Design Rationale

The first notable design choice in our system is the inclusion of the handler classes. This is an implementation of the Proxy design pattern which is structurally similar to the adapter design pattern [1]. The handler classes act as a proxy between the API classes and the rest of the system, allowing the other classes to interact with the API and the data stored on it, without needing to know how the data is actually stored and how it's accessed (ie. its implementation). This has the added benefit of making the system more modular and reducing coupling between data storage and usage. If, in the future, data was changed to be stored in a database, only the handler classes' implementations would have to change. The tradeoff for this implementation is that a whole new type of class had to be created (the handler classes), which complicates the design. There's also a loss in flexibility since the other classes can't make and interact with the API calls directly, though the handler classes should already provide all the features the other classes really care about.

Some alternative design patterns considered instead of the Proxy design pattern chosen for the API classes are the Facade and Adapter design patterns. The Adapter design pattern was discarded after some deliberation because the classes that deal with the API aren't incompatible with the rest of our code due to the fact that it will be coded by us in consideration of the rest of the software design [3]. There isn't an existing class with an incompatible interface that will be needed, only incompatible data (i.e. the JSON data returned from the API) that requires conversion into objects and the data conversion code needed will use existing classes which can be directly used by the rest of our code. The problem we were facing was that we wanted to separate the data conversion code from the actual handling of the API (code to call the patch requests etc.) to prevent the API classes from becoming god classes and to hide functionality that won't be directly used by the rest of the system.

The Facade design pattern was subsequently considered as it solves the problem of providing a simple interface to hide the unneeded functionality of the API classes [3]. The issue with this is after having the class diagram drawn out, we realised there isn't a complex subsystem in which the API "Facades" (i.e. API Handlers) were providing abstraction for. Each "Facade" only deals with the one API class it is associated with and this

design pattern more so resembles a Proxy design pattern than a Facade design pattern [1]. It's a wrapper that provides abstraction for the API classes' functionality and contains almost the same interface as the API classes with some additional logic for data conversion. For example, addBooking() in the BookingHandler has essentially the same purpose as postBooking() in BookingApi, in fact addBooking() will call postBooking() to add a booking into the database through the API. But, addBooking() will contain additional data conversion code and any necessary validation for error prevention and handling.

The separation of the various API and Handler classes was to comply with the Single Responsibility Principle. Separating the responsibilities of handling API calls for different types of data between many classes reduces the overall complexity of each class, and the reasons it might have to change in the future. The tradeoff again is that it increases design complexity, as more classes and dependencies need to be added.

One tangential choice here was the exclusion of some common API methods from abstraction. That is, some methods in the API child classes like getUsers() and getTestingSites() work almost identically, but they are still implemented separately, as opposed to having a single getItems() method in the API parent class. This decision was made with an eye to future extension. More specifically, out of the worry that the API might not support the same functionality for future data types, or changed the way some data was accessed, and the refactoring that would cause. For example, if the API one day changed to require getting a user by their username instead of their id. This does unfortunately cause us to repeat code across different classes that could have otherwise been written only once. But in this case, we concluded that the flexibility and extensibility was worth the redundancy.

Another notable design choice is the use of the Singleton design pattern, as specified in the week 4 lecture slides [2]. This design pattern can be seen in both the API and Handler classes. This design choice was made because there is no real sense in having multiple instances of an API or Handler class, since they would all be used to complete the same functions. Instead, we save memory by allowing only one instance of an API or a Handler class to exist at a time. However, this does violate the Single Responsibility Principle, since it both limits the class to a single instance, and provides the access point to that instance. [2]

The Factory Method design pattern was also used in the UserFactory class, though only lightly. This pattern was used to remove the complexity of choosing and creating the right user class from the rest of the code and instead have it be the responsibility of a single class, further supporting the Single Responsibility Principle. The downside of this approach is that it complicates the design and makes creating User classes less flexible. Our specific design follows closely to an example given by our lecturer [4]. We did not go further with the design pattern by creating separate factories for each type of user however, because we decided that it would overcomplicate our code for little gain. Although if say the Customer class gets separated into more specific subclasses in the future, a CustomerFactory class would make sense. Same goes for TestingSites.

Next, TestingSite does not actually store Booking objects within it, and Customer does not store Booking objects, but Booking objects store instances of TestingSite and Customer. This is to avoid circular dependencies as they can cause design issues [5].

Note that the API class would normally be abstract, but we decided to not make it abstract to allow for direct access to the api request methods within it should it ever be required in the future.

In regards to packaging our classes, we tried to strike a comfortable balance between CCP and CRP, only grouping together classes that are at least strongly related to each other. There is one outlier in the user_interface package however, but this is because it is a utility package that doesn't have much to do with the system itself, though it will probably be extended down the line as developers require more reusable, general utility methods.

We also tried to adhere as closely to the Stable Dependencies Principle as we could when designing our system, and were largely successful. This can be illustrated with the calculations below:

$$I = \frac{Ce}{Ca + Ce}$$

Package	C_a, Afferent Coupling (in-coming)	C_e, Efferent Coupling (out-going)	I, Instability Metric
api	3	0	0
handler	0	6	1
user	2	0	0
registration	2	1	0.33

Packages should only depend on those with lower I metrics than them [5]. And we can see that:

- api and user do not depend on any packages
- registration depends on user, which has a lower I metric than it
- handler, with the highest I metric, is not depended on by any class

No packages in our design depend on packages with higher I metrics than them. Therefore we can conclude that our design adheres to the Stable Dependencies Principle.

References:

[1] Kumar, S. “Proxy Design Pattern - GeeksforGeeks.” GeeksforGeeks. https://www.geeksforgeeks.org/proxy-design-pattern (Accessed 28 April 2022).

[2] N. Nazar. (2021). Software Design Patterns - I [PowerPoint]. Available: https://lms.monash.edu/mod/resource/view.php?id=9880736

[3] N. Nazar. (2021). Object Oriented Design Principles - II [PowerPoint]. Available: https://lms.monash.edu/mod/resource/view.php?id=9880751

[4] “Ed — Digital Learning Platform.” Edstem.org. https://edstem.org/au/courses/7600/discussion/808657?comment=1826812 (Accessed 28 April 2022).

[5] N. Nazar. (2021). Object Oriented Design Principles - I [PowerPoint]. Available: https://lms.monash.edu/mod/resource/view.php?id=9880724