Design rationale - Aaron, Brent, Harpo

Google drive link for system demonstration video:
https://drive.google.com/file/d/1BC3hV8ZYkCXveSRRnatuCViX-a79obR_/view?usp=sharing

We have explained our design choices first for each package, where we discuss SOLID principles and design patterns. Then by the entire system, looking at package principles. We have then concluded with additional design improvements for the future and any project assumptions.

COVIDbookings package

SOLID principles used: Single responsibility principle, open/closed principle
Design patterns used: Factory method

A major advantage of the **factory method pattern** in *COVIDbookings package* is that it provides an interface (*BookingFactory*) for creating objects of bookings that allows for the creation of subclasses *HomeBooking* and *FacilityBooking* based on application requirements.

This virtual constructor replaces the direct object construction calls of *HomeBooking* and *FacilityBooking*. These calls now occur in our concrete factory product *ConcreteBookingFactory.* This is great because it avoids tight coupling between the creator and concrete product classes. This also reinforces the **Single Responsibility Design Principle (SRP)**, because the product creation code is moved into a single place in the program. The benefit of this is it makes the code easier to support.

The factory method here also allows for easy extension of internal booking components. For example, if additional booking types outside of home or facility become available, like booking at a school, the existing creator and product framework is in place for expansion.

We have used the factory method across many subsystems of our system. A consequence of this is that it's introduced many additional subclasses, which has complicated our class diagram and the overall design of our COVID Registration System.

An alternative design pattern for this subsystem would be to use the **abstract factory method**. This allows the creation of a family of related products that contain several variants of each member. This pattern may be more appropriate in future design choices if our system requires the creation of booking objects that are even further specified. An example of this would be breaking *FacilityBooking* down even further, into *GPBooking, ClinicBooking,* and *HospitalBooking*, combined with walk-in and drive-through options. *GPBooking, ClinicBooking,* and *HospitalBooking* would be classified as the family members, with walk-in and drive-through options being the variants. This would be a highly expandable design option, as it would allow for additional booking types.

We decided against this in our current design because we wanted to avoid overcomplication. We instead chose to identify the type of facility booking in an attribute additionalInfo using boolean values.

Another future design choice that would promote scalability is to create a *GeneratePIN* interface that bookings such as *FacilityBooking* would implement. This would be advantageous if additional booking types were introduced that also required the generation of a unique PIN code, like a school booking.

Using the factory method here may also save system resources in the future, as the system can reuse the existing *ConcreteBookingFactory* instance to create new concrete products instead of having to rebuild it each time.

COVIDtestingsites

We adopted a simple design plan for our covid testing sites subsystem. The consequence of this simple implementation however is that it doesn't promote extensibility, is quite rigid, and does create dependencies with high level classes rather than abstractions which goes against the Dependency Inversion Principle.

However, design principles such as SRP were still followed as the *FacilityTestingSite* class was mainly responsible for storing data about the location and capabilities of a single site, and a *FacilityTestingSiteCollection* was responsible for collecting all sites in order to perform searches.

An alternative approach would be to introduce a creational pattern like the **factory method**. Similarly to how we used this in bookings and COVIDtests, this would have provided a framework for creating concrete products.

However, we did not employ any creational method because the scope of our project didn't require us to create new testing site locations, rather only interact with testing sites available through the server. We also currently only have one type of testing site, *FacilityTestingSite.* If in future designs, this expands to encompass home testing sites as well, the factory method would be more appropriate. At the moment, we handled home testing by using empty identifications for testing sites.

The reasoning behind having a single *FacilityTestingSite* which does not include home testing sites comes down to functionality. Home testing would not require fields such as address to be populated and wouldn't need as much functionality that testing sites would require. Furthermore, we included facility types and capabilities as attributes rather than individual classes because aspects such as booking processes and on-site testing don't differ between them.

This future design decision would involve creating concrete product classes *FacilityTestingSite &* *HomeTestingSite* that implement the *TestingSite* interface*,* and concrete factories *FacilityTestingSiteFactory, HomeTestingSiteFactory* that implement the interfaces *TestingSiteFactory* interface.

This implementation would improve the extensibility of our COVIDtestingsites subsystem, would better accommodate the addition of more testing types (diagnostic tests, molecular tests, antigen tests, antibody tests etc)

COVIDtests

SOLID principles used: Single responsibility principle, open/closed principle, Liskov substitution principle (LSP)
Design patterns used: Factory method

We have again implemented the factory method pattern here to facilitate the creation of covid tests. We have explained our reasoning in detail under the COVIDbookings package section of this document. Please refer to that section for our in-depth explanation. To avoid repetition, we have instead just provided a high level overview of our rationale here.

- Advantages: Easily extendable if more testing options are required in the system Reinforces SRP, avoids tight coupling between the creator and concrete product classes
- Disadvantages: Because we've used the factory method pattern a lot, its created many additional classes and has complicated the design of our system

A future design choice to improve our implementation would have been to include an Enum class *TestingType*, holding constants RAT and PCR. This would have improved our system's simplicity as there would be fewer child classes inheriting from *Test* which could lead to a more maintainable package. It is also important to note that different types of tests don't differ much in functionality for booking purposes and an Enum class *TestingType* would lead to significant reduction in repeated code arising from child classes.

Enum instances are also **singleton** and therefore reused. Our test objects would have an association with *TestingType*.

The child classes *RAT* and *PCR* extend *Test*. We've structured our design so that objects of these subclasses can be passed in place of *Test* without breaking the system. This is in line with the **Liskov substitution principle.** We've taken advantage of this principle in *ConcreteTestFactory,* a concrete factory class that's creating tests of different types (*RAT* and *PCR)* as *Test* type. This allows for better extensibility as mentioned before, since adding new tests won't require a lot of refactoring. Furthermore, as child classes can be replaced with their parent class or other child classes without breaking the program, it does suggest that code is more maintainable.

Users

SOLID principles used: open/closed principle, SRP
Design patterns used: Singleton, factory method

Our system requires user login verification. This authenticated user can then interact with the system in a variety of ways; book a test, administer a test, pick up a RAT kit etc. There can only be one logged in user at a time, and the system will require access to the authenticated user's information as they're interacting with the system. Therefore we applied the **singleton pattern** to our *Auth* class. This resolved these two requirements; singleton ensures there's only a single instance of *Auth* (a logged in user), and provides a global access point to that instance. A benefit of this is that our *Auth* object is only initialised when it's first requested (when the user logs in).

A major consequence of applying the singleton principle to *Auth* is it violates the **Single Responsibility Principle** because of its global access point to the instance. Singleton patterns also create testing difficulties that may arise in the future, as many test frameworks rely on inheritance when producing mock objects. In future designs this *Auth* object has the potential to cover bad design.

We've again used the **factory method** here to construct different *User* types. We've gone into great detail previously about the pros and cons of this pattern, so we won't repeat again. One benefit we've not mentioned yet is the adherence to the **open/closed principle.** Our *ConcreteUserFactory* promotes extension. We can easily include additional User types.

Menu & menuItems

SOLID principles used: Open/closed principle, SRP
Design patterns used: Singleton, factory method, strategy

To navigate our COVID Registration System we designed a menu that is operated through the console. This menu had to cater to different user types (*Patient, Receptionist, HealthcareWorker)* and facilitate many different system requirements (Home testing, on-site booking etc). We implemented the **strategy behavioural pattern** because it extracts these algorithms to achieve different system requirements into separate concrete strategy classes: *CreateBookingMenuItem, CheckBookingStatusMenuItem* etc. These objects implemented our strategy interface *MenuItem* allowing them to be interchangeable.

This allows our context class *Menu* to switch between different *MenuItem* strategies during runtime, a major advantage. Without the strategy behavioural pattern our system would have massive conditional operators that switch between variants of the same algorithm to achieve our console menu. This would result in a **rotten design** that's **rigid** and **immobile**.

The strategy pattern also provided the advantage of better isolating our business logic in a class and separating it from the implementation details. For example, *SearchTestingSitesMenu* houses functionality for customers to search testing sites, while the *Menu* provides the navigation and access.

In the context of our system, the strategy pattern has been largely positive. The only negative is that our menu objects (e.g. PatientMenu*)* have to be made aware of the different strategies (*e.g. CreateBookingMenuItem)* to be able to select the correct options. This means adding new menu items to these menu objects when they become available.

An alternative design choice would be to implement a GUI instead of our console menu. This would provide an increased user experience. However, this requirement was outside of this system's requirement scope, and difficult to achieve within our timeframe. This would be a future design choice consideration.

Endpoints

SOLID: SRP, LSP
Design pattern used: facade

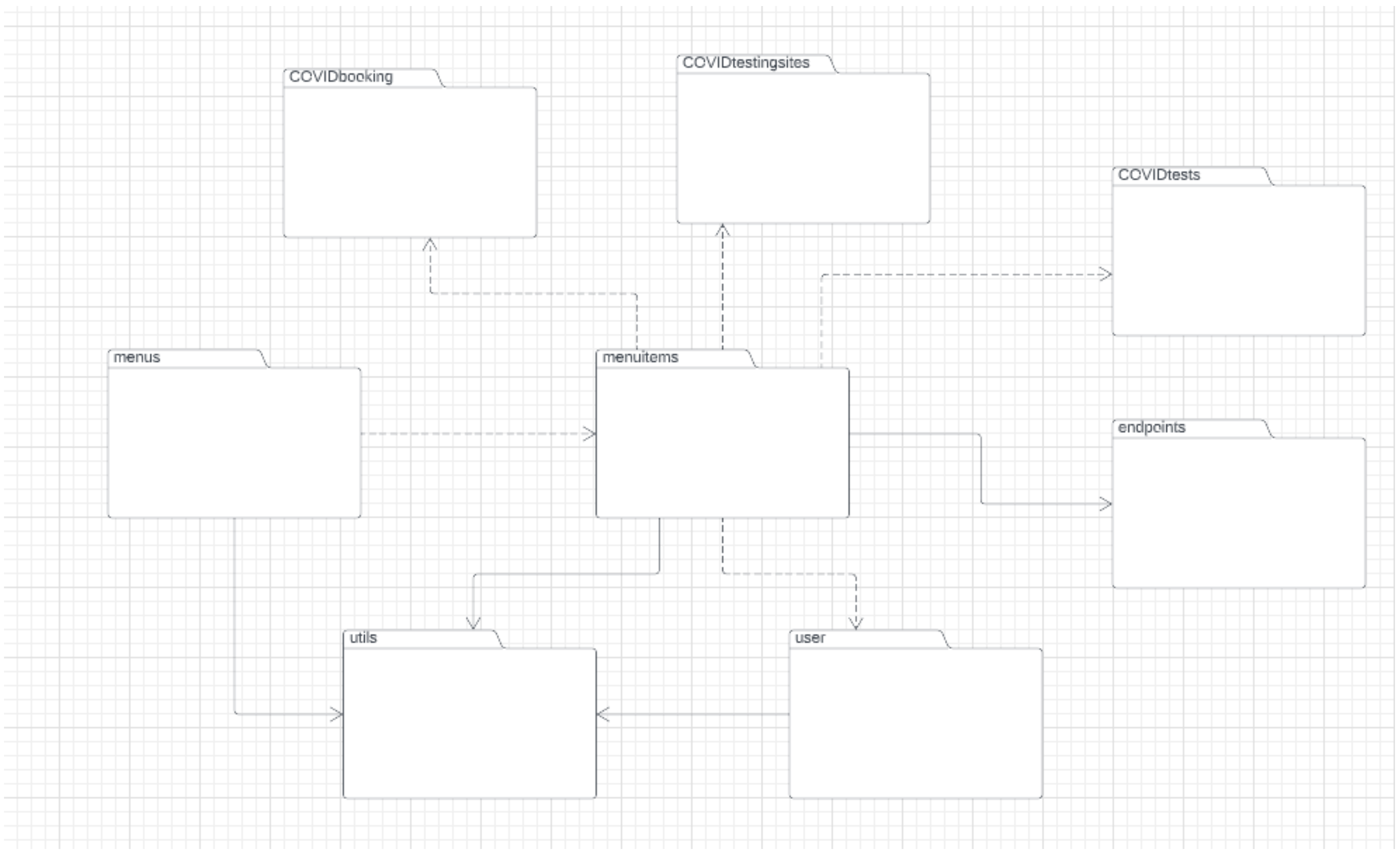
This package acts as a facade for the FIT3077 server. Providing the advantage of an easy to use interface that facilitates interactions with parts of the server we're interested in.

This package extracts implementation details away from the User. It allows for extensibility and changing details within the API. For example, if one of the URLs changes we only need to make a change in one spot.

SRP is used for each class. LSP also applies because each child endpoint acts like the parent. They are interchangeable.

System

Package principles: acyclic dependencies principle



Our design places a large emphasis on minimising package coupling. Our design is in-line with **acyclic dependencies principle** because there are no existing cycles in our design. This is advantageous because in a cycle breaking even just one class can then affect or break all other classes in that cycle.

In our design we also tried to include some stability to our packages. We tried to avoid packages having multiple dependencies going out and in.

As seen from the package diagram, we've heavily reduced coupling, meaning there's less potential for change to propagate.

Additional design improvements

- Create a *SearchableCollection* class to remove repeated code of searching through each list class (*UserList, BookingCollection, FacilityTestingSiteCollection* etc)
- List classes (*UserList, BookingCollection* etc) should be able to self populate from the server inside the class, not outside at construction
- Introduce ENUM class for
    - Testing site types; clinic, hospital, walk-in etc
    - Test types; RAT, PCR
- Improve security / privacy. Getpassword() method is currently public
- Create interfaces or abstract classes for groups of menu items, at the moment there is some repeated code around verifying user input

Assumptions

- Patient = customer
- Facility staff = Receptionist & HealthcareWorker
- No customer/patient menu item for picking up a RAT kit for home testing. This functionality is included in the facility workers menus *ProvideRATKitMenuItem*
- A testing site can support both drive-through and walk-in.
- A testing site can be any combination of hospitals, GP, and clinics. I.e. If a testing site is in a building with a GP on the ground floor and a clinic on the first floor, then that testing site would be considered both a GP and clinic.
- The default operating hours of a testing site are 9am to 5pm.
- The default waiting time for testing sites is 5min.
- There is no limit on the number of bookings a testing site can have, and there can be many bookings at a single time.
- User can only be a single type of user: patient, receptionist or healthcare worker
    - If an existing user is categorised as multiple types, our system assigns their type in this order: Customer -> Receptionist -> Healthcare Worker
- On-site booking assumptions
    - Customers can book for any testing site
    - Customers can book for home tests (this provides functionalist so a customer can book a home test for a separate patient)
    - Receptionist can also book for themselves