OOSE Assignment Report 2016

Updated: 25th October, 2016

Introduction

The following report is to accompany the OOSE Assignment 2016. It will discuss the use of polymorphism, design patterns, testability and plausible alternatives.

Design Patterns

1. Polymorphism

Polymorphism has been used in multiple places through out the assignment. This has been done decouple the system. To allow for the use of polymorphism inheritance from abstract classes and interfaces has been used.

All Events share common classfields and as such Event is an abstract class that contains all common details and functionality. All subclasses of Event must implement a performEvent() function. Thus guaranteeing any event can be invoked through a call to this common function between all event subclasses.

Similarly, Plans have common classfields and all subclasses of Plan implement a performPlan() function. This allows for programming to an interface rather than an implementaion. The caller doesn't need to know what type of plan is to be performed. Only that the performPlan() can be called to invoke the method.

Further more, polymorphism is used for WageObservers in the simulation. This allows any class interested in updating their wages to register as a WageObserver to receive updates. For further extensability other classes can become WageObservers if they implement the WageObserver interface. It doesn't matter the type of class that implements the interface, only that they implement the update() method.

2. Design Patterns

The following design patterns were implemented in the assignment:

Factory Method Pattern
The Factory Method was used to create Events and Plans. As each subtype of an Event share common fields, as well as the subtypes of Plan sharing common fields. This allows for the subtype to be hidden from the caller, but know that an Event or Plan will be returned. This also allows for extensability in the case of more Events or Plans being added in future.

Template Method Pattern
The Template Method was used for reading and parsing each of the three files for the simulation. This was appropriate as all three files shared common steps for opening, reading a line and closing the file. The template method for processing the line allows the subclass to redefine certain steps of the algorithm without altering the larger algorithm's core structure.

Observer Pattern
The Observer pattern was used for notifying all classes that implemented the WageObserver interface of a change in their current wages. The *push* method was implemented, rather than the *pull* method, in this situation to ensure all WageObservers are updated at the time of a WageEvent occurring. As the controller would know when a change needed to be reflected not the models. In the case of the simulation only Business Units were WageObservers, but this allows for other classes to become observers if they need to update their wages in a similar manner to Business Units.

Composite Pattern
The Composite Pattern has been used to allow the simulation to treat all Companies as an individual company or a composition of Properties. Companies are able to own zero or more properties, excluding multiple bank accounts. The super class for Property enforces that all subclasses implement a calculateProfit() method. Regardless of whether the company owns multiple properties or zero, profit can be calculated via a call to calculateProfit(). A leaf is a Business Unit and a Company is a composite node.

Testability

Dependency Injection has been used to assist in testability. Rather than each class having a new instance of a class it needs, an object is passed in instead. This will help when attempting to mock certain functionality of the simulation. This allows for easy replacement of classes, without other parts of the program knowing. Hard coded dependencies have been avoided by not allowing static method calls within the program.

Each model class has been given a relevant toString() method to allow for assistance in debugging logic errors. Thus the state of an object can be printed to analyse incorrect calculations or changes during the running of the simulation.

Dependency Inversion Principle has been used in the Event and Plan classes. The invoker isn't concerned with what kind of Event or Plan it is, but rather that they all perform an Event or Plan respectively. If new Events or Plans were to be added for testing or extending the system they would only need to implement the repspective abstract method in Event and Plan.

Segmenting the Simulator Controller into smaller controllers allows for increased testability. Rather than a single controller with a large amount of functionality to unit test, each controller can be tested in isolation. This will make unit testing simpler as there is less to test per controller.

Alternatives

1. Alternative One

The first possible alternative is to implement the Command Pattern. Plan and Event subclasses could be created by one part of the simulation. All the necessary details to execute the action would be encapsulated within the relevant Command object. The executer of the command is a different part of the system. The executer decides when to execute the command and the necessary details for execution, parameters, are within the Command object.

Pros

- Allows for flexibility in timing and order of execution.
- Commands can be passed like other object.
- Decoupling between sender and receiver.
- Easily extensible, if new commands were to be added.

Cons

- The normal disadvantage to this approach is the increased number of classes needed. As each command needs its own class. (Currently this would not be different to my implementation).

2. Alternative Two

Currently there are six different classes for each Event type. From the three base Event types of wages, revenue and value. There is also a decrease or increase for each. Thus six classes in total. The decorator pattern could have been used to wrap the base class of Wage, Revenue or Value. This could have been used as there are different combinations that could have been created. While the core functionality of the calculation is the same and the only difference is the increase or decrease. This follows the *Open Close Principle*, where classes should be open for extension, but closed for modification.

Pros

- Flexible alternative to subclassing, to extend behaviour.
- Supports Open-Close Pinciple
- The current wrapping for Events in this system would only extend to increase or decrease, thus not an overuse.

Cons

- Overuse of the decorator can result in overcomplicated systems.
- Instantiation of concrete classes can become complicated if multiple decorators are used to wrap a single object.

End of Report