We change part of our design and implement MVC. Our A3 is not completely starting from scratch. The A2 design is easy to extend new functions, and the original general design idea is very similar to MVC (the main logic is in the underlying classes - Booking, facility...). To extend a function, we only need to reuse the existing logic and recombine it into service class. So, service class contains all the logic of a function. We didn't change our top-level design (i.e., those stuff like main system, sub-system, menus) because it was already smart and flexible enough (it can easily help us implement user permissions changing, new features adding, hotkeys specifying, etc.).

In A3, we just changed that logic in the Service class to MVC, and completely tweaked our code to evolve from a pseudo-MVC to real MVC. There are three reasons for the change. First, in our original console-based system, we had to write a lot of logic to do validation for user's inputs, which was very cumbersome. If we have an interface, we can omit or greatly simplify the validation logic. Second, Task2 in A3 clearly requires multiple views, and multiple views can be shown at the same time, making it easy for users to use. Finally, Refactoring tech mentioned that we should have concentrated the output printing, rather than scattering it in various functions, which may cause some problems. MVC can help us to solve this problem by layering the code again, making the code more robust and extensible. The goal of MVC is to separate the user interface from the business logic to make the code extensible, reusable, maintainable and flexible. The View layer is the interface, the Model layer is the business logic, the Controller layer is used to schedule the View layer and the Model layer and organize the user interface and business logic.

MVC also consists of the strategic pattern and observer pattern. In MVC, the Model is the object being observed, and the View is the observer. Once the Model layer changes, the View layer is notified of the update. It's an embodiment of the Observer Pattern.

The Strategy Pattern is the relationship between View and Controller. Controller is a strategy of View, and Controller is replaceable for View, which is also an embodiment of the Strategy Pattern.

And about the design patterns and principles:

The Memento Pattern

Since the system wants to capture the timestamp of the booking changes and store the past 3 changes. So, I use the **Memento Pattern** here because memento is a behavioural design pattern that lets us save and restore the previous state of an object without revealing the details of its implementation. We can use the **Memento pattern** when we want to produce snapshots of the object's state to be able to restore a previous state of the object. And The **Memento pattern** lets you make full copies of an object's state, including private fields, and store them separately from the object.
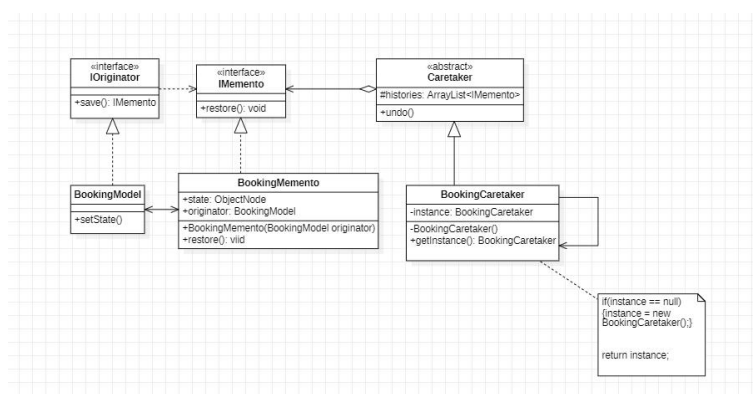
This implementation of the **Memento Pattern** allows having multiple types of originators and mementos. Each originator works with a corresponding memento class. Here, BookingMemento will link with BookingModel. What's more, neither originators nor mementos expose their state to anyone.

Caretakers are now explicitly restricted from changing the state stored in mementos. Moreover, the caretaker class becomes independent from the originator because the restoration method is now defined in the memento class.

Each memento becomes linked to the originator that produced it. The originator passes itself to the memento's constructor, along with the values of its state. Thanks to the close relationship between these classes, a memento can restore the state of its originator, given that the latter has defined the appropriate setters.

Use the **Memento Pattern**, the pros are:

1. We can produce snapshots of the object's state without violating its encapsulation.

2. We can simplify the originator's code by letting the caretaker maintain the history of the originator's state.

The cons are:

1.The app might consume lots of RAM if clients create mementos too often.

2.Caretakers should track the originator's lifecycle to be able to destroy obsolete mementos.

Considering that our programming language is Java, and we won't overuse mementos too often since we only need to store the past three changes of booking. So, we decide to use the **Memento pattern** here.

The Observer Pattern

In our code, the ConcreteSubscribers classes (UserModel and CurrentOperator) want to track changes to the Publisher Class. And Publisher are going to notify the ConcreteSubscribers classes (UserModel and CurrentOperator) about changes to bookings. So the ConcreteSubscribers classes (UserModel and CurrentOperator) are like subscribers. And the Publisher Class are like publishers. So we need to implement the **Observer Pattern** to define a subscription mechanism to notify multiple subscriber objects about any events that happen to the publisher object they're observing.

In our code, we use the **Observer Pattern** in two places, as shown below,
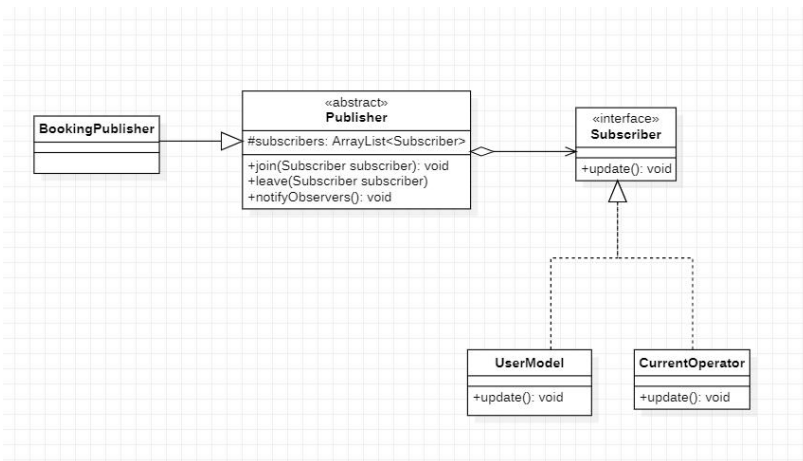
The Publisher is the publisher to notify its concrete subscribers (UserModel, CurrentOperator) everytime the state changes or behavior is executed. The Publisher class also contains a subscription infrastructure that notifies the subscribers(notifyObservers()) and lets new subscribers join(join(Subscriber subscriber)) or current subscribers leave the list(leave(Subscriber subscriber)).

The pros about using the **Observer Pattern**:

1.**Open Closed Principle (OCP)**. We can introduce new subscriber classes without having to change the code of the publisher (Publisher)

2. We can reuse these subscriber(UserModel and CurrentOperator) classes independently of each other.

2. We can establish relations between these subscriber objects (UserModel and CurrentOperator) at runtime.

3. Changes to subscriber classes (UserModel and CurrentOperator) will not affect the other.

But the con about this pattern is that the action classes are notified in random order.

It doesn't matter for our code about the notification order. So, it will be great to use the **Observer Pattern** here.

The Singleton Pattern

We use the **Singleton Pattern** for BookingPublisher class and BookingCaretaker class. In terms of the reasons for using this pattern:

The life cycle of the function served by the BookingPublisher class and BookingCaretaker class is for the whole system and will not end when a function ends as long as the whole application is running. We need them to be the same object all the time instead of creating a new object every time. That's the main reason we decided to use this pattern.

Normally, we use the global variables to store some essential objects. While they're handy, they are also very unsafe since any code can potentially overwrite the contents of those variables and crash the app. The **Singleton pattern** not only lets us access some objects from anywhere in the program, but also protects that instance from being overwritten by other code.

Here, BookingPublisher class and BookingCaretaker classes only have a single instance available. We can't create objects of the S BookingPublisher class and BookingCaretaker classes except for the creation method(getInstance()). This method either creates a new object or returns an existing one if it has already been created.
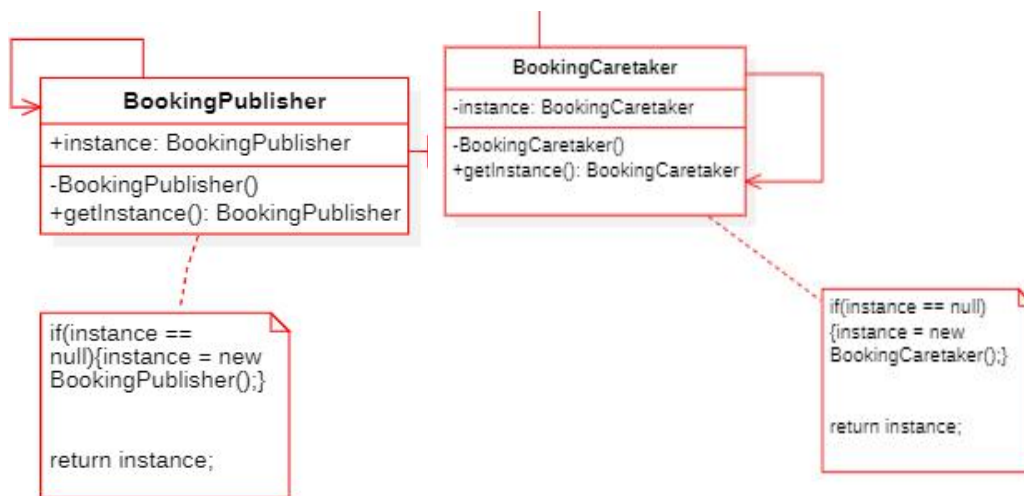
Using this pattern, the pros are:

1. We can be sure that BookingPublisher class and BookingCaretaker classes have only a single instance, which means, in our running application, there is only one memento and publisher object(so user can access histroy data at any time and do notification automatically at any time as long as the system is running)

2. We can gain a global access point to the instance of BookingPublisher class and BookingCaretaker classes. Make us easier to call relavant functions.

But the cons about this pattern are:

1. The **Singleton pattern** can mask bad design, for instance, when the components of the program know too much about each other.

2. It may be difficult to unit test the client code of the singleton because many test frameworks rely on inheritance when producing mock objects. Since the constructor of the singleton class is private and overriding static methods is impossible in most languages, we have to think of a creative way to mock the singleton.

To sum up, although the **Singleton Pattern** has certain limitations. However, the BookingPublisher and BookingCaretaker data are shared across the system and must be consistent, so we must create only one instance for them, which only the **Singleton Pattern** can achieve.

About the refactoring techniques:
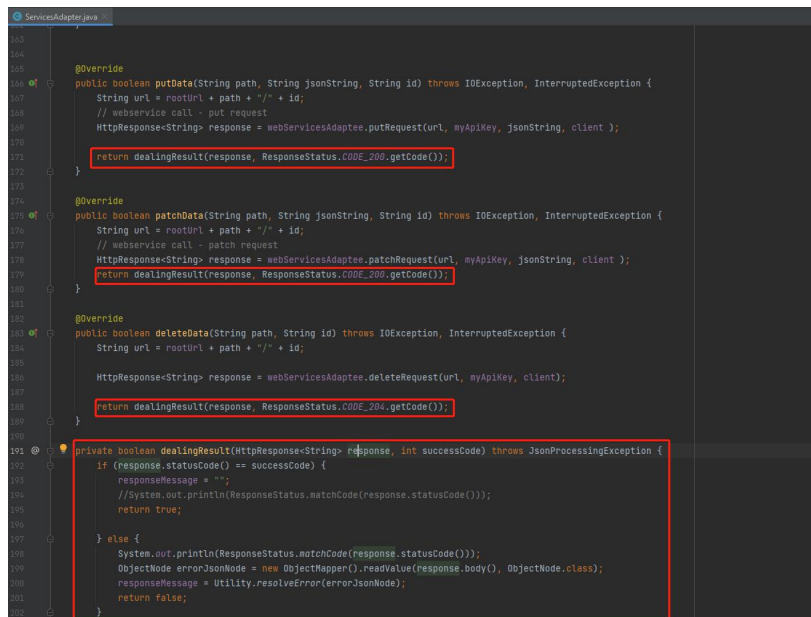
(we use many of refactoring, just list three examples in the rationale)

1. **Replace Value with Object:**
   The UserModel class contains the userType data field. The field has its own behaviour and associated data. So, we create a new class (UserType class), place the old field and its behaviour in

the class, and store the object of the class in the original class. And the benefit is to improve relatedness inside classes.

2. **Extract Function/Method:**

The code fragment about dealing with result can be grouped together since they have the same purpose in every function, which smells code duplication, mutable data and divergent change.
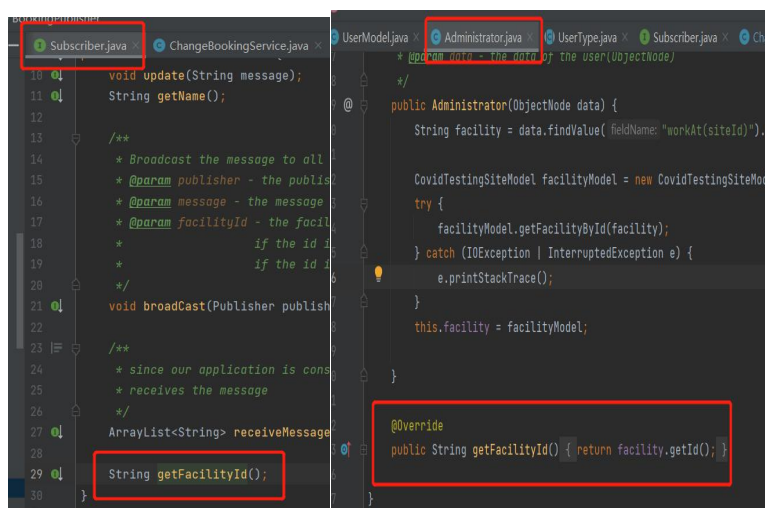
So we move this code to a separate new method(dealingResult()) and replace the old code with a call to the method to reduce the duplication of code and improve the readability and isolation.

3. **Hide Delegate**

   Problem: The client gets object B from a field or method of object A. Then the client calls a method of object B. • Solution: Create a new method in class A that delegates the call to object B. Now the client doesn't know about, or depend on, class B

The client class wants to get facilityId from BookingPublisher/Subscriber/Administrator…. Classes. Then the client calls a method of Facility class, which causes data clumps and large classes.

So, we create a new method in BookingPublisher/Subscriber/Administrator… classes that delegated the call to Facility class. Now the client doesn't know bout, or depend on, Facility class. So, we can hide delegation from the client.