FIT3077 Assignment 3 Design Report

Link for demonstration video. Please access with Monash account:
https://drive.google.com/drive/folders/1GLlJldlt9wIVEgM_77UTZ6l0oRPuvVph?usp=sharing

Model-View-Controller (MVC) Architecture

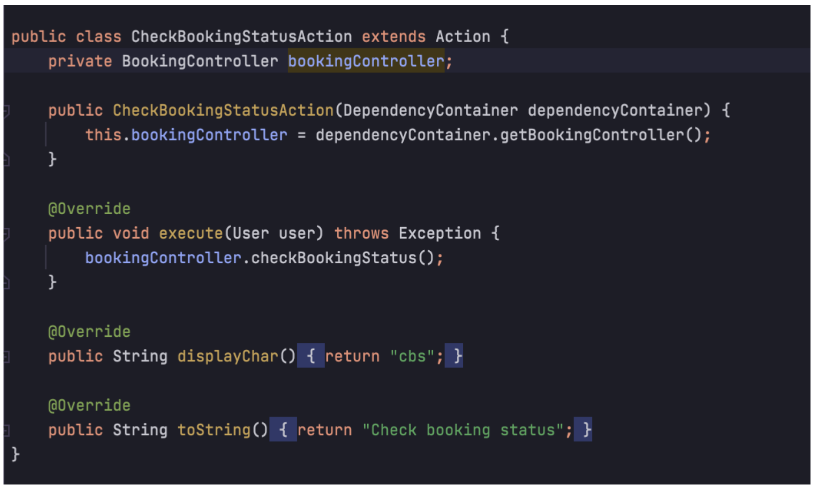
In this assignment, we are employing the MVC architecture. With MVC, our code is separated into 3 components, namely the Model, Controller, and View. With this separation and hence the independence of each component, we experienced that modifications and the development for new features become easier compared to our old approach, where we have not defined the boundaries for responsibilities in terms of software layers (i.e. presentation layer, business logic layer, data access layer, etc.).

In MVC, a Model is the bottom-most layer which is responsible for management of data. Basic CRUD functionalities are implemented here for that purpose. In our previous assignment, we have separated this layer away into one type of class called `subsystem` (i.e. we have TestSubSystem, BookingSubSystem, etc. which reads and writes data to the FIT3077 API. Now we are renaming them to TestModel, BookingModel, and so on.).

View is the layer which is closely related to UI as it is responsible for determining what the user should see at certain touch points in the application. The logic enclosed within View classes, in our case, deals with what gets printed on the CLI and takes in user input to be processed by the Controller. View is also aware of Model, as it would somewhat deal with data as well (user input).

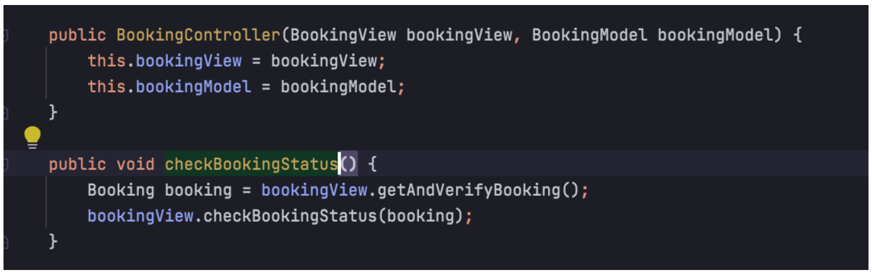
Controller holds the most amount of business logic among other components. It makes use of and combines the return values from View and Model (including user inputs) and proceeds with the logic. It is the brain of the application.

In order to convert our previous code to follow the MVC concept, it mostly involves separating the logic in our facade classes that extend from the abstract class Action. These Action child classes used to contain both the logic for View and Controller. Now that we have separated them, we feel that our code is more reusable as we continue to develop the application.

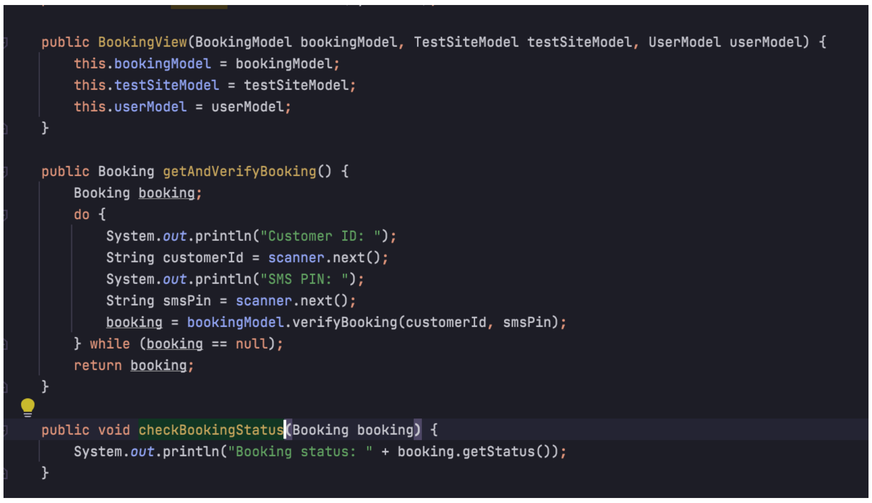


```java
public class CheckBookingStatusAction extends Action {
    private BookingController bookingController;

    public CheckBookingStatusAction(DependencyContainer dependencyContainer) {
        this.bookingController = dependencyContainer.getBookingController();
    }

    @Override
    public void execute(User user) throws Exception {
        bookingController.checkBookingStatus();
    }

    @Override
    public String displayChar() { return "cbs"; }

    @Override
    public String toString() { return "Check booking status"; }
}
```

In our code, Action-type classes have a Controller class as an injected dependency. The logic originally implemented in Action-type classes are then partially delegated firstly to this Controller class, then View and Model, where appropriate. This decoupling of logic also promotes reusability and prevents several code smells like god classes and long methods.



```java
public BookingController(BookingView bookingView, BookingModel bookingModel) {
    this.bookingView = bookingView;
    this.bookingModel = bookingModel;
}

public void checkBookingStatus() {
    Booking booking = bookingView.getAndVerifyBooking();
    bookingView.checkBookingStatus(booking);
}
```



```java
public BookingView(BookingModel bookingModel, TestSiteModel testSiteModel, UserModel userModel) {
    this.bookingModel = bookingModel;
    this.testSiteModel = testSiteModel;
    this.userModel = userModel;
}

public Booking getAndVerifyBooking() {
    Booking booking;
    do {
        System.out.println("Customer ID: ");
        String customerId = scanner.next();
        System.out.println("SMS PIN: ");
        String smsPin = scanner.next();
        booking = bookingModel.verifyBooking(customerId, smsPin);
    } while (booking == null);
    return booking;
}

public void checkBookingStatus(Booking booking) {
    System.out.println("Booking status: " + booking.getStatus());
}
```

In our code, data access (i.e. involving Model classes) may occur in either View or Controller depending on the feasibility of implementing the logic. Either way, we still hold the concept that View should only be responsible for display logic, while Controller for business logic. As we attempt to follow this consistently, our code has become more coherent as related logics are grouped together.

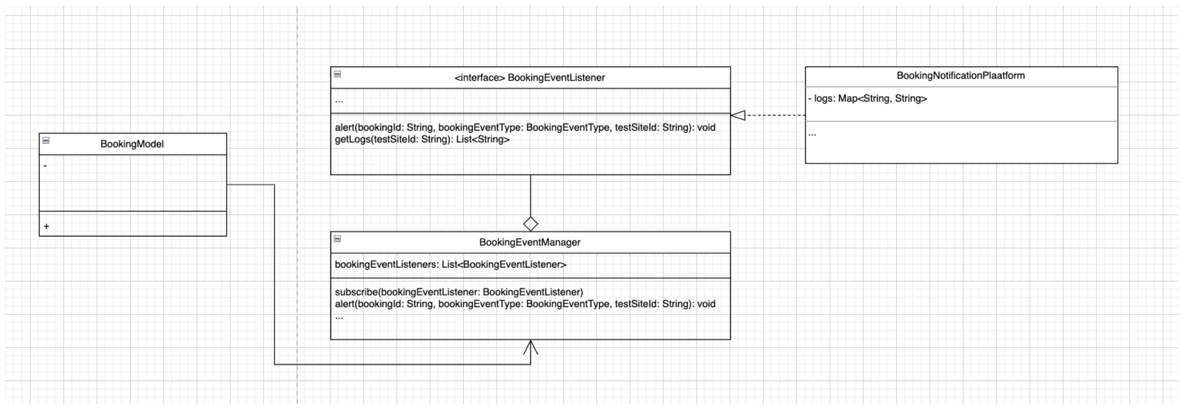
By using MVC, we have also implemented DRY Principle and Dependency Inversion Principle. For example, the Action Abstract Class depends on the DependencyContainer instead of the Actions class depending on it,this reduces the redundant code as well, making the extension of system easier in the future.

Design Patterns

Observer pattern for Admin Booking Interface

To implement the second requirement for Admin Booking Interface, we have employed the Observer pattern. As per the specification, we can say that the admin is essentially a subscriber to these booking changes events. Considering the extensibility of this feature (where there may be multiple kinds of booking events subscribers), we create an interface for the subscriber object so that they follow a contract that allows them to be able to have a method invoked or in other words receive a notification.

Without this interface (and hence without dependency inversion), changes in the BookingModel class (the ‘editor’ in our case, please refer to the diagram below) are always required as it needs to be aware of the identity of its subscriber object. It would be a violation of both the Single Responsibility Principle (as this is a rather minor part of the BookingModel class yet it requires much changes, when according to the principle, there should only be “one reason to change" for every class), and the Open Closed Principle (the BookingModel class is already well tested, it is not supposed to be open to changes anymore).

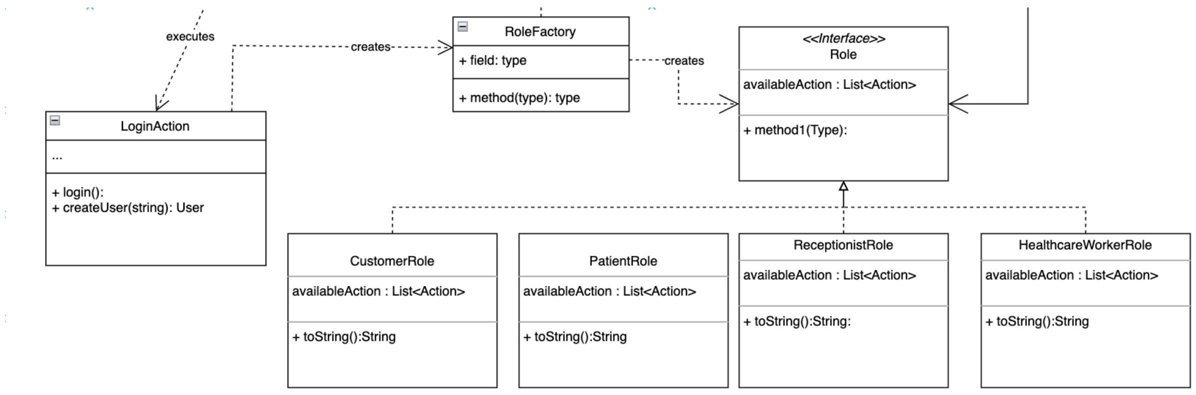

BookingModel
-
+
<interface> BookingEventListener
...
alert(bookingId: String, bookingEventType: BookingEventType, testSiteId: String): void
getLogs(testSiteId: String): List<String>
BookingNotificationPlaatform
- logs: Map<String, String>
...
BookingEventManager
bookingEventListeners: List<BookingEventListener>
subscribe(bookingEventListener: BookingEventListener)
alert(bookingId: String, bookingEventType: BookingEventType, testSiteId: String): void
...

Factory method pattern for Roles and access controls

To implement the first requirement, we are building on top of the design pattern we already incorporated in our previous assignment, regarding roles.

We are still keeping the factory method pattern for management of user roles and access controls. Using the factory method separates the instantiation logic of the Role-typed classes from the User object, which involves the designation of access controls – meaning setting which roles are permitted to do certain `actions` (stored in the field `availableActions`).

This requirement is a feature request involving access controls. Using the factory method pattern, it has become easier for us to delegate the responsibilities to certain roles and be done with it with a small amount of components needing to be aware of the roles' allowed actions. As opposed to our alternative approach which we did not end up taking (to implement some validation layer before accessing a method to determine whether the accessor is eligible or not), where many components need to be aware of that information (each method or functionality need to be aware of which roles are allowed to access them).

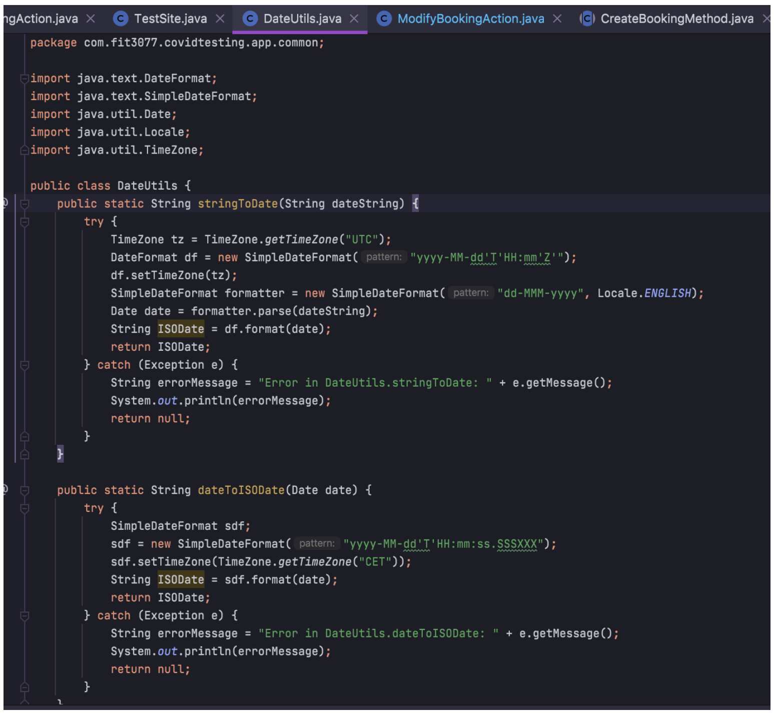


Facade design pattern for Actions

Action-type classes in our code can be treated as UI (as we use CLI as frontend). They are the entry point to our application and encapsulates the actions (features) we can do inside the app. Action-type classes are injected with one dependency, the `dependencyContainer` object which we made to mimic the Spring IoC container, to better manage and use dependencies with dependency injections. With this, the caller to Action-type classes need not be aware of the logic and dependencies of each action object, as we abstract it away inside the class itself.

Refactoring and Spotting Code Smells

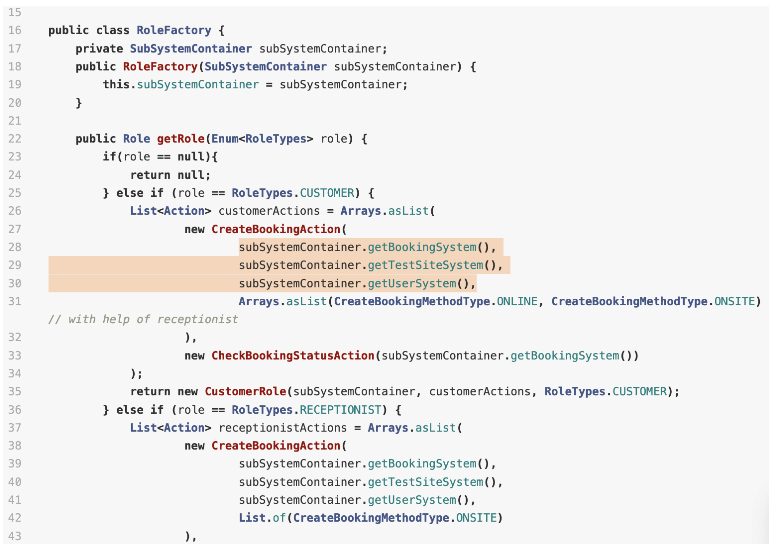
DRY principle



```java
package com.fit3077.covidtesting.app.common;

import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Locale;
import java.util.TimeZone;

public class DateUtils {
    public static String stringToDate(String dateString) {
        try {
            TimeZone tz = TimeZone.getTimeZone("UTC");
            DateFormat df = new SimpleDateFormat( pattern: "yyyy-MM-dd'T'HH:mm'Z'");
            df.setTimeZone(tz);
            SimpleDateFormat formatter = new SimpleDateFormat( pattern: "dd-MMM-yyyy", Locale.ENGLISH);
            Date date = formatter.parse(dateString);
            String ISODate = df.format(date);
            return ISODate;
        } catch (Exception e) {
            String errorMessage = "Error in DateUtils.stringToDate: " + e.getMessage();
            System.out.println(errorMessage);
            return null;
        }
    }

    public static String dateToISODate(Date date) {
        try {
            SimpleDateFormat sdf;
            sdf = new SimpleDateFormat( pattern: "yyyy-MM-dd'T'HH:mm:ss.SSSXXX");
            sdf.setTimeZone(TimeZone.getTimeZone("CET"));
            String ISODate = sdf.format(date);
            return ISODate;
        } catch (Exception e) {
            String errorMessage = "Error in DateUtils.dateToISODate: " + e.getMessage();
            System.out.println(errorMessage);
            return null;
        }
```

We noticed that we have one of the simplest code smells: code repetition. We have the following method as the FIT3077 API endpoint seems to only accept ISO dates. When doing this assignment, we realised that this method has been copied frequently throughout the application. Hence, we decided to encapsulate it into one method, inside one class DateUtils, while also making this method static so it is accessible for any object to use.

Long parameter list



```java

public class RoleFactory {
    private SubSystemContainer subSystemContainer;
    public RoleFactory(SubSystemContainer subSystemContainer) {
        this.subSystemContainer = subSystemContainer;
    }

    public Role getRole(Enum<RoleTypes> role) {
        if(role == null){
            return null;
        } else if (role == RoleTypes.CUSTOMER) {
            List<Action> customerActions = Arrays.asList(
                    new CreateBookingAction(
                            subSystemContainer.getBookingSystem(),
                            subSystemContainer.getTestSiteSystem(),
                            subSystemContainer.getUserSystem(),
                            Arrays.asList(CreateBookingMethodType.ONLINE, CreateBookingMethodType.ONSITE)
// with help of receptionist
                    ),
                    new CheckBookingStatusAction(subSystemContainer.getBookingSystem())
            );
            return new CustomerRole(subSystemContainer, customerActions, RoleTypes.CUSTOMER);
        } else if (role == RoleTypes.RECEPTIONIST) {
            List<Action> receptionistActions = Arrays.asList(
                    new CreateBookingAction(
                            subSystemContainer.getBookingSystem(),
                            subSystemContainer.getTestSiteSystem(),
                            subSystemContainer.getUserSystem(),
                            List.of(CreateBookingMethodType.ONSITE)
                    ),
```

There used to be the 'long parameter list' code smell in our RoleFactory, which makes it hard to modify and extend as this code smell also introduces the connascence of position (i.e. order matters even with minimal indication of which order is correct, function parameters need to be passed in in a certain order).

We address this along with the aim of wanting to have better management of dependencies. So instead of injecting the dependencies of each Action-type class one by one, we inject the `dependencyContainer` at once. The logic inside the Action class will determine which dependencies it needs. This way, we also eliminate the long parameter list code smell.

Package Organisation

Common Reuse Principle

Classes that are commonly used together or changed together should be grouped in the same package since the classes are tightly coupled and heavily dependent on each other. For example, the classes related to the Booking functionality, i.e. dependent on the subsystem BookingSystem, like CreateBookingAction and CheckBookingStatusAction, are grouped in the same package as they are heavily dependent on each other and have to do with the domain Booking class. Grouping related classes together increases locality, hence if some classes are actually found to be connascent, the impact of a future refactor will not be as large when they are further apart in different packages. Since the affected entities would still be in the same package, it will ease the test and debugging process when required.

Acyclic Dependency Principle

We have refactored our code to eliminate any cyclic dependencies between classes to avoid the Morning After Syndrome .This ease our job when we try to debug and extend our system. But at the same time,we also need to create some new classes to achieve this,which makes our system slightly more complicated.