Design Patterns chosen

Model-View-Controller

MVC is used to structure the Election Campaign Manager. All the basic data: people, policies, keywords, talking points, notification configurations, keyword observers and talking point observer interfaces are part of the model since these are just data and real-world concepts that the program deals with. The view contains KeywordViewer, TalkingPointViewer, MemberViewer, PolicyViewer, Viewer interface, PostMonitor, twitter and facebook messenger, sms, and menu, which helps display the data of some form and takes in user input. The controller contains PolicyController, GroupController, MemberFactory, PolicyFactory and NotificationHandler.The role of the controller classes is to interpret user input then handle the models or the views if required.

As shown above mvc ensures separation of concerns, which in turn leads to less coupling and higher cohesion since each section deals with a specific task thus improving Testability, maintainability and extensibility.

Dependency Injection

Dependency injection was used throughout. Nearly all Objects imported other Objects through constructor parameter or were set by setter if required. This reduced the amount of hardcoding when instantiating Objects with the "new" keyword. While reducing raw instantiation, it also allowed more Object reuse for example the notificationHandler could import the same GroupController that is also used by other parts of the program rather than create a new one. Maintainability is significantly improved since dependencies can be plugged in or disconnected or replaced with ease. This also helps testability by allowing mock objects to easily set up connection with the code. Overall this reduces coupling as a result.

Composite pattern

The composite pattern was used to organise required data, people such as volunteer, Strategist and Candidates, and Policy Areas which consists of many policies. The composite pattern was an appropriate pattern to represent the required data since we are dealing with many individual data that may be part of a heirachy thus can be grouped. For instance volunteer, strategist and candidates in this system are members of this system thus can be grouped into a group of members since the group composes of the members. The same can be said about policy areas, since many individual polcies can be used to form a directory of policies. In this system implementation, the Group.java is the composite class which contains a list of members of the Member.java interface(the node), Group.java also inherits Member.java. The leaf node class is implemented as an abstract Person class which the above type of people inherit from, Person inherits from the Member interface as do the childs classes.

For policy areas implementation PolicyAreas.java is the composite class which contains list of entries of the Policy interface in order to form a policy directory, the former also inherits this interface. The Leaf node class PolicyEntry.java represents an individual policy which inherits from the Policy interface since it is Policy. The use of composite pattern fulfills the requirements of managing, adding, remove, finding etc. of required data due to the heirachy-tree-structure property the composite pattern is based on, allowing recursive looks and insert/deletions of class fields of each node, the user doesn't need to know specific types when managing the data, which makes it easy, and as a result leads to reusable code since composite and leaf nodes both inherit from a common node.

Observer Patten

Observer pattern has been used extensively throughout this program since we dealing with a lot changes to data made by the user or a source. KeywordViewer, TwitterPostScout and FacebookPostScout inherit from KeywordObserver which observes events dealing with keywords, TalkingPointViewer inherit from TalkingPointObserver which observes events dealing with talking points. The subject that signals the observers above is PolicyAreas as keywords and talking points must belong to specific policy. All the observers have update methods that updates each observer keywords or talking points, however each observer handles the data differently depending on what they are designed for. The viewers display the keywords or talking points to the user. TwitterPostScout and FacebookPostScout set the keywords into a search map specific to each social media platform that is used monitor trending keywords from media posts.

Singleton pattern

Singleton pattern has been used for the Thread that monitors keywords in tweets and posts, and also for storing notification configurations. Singleton pattern is a very appropriate pattern for storing configurations since we don't need many instances of the configuration since this could be hard to manage many configuration due to variation and changes of same type of instance. Having one allows better management of a particular configuration for parts of a program that require that specific configuration. For the monitor thread, specifically for this program, the singleton pattern is implemented to allow the Facebook and twitter template classes to set search maps for the monitor threads from the sets without having to own the singleton, just access. Also just to avoid race condition by just having only one thread dealing with the searches.

Strategy pattern

strategy pattern is used for the KeywordViewer, PolicyViewer, MemberViewer and TalkingPointView classes. The strategy pattern was an ideal pattern for these classes is due to the fact they are all essentially classes that display data to the user but the data is specific to each class therefore are different strategies to

display data. All classes above inherit from the Viewer interface specifies that all child class must some how display data. The goal of this is to reduce coupling by not allowing the menu interface to fully know about inner workings of each viewer, just has to know that it can display data of some sort.

Factory pattern

MemberFactory and PolicyFactory are based on the factory pattern. The Factories handle the creation of Members and Policies respectively, it also handles which types each member or policy should be when instantiating, the user does not need to know the specific type that is instantiating. This also helps promotes dependency injection and reduces coupling throughout the system since the factory instance objects that can be passed used with construction elsewhere in the system. For example the models PolicyAreas and Group do not need to create the object when adding to collection, just pass an already instantiated object.

Template method pattern

Person class, which was used in the composite pattern as a leaf node, is also based on the template method pattern as a Person in this system can be a Volunteer, Strategist or Candidate, however they are still people thus having the same fields such as phone number, name etc. The use of the template pattern also allows more implementations of different types of people if needed without redudant code. Hook method overriden is the toString() as this is different for each subclass.

Twitter and Facebook Messenger classes are also based on the template pattern, each concrete class of the messenger implements and overrides the keywordsDetected hook method, the setKeywords can also optionally be overriden. In this system, the functions given are used to help search for trending keywords on posts, however if needed, another implementation of concrete class could be used for a different purpose such as creating a post from keywords that are trending.

Design Alternatives

Decorator pattern for people data

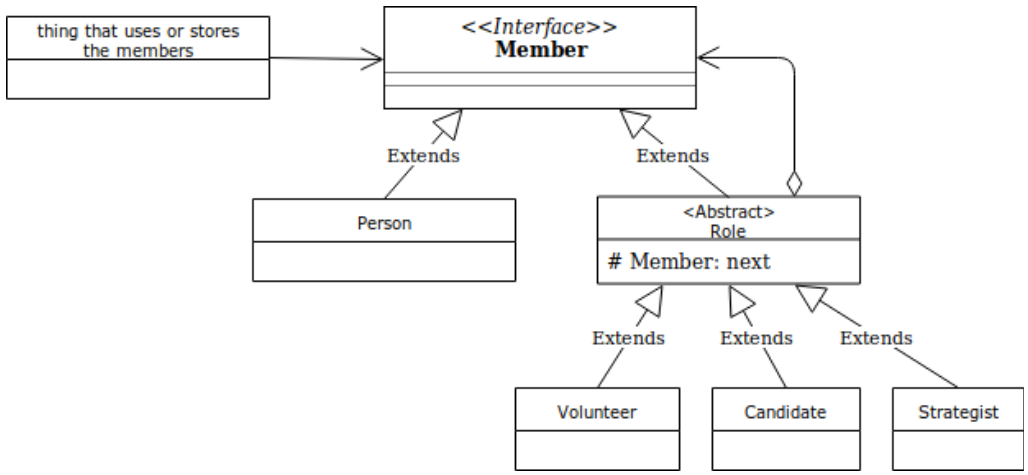
The decorator pattern could have been used to implement the types of people: volunteer, strategists and candidates. The types of people can be seen as roles, which are decorated on to a person since a person can be volunteer, strategist or candidates or if we really wanted to, a combination of these roles. In practice, role Objects will wrap around Person Objects to form the types of people with specific roles.

Pros:

- Decorator can be extended or added if more roles are needed.
- Reduces coupling due to utilisation of recursive aggregation more rather than inheritance, thus may allow better testability and maintainability.

Cons:

- Decorator isn't really used to store sequences of data in comparison to the composite pattern thus need to hold individual data in another class or part of program.
- Lots of wrapping around object can make instantiation more complicated.



Decorator uml

Template Pattern for the viewers

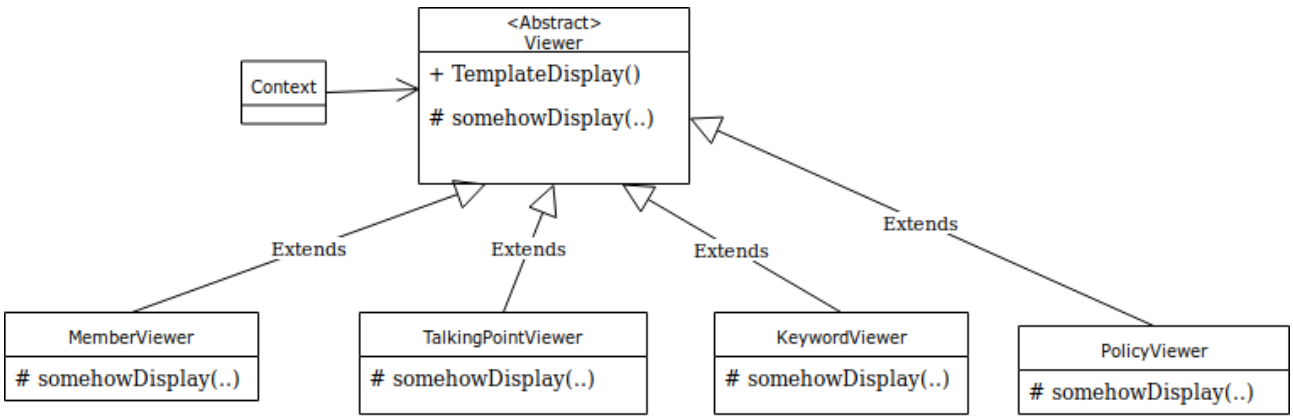
The KeywordViewer, TalkingPointViewer, MemberViewer, PolicyViewer could have been used instead of the Strategy. All the viewers share similar data such as Sets of string containing names, keywords and talking points, which are all essentially text. Algorithms to display the sets are fairly similar given the use of for each loops to iterate through the sets and printing. Each viewer could also override a hook methods if needed to peform their own specialised task. Parent class' template method usually uses the hook methods, template method shouldn't be overriden by subclass.

Pros:

- Can improve code reuse by reducing redundant code and algorithms. Subclasses should not need to rewrite the template method
- Hook method only need to be changed or implemented in subclass.

Cons:

- due to inheritance the concrete classes are more tightly coupled relying on base implementation in parent class.
- In java cannot extend multiple parents if it is required
- Unlike the strategy pattern's freedom to change and define different algorithm steps, the template method isn't supposed to be changed by sub classes. A subclass may need to perform the same function but with a different approach compared to the template method, changing hook method may not help this situation either. Therefore too much reliance on parent class.



Template method pattern