Assignment 5: Design Patterns Study Paper

Xin Tao (GTID: xtao41), Date: 07/09/2017

Topic 1. Select two design patterns and present a brief description.

Design patterns #1: Abstract Factory

Abstract Factory pattern is one of the creational patterns, which is used to control class instantiation. Abstract Factory pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes [1]. Usually, this pattern consists of a client and an abstract factory. At run-time, the client creates a set of concrete factory classes by implementing the abstract factory. Then a set of related or dependent objects were created by the generic interface of the factory per the selection of concrete factory class [2]. Since object creation is implemented in factory interface and the factory only returns an abstract pointer to the created concrete object, the client is unaware of the actual concrete type of the object. This means a) the client code deals only with the abstract type, and the access of concrete objects is done only through factory abstract interfaces; b) adding new concrete types is very easy as it only needs slightly modification of factory with pointer of the same abstract type[3].

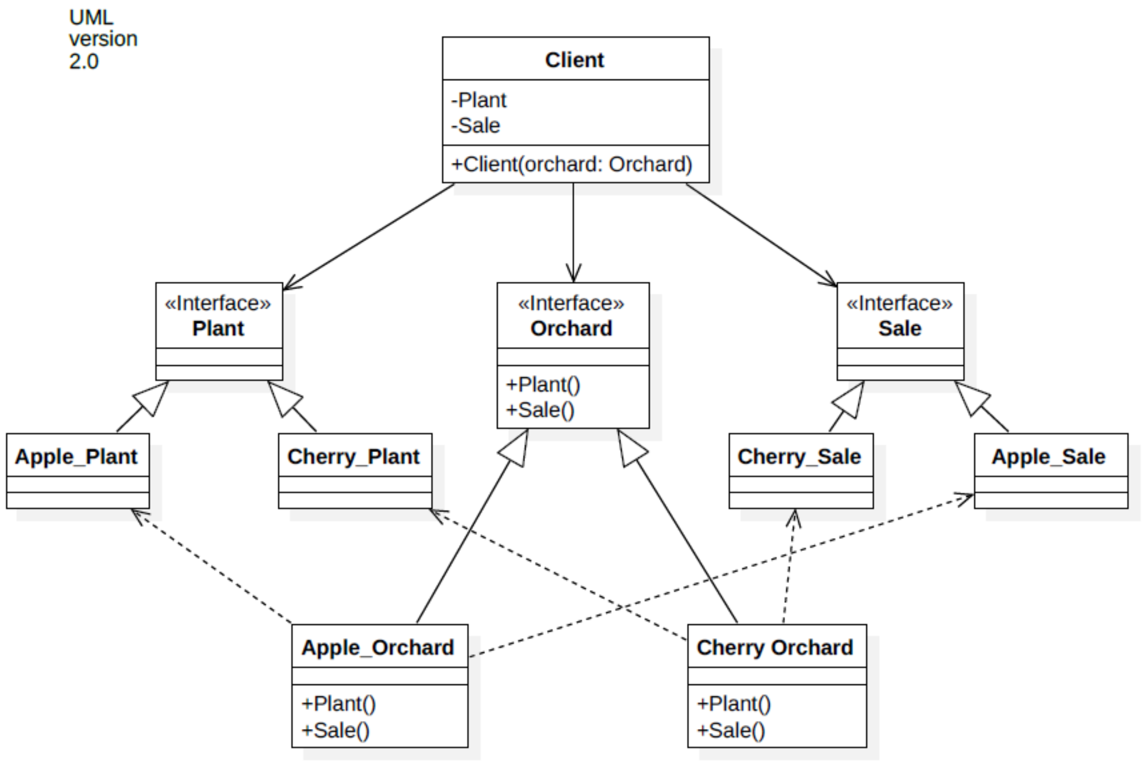
Design Pattern #2: Prototype

The Prototype design pattern is another type of creational pattern. New objects are created based upon a template of an existing objects through cloning. The new object is an exact copy of the prototype with possible modifications. In prototype, creation is through delegation whereas in abstract factory it is through inheritance (GoF, p95). This pattern is generally used when initialization of an object directly is costly. It simplifies the use case when new objects of the same type have mostly the same data.

Topic 2. Practical situation for Design Pattern #1.

A problem exemplar is an abstract factory class *Orchard* that produces any type of fruits, such as *apple orchard* (producing apples), *cherry orchard* (producing cherries) and etc. The class *Orchard* has two functions, *Plant*() and *Sale*(). For each type of orchard, it has a different implementation of *Plant*() and *Sale*(), and thus would create different concrete objects like *Apple_Plant*, *Apple_Sale*, *Cherry_Plant*, and *Cherry_Sale*. Specifically, if it is *apple orchard*, then planting method will need particular conditions for weather, site and soil. If it is a *cherry orchard*, then the T-budding planting technique is commonly used with different conditions than planting apple trees. Similarly, the Sales channels for the two types of fruits are different.

The client is aware of abstract interface class *Plant*() and *Sale*(), but is unaware of the concrete objects/types (e.g. apple_plant, cherry_sale, etc), which are all created from the same *Orchard* class functions and would share a common theme. In this example, there are two operations in abstract factory *Orchard* - *Plant*() and *Sale*(). There are multiple concrete factories, such as Apple_Orchard and Cherry Orchard. Concrete objects *Apple_Plant*, *Apple_Sale*, *Cherry_Plant*, *Cherry_Sale*… are created by implementing the abstract interface *Plant* and *Sale* for each concrete factories. The UML class diagram shows the implantation of such system, with only two concrete factories (apple and cherry) as an example.



Design pattern #1 – Abstract factory is a very good solution. First there can be any number of concrete factories depending on the demand of the system. Adding these concrete factories can be very simple- no need to modify the code that uses them (client code and abstract interfaces), even at runtime. Second, the concrete objects (*Apple_Sale, Cherry_Plant*) are isolated from the client and are controlled by the abstract factory (*Orchard*). Third, maintaining the consistency is fairly easy, as it is the concrete factory's job to make sure that the right products are used together (i.e. the selected Orchards should have the similar functionalities) [4]. Finally, by utilizing an abstract factory for creation, we can then ensure that the appropriate object sets are created based upon the type of *orchard* that is being selected.

Topic 3. How well, or how poorly design pattern #2 supports the exemplar.

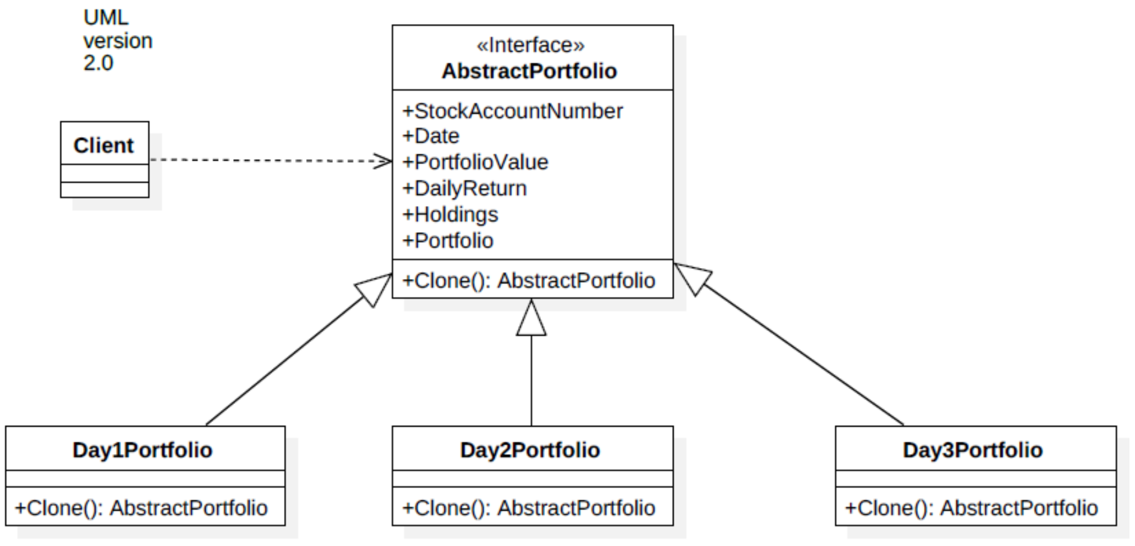
Design pattern #2 – Prototype creates a clone of itself, including all its attributes/properties. Prototype and Abstract factory can be overlapped and/or used in a complementary way. In the example of Topic 2, it is possible to create an instance, e.g. Apple_Orchard (e.g. using abstract factory), and then cloned for every other object with necessary modification to obtain object like Cherry Orchard. However, prototype would NOT be a good candidate in this scenario for the following reasons:

1) Usually, prototype design pattern is used when objects or object structures are required to be identical or closely resemble other existing objects or object structures [6]. However, different type of *Orchards* are more likely to have totally different properties in terms of how the fruits are planted and how they are sold.
2) The properties of subclasses, Apple_Plant (Apple_sales) and Cherry_Plant (Cherry_Sale) are be loosely coupled. Not only the specific conditions (e.g. site, soil, or weather) for plant apple and cherry are different, the number of requirements is also different. For example, apples may be easier to grow and thus have 30 requirements, whereas cherries are difficult to grow and have 100 requirements. Furthermore, they would have different planting techniques. We should treat them as loosely coupled individuals. Thus, cloning cannot satisfy these requirements or it needs tremendous updates. On the other hand, factory pattern assures appropriate objects can be created based on the type of *orchard*, and therefore is a better implementation.
3) Also, in this situation, flexibility is important and we would like the objects can be extended in subclasses, to which the client delegates responsibilities in order to

deliver specific objects that may come from different families. It will take care of all the instantiation logic hiding it from the clients. For prototype pattern, it would be difficult to let the system handle multiple families of objects and concrete classes that are decoupled from clients.

Topic 4. Practical situation for Design Pattern #2- Prototype Design Pattern

The example I give is daily portfolio information of stock account. Here I treat portfolio information of each day to be a different object, whose attributes are *StockAccountNumber*, *Date*, *PortfolioValue*, *Portfolio*, *DailyReturn*, *Holdings*, and more (such as trading data, which is not included in the UML). The UML class diagram below describes an implementation of the prototype pattern. The *AbstractPortfolio* is the base class for the types of objects to be cloned and generated. This class contains a *Clone()* method that returns itself.. *DayxPortflio* (x=1, 2, 3, ….) all inherit from the base class *AbstractPortfolio* with necessary modification and include any additional required functionalities.



The prototype design pattern here is a very good solution because the portfolio info and transactions are changing everyday, and you would want to make a copy of the object that holds your account information, perform transactions on it and update portfolio info and then replace the original object with the modified one. The new objects (*DayxPortflio*) to be instantiated are specified at run-time, thus they can be cloned very dynamically. The implementation of prototype invokes the *clone()* method on the prototype. This method is preferable because we want the new objects created with a pre-configured state. Because there might be tons of information in stock account and many of them are not varied daily, prototype design pattern could simplify the onerous configuration code required using other design pattern which involve *new* operator.

Furthermore, in this example, the object (portfolio) in this example could be extremely large since it may store a lot of stock data, trading data, account data as well as calculation methodologies and pre-existing stock information. We don’t want to use the *new* method to instantiate new object because it is costly and requires a lot of time and resources in terms of memory or computation. Since a similar object is already existing from previous days and cloning is considerable less expensive than creating new object afresh (using *new* method), thus Prototype pattern is preferred. It also adds the benefit that clones can be streamlined to only include relevant data for their situations. [6]

Topic 5. Explain how well, or how poorly, design pattern #1 supports the exemplar.

Design pattern #1- abstract factory would NOT properly support the example in Topic 4 for the following reasons.

1) Abstract Factory design patter is usually used when we have a super class with multiple sub-classes based on the input. However, the *portfolio* example is basically just different objects of the same class.
2) The results of the abstract factory implementation is to return one of the sub-class. It takes out the responsibility of instantiation of a class from client program to the factory class. However, the return of my *portfolio* example is just another object and no subclass is involved. Also we don't have properties passed from client program.
3) The cost of subclassing is inherent in Abstract factory design pattern. The stock account may consist of large amount of data and is too expensive to use *new* operation every day (let alone if we need such instances every second). We need to modify this data in our program multiple times. So it's a bad idea to create the object using *new* keyword and fetch all the data again and again from the database every time. This would be extremely resource and time consuming The better approach would be using clone/prototype method as discussed in Topic 4.
4) In abstract factory design pattern, if we design to avoid the use of *new* keyword in client applications, we often end up with factory classes mirroring product classes, resulting in a parallel hierarchy- one for each concrete product classes and corresponding factory classes. If we have multiple product classes of the same interface, it will lead to massive duplication of factories [5].
5) In abstract factory, adding a new product (property) to the system requires extending the abstract interface. All of its derived concrete classes also must change, including adding new abstract product class/new product implementation,

extending abstract factory interface, implementation of extensions for derived concrete factories, client extension to use new product. This prevents efficiently adding new properties, e.g adding a accumulative return attribute to the portfolio info, to the account object if using abstract factory design pattern.

References

[1] Gamma, Erich; Richard Helm; Ralph Johnson; John M. Vlissides (2009-10-23). "Design Patterns: Abstract Factory". informIT. Archived from the original on 2009-10-23.

[2] http://www.blackwasp.co.uk/AbstractFactory.aspx

[3] https://en.wikipedia.org/wiki/Abstract_factory_pattern#cite_note-3

[4] https://www.codeproject.com/Articles/35789/Understanding-Factory-Method-and-Abstract-Factory

[5] Gang Of Four, Content Creation Wiki for People Projects And Patterns in Software Development.

[6] http://campus.murraystate.edu/academic/faculty/wlyle/430/rc008 designpatterns_online.pdf