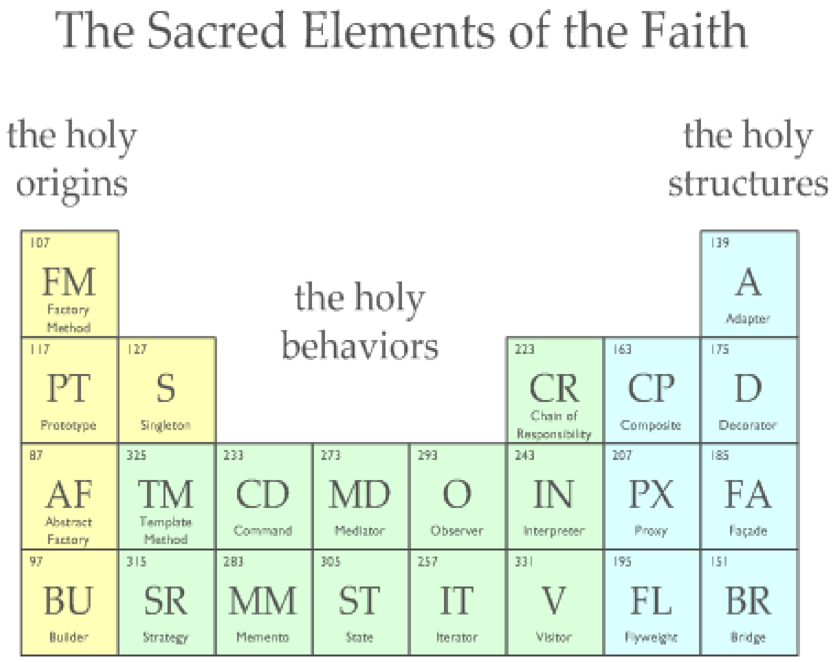
OOSE200 Report
Company Training Simulation

Connor Beardsmore - 15504319



Curtin University
Science and Engineering
Perth, Australia
October 2016

"Company Training Simulation"

Polymorphism

Throughout the Company Simulator, polymorphism is extensively utilized to both generalise and decouple code, leading to increased testability. To allow for the use of polymorphism, both implementation inheritance and interface inheritance has been employed.

Utilisation of the Strategy patterns in both Event and Plans allows the context to use the subclasses without knowing their exact types. The use of polymorphism here encapsulates the implementation details, letting each family of algorithms be used interchangeably. The Strategy pattern is also used within Properties, allowing us to call calcProfit() on any Property, without needed to know anything about the actual class or its implementation. This use of polymorphism enhances the code by improving both its extensibility and maintainability, due to the ease at which further subclasses can be extended from the abstract class. As calcProfit() is a given method within any Property or its subclasses, all Properties can be simply stored within one map.

Polymorphism is further employed within the Observer pattern. A list of WageObserver objects are stored, allowing us to store any class that extends the interface. This results in loose coupling between the subject and the observers, since all the subject needs to know is that a WageObserver implements the interface and nothing more. A by-product of this loose coupling is increased testability. No changes are required to further extend the system to add new observer types, enhancing the systems extensibility.

Testability

The heavy use of design patterns and polymorphism produces a system that is easily testable. The use of the Factory pattern allows us to instantiate both Events and Plans with testing in mind, simplifying the use of mock objects in this situation. Dependency injection further allows for increased testability, with mock objects able to be created in the main due to the low level of hard coded dependencies.

Each model and controller has a relevant toString() method to allow for simple debugging of each objects state. This allowed for numerous test files - both valid and invalid - to be run to ensure the correct logic of the simulator. The clear and concise exception handling throughout the system further helps testability.

Design Patterns Implemented

Factory Method Pattern

A Factory was employed to encapsulate object instantiation for both the Event and Plan subclasses, allowing the specific subclass type to be hidden from the calling method.

Dependency Injection Pattern

The Dependency Injection pattern worked to remove all hard-coded dependencies, with the primary injector code being located in the main method with all calls to *new* located in either the main or the two Factory classes.

Model View Controller Pattern

The MVC *compound* pattern was utilized for the overall architectural design of the system, due to its flexibility and its strong separation of concerns.

Observer Pattern

An observer was set up for WageEvents, allowing all relevant Property's to be updated easily by the notify() method. This also allows future models interested in Wage changes to be easily implemented by simply implementing the Observer interface.

Composite Pattern

The tree of Properties form a version of the Composite pattern, where each Company owns zero or more other Properties. These properties can either be leaf nodes (BusinessUnit) or further composite nodes (Company). This allows for simple recursive calculation of profit throughout the entire hierarchy.

Template Method Pattern

The Template Method pattern was used for file reading. The common code for opening and closing files was kept in the superclass. Since every reader parses each file differently, the subclasses provide their own implementation for the protected abstract processLine() method.

Strategy Pattern

The run() method located in both Event and Plan subclasses is a form of the strategy pattern, with each subclass implementing this method differently. Also utilizing the Strategy pattern is the calcProfit() method in Property subclasses, as all Properties calculate profit differently.

Miscellaneous Patterns

The use of for each loops throughout the system illustrate a form of the simplistic but ever useful Iterator pattern. While the Decorator pattern was not used in any of the designed classes, the objects used for file reading from the Java API illustrate an example of the Decorator pattern.

Alternative Design Choices

Despite the design having a high level of testability and maintainability, there are alternative design choices that could have been employed. The main alternative choices are the use of Factories, the controller layouts and additional classes for the multiple Event choices.

Controllers

Currently, there is one primary controller and one controller for each major model set. Since the primary controller is passed around, there are numerous calls to get the sub controllers, resulting in breaking the Law of Demeter. Solving this would require extra methods in the primary controller and since the number of calls to getters was minimal, this was not a major concern. Another alternative was to keep one large controller to perform all the Property, Event and Plan actions. However, this set-up would void separation of concerns and result in a highly coupled controller class.

Factories

Factories were employed for the instantiation of both Event and Plan objects, but not for Property objects. This is due to BusinessUnits having specific fields that are not relevant to a Property object. Thus, to set these fields, the returned Property from the Factory would need to be downcast to be a BusinessUnit. A Factory allows us to instantiate a Property object without knowing its specific type, yet having to downcast voids this principle. This could have been solved by passing the whole String to parse to the Factory, yet this seems to void separation of concerns, as it is the PropertyReader's role to parse the input file. A form of the Builder pattern could have been modified to work in this situation, yet it was more simplistic to simply instantiate Properties from a method within the PropertyReader class.

Additional Event Classes

There are currently three Event subclasses, each having a boolean field representing whether the class is an increase or decrease Event. An alternative solution was to write six subclasses, one each for an increase or decrease Event. This results in double the amount of classes, with each class being identical other than one plus or minus operation. However, the extra classes would remove the additional control flag required in the current set-up. Weighing up both the pros and cons, for the scope of this project it makes sense to minimise the number of classes required.