution of all tasks encountered in the loop will be accounted for.

Model answer

The model with flexible discounts is presented below (check out the VP project for further details). I introduced a new model FlexiDiscountDesign, in which the Visitor pattern was applied to model the flexible discounts. Thus, I could retain the original model (and the model of the Fixed discount) unaffected by the changes made when working with this answer.



The model answer is a straightforward application of the visitor pattern. It uses two interfaces ITask (which is the adaptation of abstract Component in the Visitor pattern) and IVisitor.

*ITask offers the accept() method as defined in the pattern. The interface is implemented by the class Task, thus this class **had to be extended**.*

*IVisitor has several implementation classes (concrete visitors in the pattern description) – NoDiscountVisitor, DefaultFlexiDiscount and CustomisedFlexiDiscount. They implement ComputeTaks(t:Task), which is somewhat obscure name but the intention is that this method computes the **contribution** of a t:Task to the :Job price. The different implementation classes do the calculations of this contribution differently:*

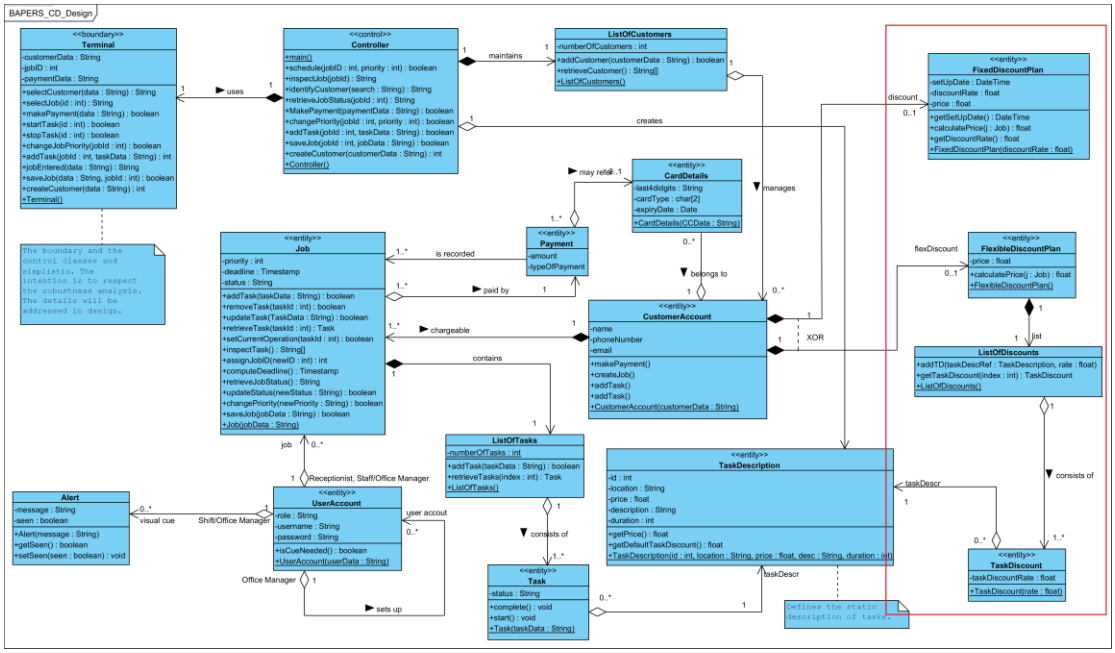
- *NoDiscount would take the full price of a Task as defined in the respective :TaskDescription,*
- *:DefaultFlexiDiscount will apply the value kept in the attribute defaultTaskDiscount of the respective :TaskDescription.*
- *Finally, :CustomisedFlexiDiscount will search for a matching task in the :ListOfDiscounts and will take the discount rate from there.*

The particular discount plan, which applies to a given :CustomerAccount (no discount, default flexi or customized flexi) is captured in the attribute visitor:IVisitor, added to the CustomerAccount class supplemented with a getter and setter methods. Changing the discount plan will be a mere change of the value of visitor:IVisitor (by using the setter method).

In this diagram I show more explicitly how the interfaces are used. I added a new method to the Job class, calculatePrice(), which is expected to iterate through the :Task objects of a particular :Job. Before the start of each iteration, the value of the visitor attribute held in the object c:CustomerAccount, which "owns" the particular :Job must be retrieved and assigned to the attribute visitor of :Job. Thus, the current discount plan set for c:CustomerAccount will be taken into account. Once a new :Task is located (by the iterator) its method accept(v:IVisitor):float is called. The :Task then follows the visitor pattern and calls v.calculateTask(this) passing as an attribute a reference to itself. The calculations of the contribution of this :Task to the price of the :Job are established by the object referred to by the attribute v:IVisitor. This will be an instance of the implementation class, which implements the discount plan currently assigned to the object :CustomerAccount.

Additional assignment (combination of Visitor and Decorator): Now consider the possibility that a new discount plan is introduced which combines both – the fixed and the flexible discounts. Can we combine the two design patterns that we have used in Q1 and Q2?

Appendix 1. Design class diagram for BAPERS



Peter Popov

Last modified: 25th October, 2018