Assignment 2: Design Rationale

Creational Design Patterns

Abstract Factory

We used the abstract factory design pattern for Role and its subclasses AdminRole, CustomerRole and HealthcareRole.

- The client needs to work with various user roles (such as administrators, customers and healthcare workers), but the specific type is unknown until runtime when the user has logged in. By using this pattern, we solve this issue and avoid tight coupling between concrete roles and the client.
- This pattern promotes Open/Closed Principle since we can introduce new roles without breaking existing code.

We used the abstract factory design pattern for BookAction and its implementations OnlineBookAction and OnSiteAction.

- All bookings require the same basic data and functions such as inputCustomer() and inputSite(). These methods exist in the super class to avoid code duplication.
- This pattern promotes Open/Closed Principle since we can introduce new types of booking actions without breaking existing code.
- However, if there are no new types of booking actions (i.e. extensibility not required), this pattern may have led to unnecessarily complicated code through introduction of extra classes.

Singleton

We used the singleton design pattern for SiteCollection and UserCollection.

- The class needs to be accessed globally due to other classes depending on its stored data.
- There should only be one instance of the class being used at any one time. This class encapsulates a database snapshot at a specific time. All classes that rely on SiteCollection should be using the exact same data.
- If the program moves to a multi-threaded environment, this design may need to be revised or treated differently.

We used the singleton design pattern for Authentication.

- The class needs to be accessed globally due to other classes depending on its verification methods.
- There should only be one instance of the class being used because only one user can be logged in at a specific time.
- If the program moves to a multi-threaded environment, the design may need to be revised or treated differently.

Structural Design Patterns

Adapter Pattern

We used the adapter design pattern for Api and classes that implement its interface.

- The https://fit3077.com API returns raw data in a format that is incompatible with the client. As such, this data needs to be transformed into a usable format through an adapter.
- This pattern promotes Single Responsibility Principle since we separate the client logic from data conversion. The client should not be responsible to transform the data from the API.

We used the adapter design pattern for QRCode and classes that implement its interface.

- The generation of QR codes and conference URLs may require data in a format that may be incompatible with the client. For example, a JSON string. As such, the data needs to be fed into an adapter to be transformed into a usable state.
- This pattern promotes Single Responsibility Principle since we separate the client logic from data conversion. The client should not be responsible to transform its own data for use with a specific QR code or conference URL generating package.
- This pattern promotes Open/Closed Principle since we can introduce new adapters that follow a common interface. In the future, we may wish to generate QR codes using a different format (there are 40 versions) that needs data in a specific format. For conferences, we may wish to use a different conference service such as Google Meet over Zoom.

Facade Pattern

We used the facade design pattern for Menu.

- From the user's perspective, they should see a simple menu that hides the complex subsystems required for software functionality.

- However, as the functionality of the system grows in the future it is important to ensure that Menu does not become a “god class” that is coupled to too many classes in the system.

Behavioural Design Patterns

Memento Pattern

Though the memento pattern was not used, there is a reasonable discussion to be had regarding its viability in the design.

- The memento pattern provides recovery capability that could be useful in programs where a user inputs data that changes the state of the system (such as this).
- Recovery functionality is not a requirement stated in the specifications, so we concluded that it should not be included. However if it was implemented regardless, this would provide another avenue for future extensibility of the system.

Command Pattern

We used the command design pattern for Action and its implementations. In this pattern, each class only needs to implement name() and execute() to perform a given action.

- Promotes Single Responsibility Principle. Each Action subclass encapsulates all information that is required to perform a certain action (such as searching for a testing site).
- Promotes Open/Closed Principle since new actions can be introduced without breaking existing code.
- However, this pattern may lead to high coupling between a certain action and other classes. This may occur if a certain action depends on many parts of the system.

SOLID Design Principles (Not Mentioned Prev.)

Liskov’s Substitution Principle

All actions extend to the Action interface hence, the client can expect all actions to have the same basic functionality.

- As such, the client can use any Action without worrying about its concrete implementation.
- Reduces coupling from Roles to all required Action.
- Reduces the need for maintenance and modification of other classes when implementing actions.

Interface-Segregation Principle

Since there are many different user types (such as administrators and customers), we made sure the system does not expose unauthorised methods to users. We achieved this by using a Role class that depends on the API information provided about the user. These roles determine what actions the user can execute, so users performing unauthorised actions is not a possibility.

- Maintenance is easier as not all methods are grouped together (require less checking)
- Modification to authority levels (method exposure) is easy as only roles need to be changed not the action codes
- In this case, does not increase coupling regardless of how many roles as users are already coupled with the Role abstract class regardless.

Dependency-Inversion Principle

Role is a high-level class that represents complex business logic. In contrast, each action is a low-level class that implements basic functionality. A high-level class should not depend on a low-level class. Instead, Role depends on the Action interface which all actions will commonly implement.

- Reduces coupling between roles and actions.

Package Cohesion Design Principles

Common Reuse Principle (Thadisha)

All classes in a component should be inseparable and be used together. For example, when the system requires use of an ApiAdapter, they must use the entire ApiPackage. A similar design was applied when developing QRCodePackage and ConferencePackage. Since this is not violated, it would suggest that classes are contained in the package where they truly belong.

Common Closure Principle

In the ApiPackage, introducing a new method signature for ApiAdapter often requires modification of Api since there is an interface link between the two classes. As such, we have packaged these classes together. A similar reasoning was applied when designing the QRPackage and ConferencePackage.

- Improves maintainability as likely files checked are within the package during modification.

Package Coupling Design Principles

Acyclic Dependencies Principle

Acyclic dependencies occur when there are cyclical relationships in the dependencies between components of different packages. When considering the design of the system, we have ensured that no acyclic dependencies have surfaced when implementing the required functionality. This makes the codebase easier to work with since we can release components without encountering many dependency troubles.

Stable Dependencies Principle

Since TestPackage does not depend on other packages (and will most likely continue not to in the future) so it is very stable.

- Prevent a chain reaction of changes if a modification is made.
- Multiple packages can depend on it as this package will likely not change due to influence from dependent packages.
- However, it will be difficult to change this package as many packages are dependent on its current implementation.

$$I = \frac{C_E}{C_A + C_E} = \frac{3}{0 + 3} = 1$$

Stable Abstraction Principle

Under the current design, the TestPackage is difficult to extend because it is not highly abstract. As such, it violates the Stable Abstraction Principle. Despite this, the decision was made because currently every Test behaves the same and the TestType can be represented by an enumeration. As such, we concluded that there was no need for an abstraction of Test.

$$A = \frac{N_A}{N_C} = \frac{0}{4} = 0$$

References

https://refactoring.guru/design-patterns/command

https://flylib.com/books/en/4.444.1.166/1/

Miscellaneous Notes for Assessor

Task 4 was completed prior to the following requirements update: “The type of covid test can be placed under the additionalInfo covid-test field of the “Booking” object [updated 23/4/2022]”