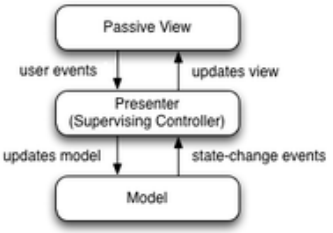
Running The Application

Please read the README.md in the code repository to start the application locally.

Architectural Changes



Team-34 decided to implement a Model-View-Presenter (MVP) [1] [2] [3] [5] [6] architecture in the booking system. In this architecture, the model is the interface which defines the attributes & functionality of the application and the view consists of the components which display the data. The presenter retrieves data from the model and passes it to the view for display. It also works in the other direction, handling user actions in the view and applying changes to the model, which are then propagated back to the presenter and then the view.

To implement this architecture, we refactored our codebase into three primary packages which reflect this: the DataModel package into the model package, the components package into the view package and transforming the DataModel facade into a more robust presenter package. UI specific functionality was extracted from the model and put into the presenter to follow the pattern, such as handling precondition checks for creating bookings, allowing for better SRP. This allows the model to define the attributes and specific functionality of the application and the view to be without logic.

The presenter package is further separated into different presenter interfaces and implementations based on the SRP. For example, making booking data and functionality available to the view is separated away from test site data and functionality. Thus, the interface segregation principle is followed (ISP) as components in the view can depend on different presenter interfaces depending on what resources they need to access. This improves maintainability since items which change together are housed together and thus when improvements are to be made, the improvements can be done in a single area. This also allows for extensibility, as new presenter functionality can be housed in a new presenter class, without requiring interaction with previous presenter functionality.

Furthermore, a data layer which stores the current state data is implemented between the model and the presenter. This follows the SRP since the state of the application is kept separate from the presenter and the model. Thus, the presenter component focuses on retrieving data from the API through the model and storing it in the data layer for display by the view while pushing handled user actions from the view to the API through the model. Each presenter implementation depends on a data layer interface specific to the data which the presenter manipulates. This follows the interface segregation principle (ISP) as the presenters only depend upon the data layer interfaces which they require, instead of having to depend on all of the functionality and data of all data layers. This highlights the maintainability of the data layer, as implementations of each interface do not need to implement so many functions. All the data layer interfaces are grouped together into a single *DataLayer* interface which is implemented by the *DataLayerProvider*. This class maintains the application state throughout the application lifecycle and provides the data layer to all of the providers of the presenters.

A disadvantage of having the application state stored in a single class is the possibility of a God class. However, having various data layers for each bit of information would introduce unnecessary complexity since a parent class would still have to manage all of the individual data layers. Therefore, the compromise was chosen where the data layer is separated from the presentation layer, but stores all of the state information. Furthermore, the design is still extensible due to the separated data layer interfaces; if another layer is to be introduced, a new interface can be added and implemented in the *DataLayerProvider*.

The MVP implemented follows an active variant, where the React Contexts [4] are utilised to provide the data and functionality throughout the view. When updates are made to the model, the changes are actively propagated back to the view through the presenter without the view having to call for an update. This makes the model completely independent from the presenter and the view, with the presenter component dependent on the model while providing the data and functionality to the view through the React Contexts. The active characteristic of this MVP design architecture was a huge reason for choosing to refactor the codebase to fit

this pattern. The active variant allows reusability of asynchronous retrieval of model information in a single location (the data layer), so that results from handled events can be seen immediately in the user interface.

Presenters are able to call for updates to the other presenters' data from the API because of the *BaseDataLayer*, which is extended by all other data layer interfaces. This defines the method to update all other data from the API and is implemented by the *DataLayerProvider*. An updater class was designed to keep track of when to update information, which is available to each presenter for their relevant information. Whenever a presenter updates information in the API, the presenter calls the update method provided to it, which calls each *Updater*'s update method, thus causing all presenters to re-fetch data. Separating the updater class away from the data layer and presenters aids in modularity since it is easily reusable, and maintainability since the data layer and presenters do not have to worry about the implementation of the *Updater*, only that it can do certain things.

This architecture was chosen because a number of design principles are followed. The SRP is followed since each component is cleanly separated from other components based on specific functionality, discussed above. Furthermore, the architecture allows each component to be closed against outside changes; changes that affect one component only affect that component and no other component (CCP). The primary advantage of the SRP and package cohesiveness is maintainability; when requirements are updated, the updates only need to be done in a single component. For example, when UI design requirements are changed, updates only have to be made in the view package, with no changes in the presenter or model required. The other advantage of the MVP is the added extensibility. New functionality can be easily added since the purposes of each component are well segregated, yet with a clean flow of execution between them. Furthermore, there are no cyclic dependencies in the MVP, as the components pull and/or push from another component only; the view pulls information and pushes events to the presenters, which pulls and pushes information to the clients in the model, while using functionality from other packages in the model.

One main disadvantage of this system is its added complexity. The design has been further segregated into different components, each with a specific purpose and function. However, this disadvantage is outweighed by the maintainability and cohesiveness benefits of the system. The other disadvantage is the cost of frequent updates; whenever the user makes an action through the view, changes associated with the action are pushed back to the view, causing an update. This has the potential to cause performance issues with re-rendering of the view. However, the asynchronous nature of the active MVP is still better than having the view poll for updates (which would be very expensive or cause non up-to-date information to be presented) or having the model depend on the presenter to push updates (which would break its independent nature), therefore being an acceptable solution.

The Active Model-View-Controller (MVC) architecture was considered as an alternative to the Active MVP utilised. The primary difference is that the view in our Active MVP retrieves data through the React Context in the presenter, whereas in the Active MVC the presenter implements an observer which is depended upon by the model to push updates. Both variants would have been applicable to the system. However, we did not want to have the model depend on anything, only depended upon, but still wanted to have the ability to asynchronously update the view when events were handled (this ability is explained previously).

General Refactoring

One code smell that was present within the codebase of A2 was where constructors of classes had very long parameter lists since they had many attributes. Because not all of the attributes were always present within the classes, the constructor was very difficult to use. Therefore, a refactor occurred where the parameter list of the constructors were shortened to only have required parameters, such as name or ID. The attributes were then able to be set using setter methods. The main benefits of the refactor are the better code readability and easier use of the constructors.

Another code smell found within the codebase was speculative generality, where there were many getters and setters that were not used. A refactor occurred where these unused methods were removed. Similarly, this improved the code readability.

Refactors that are more specific to a piece of functionality are explained throughout the document.

Added Requirements

1: Booking Modifications

1.1: Resident Booking

1.1.1: Modifying and Cancelling Bookings

Modifying and cancelling a booking required new methods to be created in the *BookingClient* interface for updating bookings. This method was then implemented in the *AxiosBookingClient*, making PATCH requests to the API. Designing the API calls to be made separately from the rest of the application in Assignment-2 (A2) proved to be beneficial as the package cohesiveness and SRP meant that the new feature was easily added without affecting other packages. Another benefit as a result of the SRP is that the API update method in the *BookingClient* is able to be reused for both cancelling and modifying a booking. A new method to handle user input for changes in a booking was also required in the *BookingPresenter*, which was implemented within the *BookingPresenterProvider*. However, the pre-existing *Booking* class within the model was not required to be updated, as the presenter is able to check the inputs and update the *Booking* using the attribute setters in it. Refactoring the form precondition checks into the presenter and away from the model allowed the changes to be made in the *BookingPresenter* without affecting the *Booking* in the model. The extensibility advantage of the MVP is highlighted here as changes to each component can easily be made without affecting other components.

1.1.2: Searching for Booking with PIN or Booking ID

Searching for a PIN code was a feature present in A2 when admins verified bookings. The search logic is housed in the *BookingCollection* class in the model but the logic searching for a booking with a PIN code or a booking ID are separated. Previously, the combination of searching for a booking for both PIN and booking ID was done in the *VerifyVerifiable* view component. However, the refactor to the MVP means that user-experience (UX) logic is to be moved away from the view and into the presenter. Thus, the move method refactor principle was applied to move the functionality into the *BookingPresenter*. This allows the system's component structure to be aligned with the MVP.

1.1.3: Changing Booking to Previous Booking

The ability for users to change a booking to a previous booking was functionality which required use of the memento design pattern. In this case, the *Booking* class is the originator, which depends upon *BookingMemento* to save its state, and a *BookingCaretaker* class was created to house the mementos and allow for rollbacks to occur. Whenever updates to a booking are made, the *Booking* instance creates a new *BookingMemento*, which is stored within the *BookingCaretaker*. Then, the *Booking* instance along with the *BookingMemento* are saved using the update booking method in the *BookingClient* mentioned previously.

The main advantage of this design pattern is being able to restore previous snapshots of the *Booking* instance. Furthermore, the memento allows a history of the *Booking* to be saved without violating its encapsulation.The SRP is followed since the state is saved within the *BookingCaretaker*, away from the *Booking*. This helps to maintain cohesion within the package as well, since things that change together stay together (CCP). There are some disadvantages to the memento as well: the system may consume a lot of memory if a lot of mementos are created. However, this disadvantage is mitigated since only the last three snapshots are saved, as these are the only ones required. Furthermore, the memento being stored within the API on *Booking* update means that the history of the *Booking* is kept accurate.

1.2: Phone Booking

1.2.1: Searching for Booking with PIN or Booking ID

The functionality for admins to verify the status of a booking given a booking ID or a pin code is the same functionality for a single patient, except for all patients. Thus, the functionality mentioned in 1.1.2 was reused for this requirement. The same method in the *BookingPresenter* was able to be reused in the view. This highlights the reusability aspect of the MVP, where the presenter methods can be reused in different locations of the view component.

1.2.2: Modifying a Patient's Booking

The edit form in 1.1.1 was reused for 1.2.2, although an admin can edit any patient's booking, whereas patients can only edit their own bookings. This permissions functionality is explained later in the Admin Booking Interface. Similar to 1.2.1, the update booking method in the BookingPresenter was able to be reused for when an admin modifies a patient's booking. This further highlights the reusability aspect of the MVP.

1.2.3: Changing a Patient's Booking to a Previous Booking

The functionality for a booking to be restored to a previous state is the same as in 1.1.3. The ability to restore a booking to a previous snapshot is part of the edit booking form, which is used for both patients and admins. Again, the only difference here is that admins can restore any patient booking, whereas a patient can only use this feature for their own bookings.

2: Admin Booking Interface

The admin interface was built in A2 so that admins could see all users, and then navigate into users to see their bookings. Thus, only updates to the admin interface were required.

Admins can view all users since they have the receptionist or healthcare types. These types are checked upon login and can only go to this interface if they have these types. This functionality was implemented in A2 and is part of the *AuthenticationPresenter*, which was refactored from the *Authentication* facade from A2. In summary, the *AuthenticationPresenter* provides functionality for a user to log or register to the system and provides the logged in user information to the view. It uses the *UserClient* in the model to retrieve information from the API. Having the authentication functionality separated from the user functionality allows permission-specific behaviour to be separated from user functionality, which has to do with manipulation of user information. This design choice allows the SRP to be followed, and enhances maintainability as changes required for a presenter only have to be made in that presenter, and the changes would affect the entire presenter (cohesiveness).

2.1: View & Modify

Since the admin interface was already built in A2, the functionality for admins to view users' bookings was already there. The edit functionality was new to A3 and was discussed in 1.1.1 and 1.2.2.

2.2: Delete

The functionality to delete a booking once again showed the extensibility of the design in A2 and of the newly refactored MVP. Having the client interfaces separated from one another (ISP) made it very easy to understand where new API method calls were to be added (designed in A2). A new method was thus created in the *BookingClient*, thus requiring a DELETE request to be made to the API from the *AxiosBookingClient*. The delete method of the *BookingClient* is used by the *BookingPresenter* to delete a booking, functionality of which is provided to the view and is only accessible by receptionists.

The MVP flow is able to be seen again here; user clicks on a button, the event is sent to the *BookingPresenter* which makes a call to the *BookingClient*, then makes the request to delete the booking from the API. Once the request is asynchronously completed, the *BookingPresenter* refreshes the booking data from the API, updating the data in the data layer. Thus, the data provided to the view is updated, updating the view as well.

2.3: Notifications

The requirement for notifications required a new notification package to be created in the model. A new *Notification* class was created to house the behaviour and information of a single notification, a *NotificationCollection* was created to house multiple notifications and behaviour to manipulate and retrieve them, and a *NotificationCreator* was created to aid in creating *Notification* instances given information from the API. The creator method design pattern helps segregate the functionality of creating *Notification* instances from the class itself, strengthening the maintainability of the package. The package also has strong cohesiveness, since all three of these classes depend on each other and are reused together as they all have

to do with notification functionality. This therefore follows the common reuse principle (CRP), where classes that change together belong together; all these classes are to do with notifications and thus belong together. Notifications are sent to the API under the additionalInfo of a booking request. Being able to create a new package in the model shows the extensibility of the model package, where new functionality can be added without affecting previously existing ones.

When updates, deletes and cancellations of a booking are made, a *Notification* is created in the *Booking* instance's *NotificationCollection*. Furthermore, a notification is added to a *TestSite* instances' *NotificationCollection* if the test site has changed on a booking update. Thus, both the *Booking* and *TestSite* store *NotificationCollection*. The reusability of the *NotificationCollection* is possible because it is separated into its own class in a package of other notification type classes.

To view the notifications, the notifications for a test site and the bookings under it are combined into a single *NotificationCollection*, where notifications are sorted by date and then presented to the user. This is possible because of the *NotificationCollection*, which has methods for manipulating the notifications stored within it. This is the benefit of having the *NotificationCollection*, instead of simply storing notifications in lists; combinations and sorting is more efficient this way.

The disadvantage of the notifications being stored within the *Booking* and *TestSite* is that these classes now have more responsibility to look after. However, this is mitigated by having them only store a single *NotificationCollection*, such that they do not have to maintain the behaviour of notifications, only store them.

References

[1] J. Boodhoo, "Model View Presenter." Microsoft. https://docs.microsoft.com/en-us/archive/msdn-magazine/2006/august/design-patterns-model-view-presenter (Accessed May 26, 2022).

[2] *Unknown*. "Model-view-presenter." Wikipedia. https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93presenter (Accessed May 26, 2022).

[3] *Unknown*. "What are MVP and MVC and what is the difference?". Stack Overflow. https://stackoverflow.com/questions/2056/what-are-mvp-and-mvc-and-what-is-the-difference (Accessed May 26, 2022).

[4] React. "Context." React. https://reactjs.org/docs/context.html (Accessed May 26, 2022).

[5] G. Teles. "Let's talk about React and MVP." Medium. https://medium.com/pdvend-engineering/lets-talk-about-react-and-mvp-67ae35b8968c (Accessed May 26, 2022).

[6] baeldung. "Difference Between MVC and MVP Patterns." Baeldung. https://www.baeldung.com/mvc-vs-mvp-pattern (Accessed May 26, 2022).