IN2013, Week 5 – Design Patterns

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 (captured by the class TaskDescription) 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 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 for discount plans 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)?
- How easy is it to change the discount plan of a customer from say – no discount plan to a fixed discount plan of say 5% and from a fixed discount of say 5% to a fixed discount plan with 10%? 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 :Job price.

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 of extra calculation/processing) 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.

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

- whether the new model allows us to switch between no discount, and fixed discount and between fixed discounts 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 insight is to recognize that calculatePrice(j:Job):float of the implementation class FixedDiscountCalculator relies on calculatePrice(j:Job):float of the class FullPriceCalculator, i.e. the former invokes the latter to get the full price and then "decorates" the full price by applying the due discount.

Question 2 (Visitor): Consider now the ***flexible discount plan***.

a) Let's start with the ***default*** discount plan. This default plan is one, for which the discount rates for the 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 their respective jobs the defaultDiscount rates will apply to the tasks of the jobs.
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 class diagram provided in the, i.e. by the classes FlexibleDiscountPlan, ListOfDiscounts and TaskDiscount.

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

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

Hint: In all job price calculations :Job will traverse (i.e. access one by one) the tasks that "belong" to :Job. The job price is the sum of prices of 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 can be extended by adding an operation, calculatePrice(), which will traverse (i.e. access them one by one in a loop) the individual :Task objects, which belong to the particular :Job.

Let us arrange this loop according to the Visitor pattern:

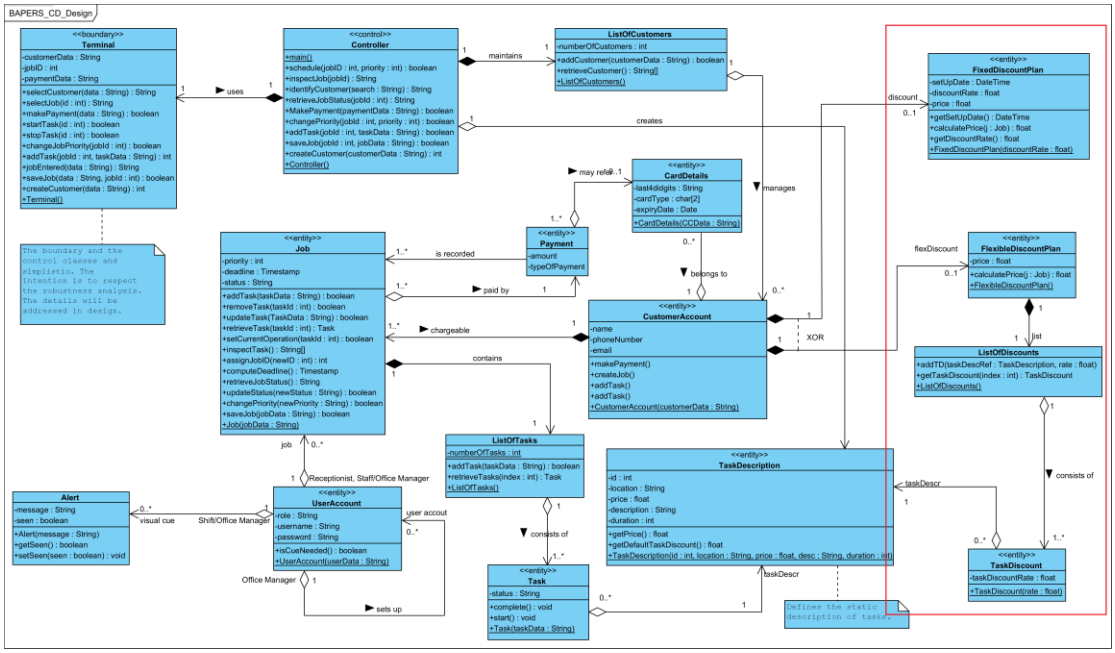
- *For each :Task calculatePrice() should call accept(v:IVisitor) via ITask interface, passing to it as a parameter a suitable reference to an instance of an implementation class of the interface IVisitor.*
- *accept(), will then "accept the visitor" and will invoke v.computeTaskPrice(this) passing a 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.*

The :Task contribution will vary depending on the implementation class of the IVisitor interface – from no discount to some discount – default or unique.

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

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