Task 2:

Design Rationale

Observer Pattern

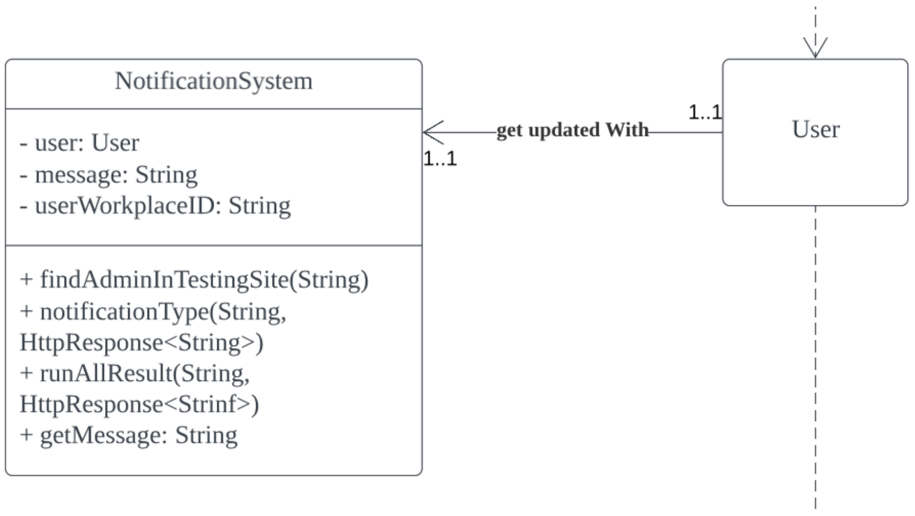
In this extension of the COVID Booking and Testing System, the team is required to extend it such that the system will automatically notify all the admins working at the testing facilities of the updated booking (including cancellations) in real-time, as well as the other testing facilities if the change is relevant to that facility. Being said, a few problems and points should be addressed here:

- This extension would very likely introduce a great degree of dependency among the users as well as the classes created to do the actions, that will ultimately notify all relevant admins and testing sites' personnel.
- When a booking changes state, all relevant admins must be updated according to the change of the state, during run-time.

With that out of the way, the team has decided to design a notification system that is inspired by the observer pattern. It should be noted that the team designed not with the conventional design pattern but a slightly modified version of it which will be explained in the following passage.

Using the terminology of the mentioned pattern, the team has identified that the users are the so-called "subscribers" whereas the notification system and booking modification entities are what they termed as the "base publisher" and "concrete publisher" respectively. In the conventional Observer Design Pattern, the publisher would store all the subscribers so that when an event happens that concerns the related subscribers, the publisher would call the notification methods on the objects. However, these subscribers are stored in an API web service (i.e. fit3077 API Web Service) which requires the team to adapt to the situation.

Being said, the team decided to introduce a notification system class (i.e. *NotificationSystem*) in the User base class (i.e. *User*). This is because when a user is modifying/deleting/cancelling a booking, it is required to have the user's information (e.g. The testing site he or she works in, that is if he or she is an admin) so that the relevant users can be notified by the event. When the relevant event happens, the concrete publisher (e.g. *BookingModification*) would execute the base publisher (i.e. *NotificationSystem*) and notify all relevant parties.


NotificationSystem
- user: User
- message: String
- userWorkplaceID: String
+ findAdminInTestingSite(String)
+ notificationType(String, HttpResponse<String>)
+ runAllResult(String, HttpResponse<Strinf>)
+ getMessage: String
get updated With
1..1
1..1
User

It can be speculated that this design does not introduce heavy coupling when a user notification feature is built in, with the aid of the design pattern of course. The only issue that may occur with this design is that the system's complexity would increase, ultimately affecting the performance of the system. But this should be expected when the feature is implemented and it can be mitigated by carefully implementing the design.

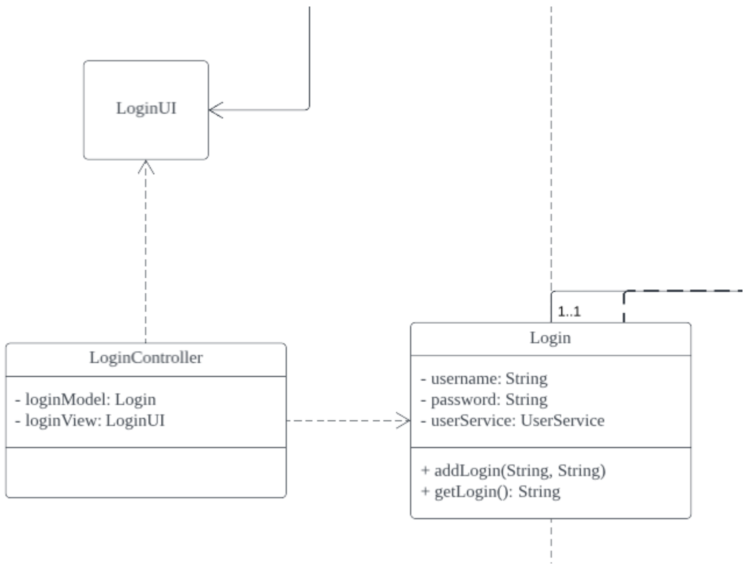
Model-View-Controller (MVC) Design Pattern

With the release of the further development that the team has to design and implement, it comes to our attention that the user interface would have an amount of changes that the team did not expect from how manageable the extension requested. The issue with this is that when the user interfaces and business logic are heavily dependent on each other, one would very likely have to change if the other was to change as well (making the system less robust).

Instead of ignoring this phenomenon, the team has decided to apply MVC design pattern to delineate the responsibilities into three categories:

1. Model
   - Manages business logic and data of the system.
2. View
   - Presents content through user interface, and has minimal logic.
3. Controller
   - The middle layer that intersects between the model and view. It accepts input and converts them into commands for the model or view.

It was identified that the COVID Booking and Testing System has a few components that can be benefited by this framework such as the classes and user interfaces related to the booking mechanism, testing mechanism, login mechanism etcetera.

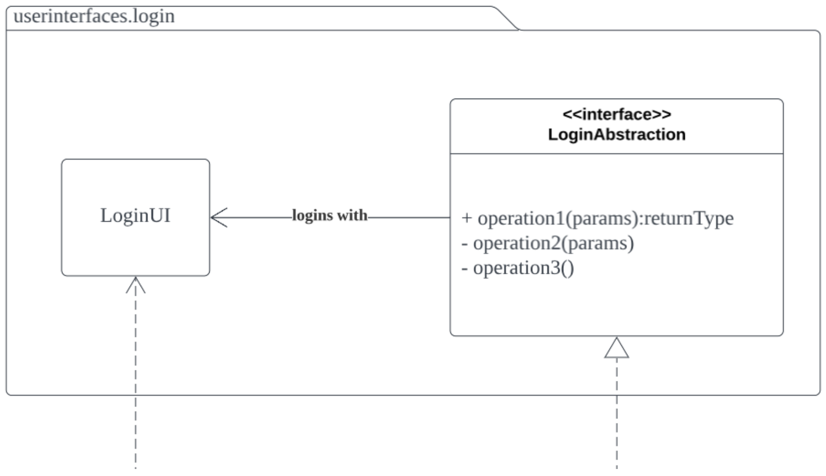


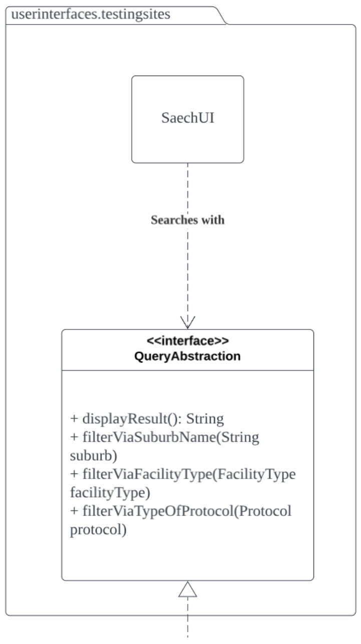
By alienating the responsibility of the three components mentioned above, within the mechanisms, the system would have higher potential to scale according to complexity, leading to a system that is easy to code and debug. Furthermore, this framework being applied to our design may have some disadvantages but they are mostly outweighed by the advantages of using it. The only disadvantage of the design is that it may be hard to adapt as the components are strictly separated and some models are encouraged to minimum logic (i.e. View) but again, they can be neglected.

The only further improvement that the team can suggest for the future is that the team can consider further alienating the logic in the View components into other components such as View-model. This is because despite the effort to separate the concerns of the components, the team found themselves needing a great deal of logic to display data from a complex model.

Common Closure Principle

Since the last extension, the team has realised that the interface that exists for the purpose of applying dependency inversion principle can be compartmentalised into components to better interconnect these packages. The interfaces used that allow the user interface to interact with the business logic with their respective interfaces can be grouped into the interfaces. This adheres to the common closure principle where it states that classes that change for the same reason should be grouped together. To further elaborate on this, it was observed and stated in the last paragraph that the user interface has a lot of changes when business logic is updated, thus, it would make sense to modify the interfaces that are related to the mentioned logic.




userinterfaces.testingsites
SaechUI
Searches with
<<interface>>
QueryAbstraction
+ displayResult(): String
+ filterViaSuburbName(String suburb)
+ filterViaFacilityType(FacilityType facilityType)
+ filterViaTypeOfProtocol(Protocol protocol)

Citation

1. Observer. Refactoring.Guru. (n.d.). Retrieved May 27, 2022, from https://refactoring.guru/design-patterns/observer
2. Wikipedia contributors. (2022, February 6). Observer pattern. Wikipedia. https://en.wikipedia.org/wiki/Observer_pattern
3. Learning Python Design Patterns - Second Edition. (n.d.). O'Reilly Online Learning. Retrieved May 27, 2022, from https://www.oreilly.com/library/view/learning-python-design/9781785888038/ch06s06.html
4. A. (2022, March 26). Overview of ASP.NET Core MVC. Microsoft Docs. https://docs.microsoft.com/en-us/aspnet/core/mvc/overview?WT.mc_id=dotnet-35129-website&view=aspnetcore-6.0
5. Wikipedia contributors. (2022, May 7). Model–view–controller. Wikipedia. https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller
6. Citu, A. (2018, January 12). The Common Reuse Principle –. Adventures in the Programming Jungle. https://adriancitu.com/tag/the-common-reuse-principle/