OOSE Assignment 2016 S2 Report

Jordan Pinglin Chou
18348691

Object Oriented Software Engineering

October 26, 2016

- Where have you used polymorphism and why?
- Discuss design patterns that incorporate polymorphism.
- How does the design achieve testability, how does it make unit testing easier.
- Discuss two plausible alternate design choices and explain their pros and cons.

Where have you used polymorphism and why? Discuss design patterns that incorporate polymorphism.

I have used polymorphism throughout the assignment. There are many instances in the implementation where it does not matter what the subclass is, the calling method will just invoke a method call on an interface and delegates the action to a subclass.

The use of the **strategy** pattern throughout the program allows the context of the program to use instances of `Event` and `Transactions` without having to know what they are. Both `Event` and `Transaction` implement an `Updateable` interface. This allows a common controller to be used amongst them. This controller will just call `update()` on its set of `Updateable` objects and the program will figure out which methods to call from there.

Having this `Updateable` interface allows for the program to be extended with ease in the future. If more events or plans are to be added to the program the programmer can just implement the `Updateable` interface and specify a controller for it. This also allows for great reusability as currently both the Event and Transaction controllers are instances of the same class.

This was done because Events and Transactions had very similar behaviour between them, they each carry out a task and because they are models that store all the logic they need.

Polymorphism was also achieved via the **observer** pattern. Wage events hold a list of observers which implement a `WageObserver` interface where the event just has to call `update()` to notify the observer. This allows for great extensibility as any future events that require such behaviour can just implement the interface and implements.

Another way polymorphism was achieved was through the usage of abstract `Reader` and abstract `Property` classes. For property the `update()` method is delegated to the subclass. Regardless of the type of property they all have a way of calculating profit.

How does the design achieve testability, how does it make unit testing easier?

The program makes use of dependency injection to make testing simple. Because the only instantiating is done in either a factory or main it makes it easy to mock objects and stub the return to whatever is required for the test.

Each controller is simply passed the the objects it need. This allows for unit tests to mock objects easily.

Low coupling through polymorphism allows for easier unit testing because you can isolate the code that you want tested.

MVC also allows for sections of the code to be separated out and tested properly on its own.

Discuss two plausible alternate design choices and explain their pros and cons.

Structural Alternatives

Singleton

The singleton pattern is an alternative structural pattern can be used for the program. The singleton pattern restricts a class so that only one instance of it can be active at any given time. The singleton pattern can be used for the primary company of the system. There can only be one primary company for the system and the primary company does not change for the duration of the simulation so it could be an alternative way to designing the system. The singleton can also be used to ensure that a company only has one bank account.

Pros

- Makes sure that only one primary company is active at a given time in the simulation
- Makes sure that each company only has on bank account at a given time in the simulation.

Cons

- Adds complexity to the system
- Have to create extra singleton classes for Primary Company and BankAccounts
- Could break polymorphism depending on the way the extra classes are laid out
- Would reduce the testability of the system, it's difficult to test the singleton pattern

Composite

Because each company owns a list of properties the composite pattern can work well. The composite node will be a company, and the leaf nodes will be business units. This works well because companies can own zero to many properties but properties cannot own other properties. The difficult choice for this is to decide where a bank account fits in. You could possibly place it within the company or have it as a leaf node. It would probably be better to have it separate to the tree of properties due to how frequently you need to access the bank account.

Pros

- Makes sense logically to do it. Each company can own multiple properties so it makes sense for it to be a composite node, each business unit cannot own anything so it makes sense for it to be a leaf node
- It allows for easy traversal of the tree to get the properties associated with a certain company
- Because components of the tree use an interface, it makes it very general. We can reuse the tree structure in the future if we need to.

Cons

- Because the tree is composed of an interface we have to add in checks to make sure that the types within the tree are correct (what we expect)
- Once the tree has been made it can be difficult to change it, varies greatly with how the tree is implemented but buy/sell decisions may need to have extra steps implemented.

Behavioural Alternatives

Command

The command pattern defines a command interface with a method signature defining a command to execute. For the simulation you can pass the commands from the creators (factory or reader) to the controllers (invokers). The controller decide when to execute or invoke the command. This could be suitable for events, plans, and properties as they all have things to 'do', i.e events have to carry out the event, plans have to carry out the plan, and properties have to update their values.

Pros

- Useful for separating concerns, especially since creation of the command (event, plan, properties) is not dependent on the invocation of the command (controllers) and vice versa
- The command instance can be instantiated by the client but run later by the invoker. The client and invoke need not to know anything about each other.
- Extensibility: We can add new commands easily by implementing the command interface

Cons

- Using the command pattern increase the number of classes we need since we would need to have a class for each command.