2016 OOSE Assignment

Brendan Lally 18407220

Design Patterns and Polymorphism

One of the patterns that has been employed in this design is the composite pattern. The composite pattern lets you treat individual objects and compositions of objects uniformly. In the case of the assignment design, there are three “objects”: companies, business units and bank accounts. All three of these extend an abstract class, Property. The composite pattern comes into play because companies are treated as a composition of properties. This pattern was chosen because: (1) the structure of companies in relation to business units and bank accounts represents a part-whole hierarchy and (2) it enables all three properties to have their profit calculated using polymorphism. Having each type of property inherit from Property also allows many basic setter/getter methods to be polymorphic as well.

Another pattern used was the template method pattern. The template method pattern defines the skeleton of an algorithm in a method, while deferring some steps to the subclasses. This was achieved by having each type of event (wage, revenue and value) inherit from an abstract Event class. The Event class defined the skeleton of an algorithm for running an event, leaving the subclasses to encapsulate what varies, i.e. how they actually handle the increasing and decreasing steps. This incorporates polymorphism as the client does not need to know which type of event it is running, as it will run any event that inherits from Event.

Using the template method pattern allows you to reuse code that would otherwise be duplicated in each event type class. It also increases the extensibility of the system as the client could choose to add new types of events in the future and only require them to inherit from the Event class and implement the increase() and decrease() methods.

A similar design was used to implement plans, however, this time, the strategy pattern was used instead of the template method pattern. The structure is very similar, except this time the whole algorithm for buying and selling property is deferred to the subclasses, BuyPlan and SellPlan. The polymorphism works by the client just having a plan object and executing runPlan(), regardless of its type.

Alternative Designs

One alternative design choice could be to introduce the factory method pattern[1] for instantiating the property, event and plan objects. Currently these objects are created in the same method that reads in their respective file. The factory method lets a class defer instantiation to subclasses. In the case of creating properties (business units or companies) you could create a property Factory which would determine which property to create based off a string “type” in the property file.

1. Introducing the Factory Method Pattern would more than likely be a better design choice than the current implementation. I did try and use it but I couldn’t quite get it to work how I envisioned it.

The benefits of changing this to use the factory method pattern are that it introduces separation between the client (the code using the object) and the families of classes (Properties, Events, and Plans). This results in lower coupling by having the factory choose which subclass to create, removing the hard-coded dependency of the client code calling the constructor of a Property, Event or Plan.

The cons of using the factory method patterns in this instance are that it would introduce more classes into the system, potentially complicating it.

Another alternative design choice would be to implement Event using the strategy pattern. Currently this implement the template method pattern, where each subclass only implements the increase and decrease steps in the run event algorithm. Implementing these using the strategy pattern would involve each event subclass implementing their own run event method, instead of just the increase and decrease steps, similar to Plan.

In this case, there doesn't seem to be much benefit gained from using the strategy pattern over the template method patters, as both patterns will achieve a similar result. Instead of having each subclass only implement certain steps, each subclass would instead have the entire run event or run plan algorithm. Using the template method will slightly reduce the amount of code reuse as you are encapsulating the steps that vary instead of just encapsulating the entire algorithm.

Testability

The design achieves testability by following the model-view-controller pattern and by trying to separate concerns where possible. Having separate models for events, plans and properties allows each of these to be tested individually. One way of increasing the testability of a piece of software is to minimize coupling between classes and reduce dependencies. To reduce dependencies, it is better to pass information between classes as parameters rather than instantiating new instances inside a method.

All the model containers are stored in the self-contained controller class. Encapsulating all this information into one class enables good unit testing as all the program functionality only requires access to the information in the controller class. Once the objects have been instantiated there is minimal dependency between the classes. If the model objects could be mocked, this would increase the testability of the design.

Currently, the creation of model objects in the file reading methods does not promote good testability. As previously mentioned in the 'Alternative Design' section, this design could be changed to use the factory method pattern. Using this pattern would allow for increased testability by allowing the factory to return mock objects (similar to the situation from the lecture slides. Lecture 6, slide 12).