OOSE - Assignment 2019

Written by Ahmad Allahham

Table of Contents

Design Patterns Used, their Adaptation, and their purpose.

For my assignment, I chose to tackle problem B: Election Campaign Manager

Observer Pattern

In this assignment, we are create an application that searches twitter for words that are currently trending, then reports them back to the application, which then decides who, from the list of people stored in its database should it notify. The application must notify only a select group of people; which depends on the role of the person in the Election Campaign (ie: Volunteer, Candidate or Strategist) and if a special connection has been set up (notification setting) by the user between the person and a particular policy area.
Given the problem at hand, we know a number of things. Firstly, that different types of people must be notified when an event occurs (such as adding a keyword or when a keyword is trending). Secondly, that the keywords and the talking points (short responses to what is trending in social media) are relevant to particular groups of people through policy areas. And lastly, that we must output to the user any changes that occured due to an event taking place.
The outlined information suggests that we should incorporate the observer pattern in our implementation. The observer pattern is typical used to notify objects in the system when an event occurs. These notifications may (and should) differ depending on the observer. In our case, there are three distinct observers: a PersonObserver, a PolicyAreaObserver and a NotificationViewer. Each of these observers respond differently when an event takes place (events in this case are adding of keywords and talking points and when certain keywords are trending).

Template Method Pattern

While all types of people posses the same information (Name, type, id and contact details), not all people are notified when an event takes place. For instance, if a new keyword is added, only volunteers (not including the PolicyAreaObserver list and the Notification viewer) are notified about this event. This suggests that while all types of PersonObserver objects (types) behave mostly the same, when an event takes place, they must respond differently.
Having mostly common functionality with exceptions suggests that the Template Method Pattern should be incorporated into the solution. The template method pattern is made up of an abstract class; which houses all of the common functionality as well as classes that “extend” (inherit) from this abstract class. These classes will receive all of the common functionality of the abstract class, however they handle their own functionality differently. For this problem, the template method pattern was used to establish the relationship between the PersonObserver (abstract class, which is also an Observer - as mentioned in the discussion earlier about the observer pattern) as well as the three types of people: VolunteerObserver, CandidateObserver and StrategistObserver. These classes that inherit from PersonObserver all possess the

information and functionality present in PersonObserver, while responding differently to when an event happens (ie: different update implementation depending on the notification received)

Coupling, Cohesion, and Reuse

For a class to have low coupling, it must have low dependency on other classes and be able to "stand on its on". One of the ways lower coupling between classes is through Dependency Injection. Dependency Injection is the idea of "injecting" (importing) in classes that the current class depends on. This is done to avoid a class "containing" or encapsulating another class and instead, have the classes use each other, while being able to exist on their own. In my application, I used factory methods to produce any objects that must be created on the fly and injected them into the classes that either store them or make use of them.
Another way of decreasing coupling and increasing cohesion (for methods and classes to have a single well defined task) is to use an architectural pattern known as the MVC. The MVC (Model, View Controller) seeks to separate the functionality of the program into three distinct packages. The model is responsible for storage of data, the view for managing data input and output and the controller for facilitating communication between the view and the model as well as performing any complex logic prior to data transfer. In my implementation, I placed the input and output of data as well as the display of available options in a class called "NotificationViewer"; which resides in the package "View". The storage of data (people, policy areas and their associated fields) is placed in the "model" package in a class called "NotificationsManager". The processing of requests and inputted data and the passing of this processed input is handled by the "NotificationController", placed in the package "Controller". As for code reuse, the classes NotificationController and NotificationManager each implement their own interface. This provides the ability to create different implementations of NotificationController and NotificationViewer in the future, should a completely different approach (or problem) be implemented. Furthermore, the use of the observer pattern allows the addition/ removal of observers, given that they implement the methods specified in the Interface "NotificationObserver".

Plausible Alternative Design Choices as well as their Advantages and Disadvantages.

- **Alternative Design #1: Use a field for the “type” property instead of implementing the Template Method Pattern**

When an event happens that triggers notifications, the system must know the type of the person being notified (as different types of people are notified differently). For my current implementation, I chose to use the Template Method Pattern as it allows their to be a class which houses all the implementation and data handling shared across all types of people, while allowing for unique implementation of certain methods. This approach meant that the model could cycle through the entire set of PersonObservers and simply call their update methods, letting the Observer (depending on its type) respond to the notification call. An alternative approach could have been to not implement the Template Method Pattern at all and instead store the type of person as a private field inside of the PersonObserver class. This implementation reduces the number of classes that exist in the system. However, the use of “type” as a field means that the database must check the value stored in this field and call an appropriate method depending on this value. For example, if it checks the value stored in the type field in a particular PersonObserver and finds that this person is a “Candidate”, it must only respond to the addition of new talking points (as per the assignment specifications). However, This approach was not implemented, as it increases logic complexity and reduces code reuse.

- **Alternative Design #2: replace MVC with simply M←→V**

In order to decreasing coupling and increasing cohesion, the MVC architectural pattern can be used. The MVC pattern separates the functionality concerned with data input and output from the manipulation of data to the storage of data. In the current implementation, I used the MVC model to separate the functionality as mentioned. However in my current implementation, the controller quite often simply passes the data to the model with little to no complex modification. This suggests that the system might not need a controller at all and simply allow the view to know of the existence of the model and the model to register the view as an observer in its implementation of the observer pattern. However, In order to eliminate the controller from the implementation, the model must hold a reference to the observer factory which is used for creating instances of PersonObserver and PolicyAreaObserver. Furthermore, any data manipulation that existed in the controller must be shifted to the corresponding method in the model.