Explanation of key aspects of the chosen design

My program has high level of cohesion as each method does 1 task and because of high level of cohesion, there is some degree of coupling between methods as some operations in the program would need to do more than 1 task therefore I split them into separate methods eg: UserManipulatedOption.doOptions would need to rely on the hook methods and Menu.run would rely on doOption of MenuOption. I believe my program has high degree of reuse as there is little repeated code throughout the source code.

All user input and output are performed by UserIO class. Menu class is practically my main controller class as it holds classfields of MenuOption, Person and PolicyArea. I have chosen to use Maps as the classfield for O(1) searching there are 4 Map classfield for person, policy area, keyword, and talking points. The reason for 4 Maps is because the MenuOption can quickly report to user if any errors are to occur such as policy area does not exist and I can easily find the corresponding policy area for any keyword or talking point and perform my menu options on that policy area without needing to loop through all policy areas just to find which policy area does this keyword or talking point belong to.

For the purpose of using 4 maps, I made keywords and talking points to be unique in the sense that they can only exist in one policy area and will restrict users from trying to add the same keyword/talking point to another policy area so that I can have a keywordMap and talkingPointMap with keyword and talking point as the key respectively and if the keywords and talking points were to be not unique, generating the total set of keywords to be monitored would be difficult and would need 2 loops to cycle through all policy area and check if this specific keyword exists in this policy area. Person ID is guaranteed to be unique therefore it is the key for personMap and name of policy area would be unique as well so it is the key for policyAreaMap.

FacebookMessenger and TwitterMessenger are stored as classfield in Menu as well so that the set of keywords monitored for occurrences can be updated after every option performed and by using polymorphism in the menuOptionMap, the Menu won't know which option would modify keyword map and having the keywordMap allows me to not need to always loop through all policy areas to find the updated set of all keywords.

I have used a Keyword class to store the keyword, its trending status and timestamp since trending so that notifications on keywords trending can be off for 24 hours if it continues to be trending until then. I designed mine to send another notification if keyword was trending but no longer is trending from previous search but becomes trending again in the current search.

I have used the Observer pattern for notifications since notifications only happen when an event happens and doesn't need to continuously check for changes and simply have the event source being Policy Area to notify the observers of the particular event type since Policy Area would contain all of the keywords and talking points and circumstances for notifications happens when a new keyword is added, keyword is trending or a new talking point is added

Policy area has 3 maps of observers for the 3 different circumstances where a notification can occur and 3 interfaces are used as for modularity it is not guaranteed that each circumstance would take the same action. The person object is responsible for adding the appropriate observers to policy area by using ObserverFactory to construct the observer objects and ObserverFactory has classfields of SMS, FacebookMessenger and TwitterMessenger so that the observer subclasses can gain access to the given social media classes as a classfield without having to pass those social media classes to

person thus reducing what the Person class can see therefore better separation of concerns. With 3 maps, whenever the specific circumstance happens for a specific observer, I would simple call notify on all observers in that map for that event type.

When viewing notifications, the mapping of person ID and policy area name is shown for simplicity and to not flood each line with lots of detail that may be hard to read.

KeywordMapGenerator is simply a class that runs on its own thread by extending TimerTask and using timer.scheduleAtFixedRate method. It is in the same package as the given social media classes so that it has access to the protected keywordsDetected method and KeywordMapGenerator can pass the maps with the key being the keywords from the set of keywords obtained from the FacebookSubclass and TwitterSubclass and the value is the number of occurences which is generated using a random number generator for the purpose of showing that the assignment works as intended. keywordsDetected method calls KeywordMapCombiner's set methods to update the map from FacebookMessenger and TwitterMessenger so that KeywordMapCombiner can combine the map from Facebook and Twitter and pass that map to Menu so that Menu can pass the occurrence information to the keyword in its respective policy area and update the keyword trending status and whether to send out a notification regarding keyword trending.

TwitterSubclass and FacebookSubclass override the setKeywords so that set of keywords can be stored as we can't modify the declarations in the given social media classes, and also have a getKeywords method for KeywordMapGenerator to access the set of keywords.

I have used a Strategy pattern with a Template pattern in the Strategy pattern for MenuOption and its subclasses because all of the menu options would be performing one task which is the option selected. SaveToFile and ReadFromFile class simply implement the MenuOption since they would already do 1 distinct task without repeated code. For add, view and remove options, I used Template pattern within the Strategy patern with an abstract UserManipulatedOption because add, view and remove would require the selection of a subject (person, policy area, keyword, talking point, notification) and methods to perform option selected on a subject wouldn't require all 4 maps imported from Menu so I have 4 abstract hook methods, one for every subject and UserManipulatedOption would implement the doOption method from MenuOption interface and request user input for subject to perform option on and call the appropriate hook method. There would be 3 subclasses of UserManipulatedOption being AddOption, ViewOption and RemoveOption since these 3 options would require more user interaction and similar code than the File IO related options. MenuOption and all of its subclasses are in controller since they simply just control the flow of the program and communicate to the View classes and Model classes.

I used an abstract Person class as all person would have similar classfields and the type of person is not stored as it would be implicitly implied by the subclasses. There is abstract method called subscribeToPolicyArea in Person so that the subclasses can override and put down their implementation of subscribing to a policy area since each type of person would only be notified of certain event types that happens to a policy area.

There are static factory methods in Person class and PolicyArea class so that the MenuOption subclasses don't need to know how to construct those objects or determine which subclass to use for Person but just need to pass the necessary information to the factory methods thus improving separation of concern.

Design alternatives

The first design alternative would be to not use Template pattern in the Strategy pattern in MenuOption and its subclasses and get rid of UserManipulatedOption, AddOption, ViewOption and Remove Option. Instead, all possible UserManipulatedOption subclasses and hook methods from the Template Pattern would be a subclass of MenuOption. There will be more subclasses where its super class is MenuOption. The extra individual classes that would be introduced would be AddPerson, AddPolicyArea, AddKeyword, AddTalkingPoint, AddNotification and the respective View and Remove classes where all of them implement MenuOption.

The way to find the correct MenuOption in menuOptionMap in Menu would be a combination of integer entered from selecting option and the integer from selecting a subject to perform option on eg: 1-Add and 3-Keyword are selected so it finds the value of 13 (1*10 + 3) in the map and calls doOption method of that object retrieved from the map.

This design alternative would allow for any future changes such as adding in more menu options to program such as changing information of an existing person instead of removing the person and creating a new one just to change information and not need if statements to call the appropriate method as how it is done with the original design of using Strategy and Template together. However, it would require a more complex way to identify which MenuOption object to use for performing menu option operations and all subclasses of MenuOption would import all 4 maps but only one or half of the imports would be used in some of the subclasses.

The second design alternative would be to use Sets instead of Maps for the classfields of Menu so that there is less class fields in Menu and import parameters in MenuOption.doOption method to maintain and save space. By using Sets, you don't need to worry of forgetting to remove a mapping of keyword or talking point to a policy area but additional methods would be needed in MenuOption subclasses when that subclasses needs to deal with keywords and talking points and keep track if this keyword or talking point already exists or not. With additional methods in MenuOption, this would increase coupling as more methods are dependent on by the doOptions method. There will also be less repeated data such as multiple maps that have reference to PolicyArea objects as in the original design will be gone.

This design would be good if the system to run this program on were to have very limited memory at the cost of speed since it would require more looping to find the correct keyword and talking point while compared to the first design alternative and the original design where they prioritise speed than space.