

# Assignment 3: Extending and Refactoring the Design

Group - MA\_Lab04Team1

Name: Chok Ming Jie, Suchit Sudhir Krishna

## 1. Introduction

As a result of previous design choices, implementation of Assignment 3 was significantly boosted due to the existing principles and patterns. However to further accommodate the new requirements, new principles, patterns and architectural changes were made. The document outlines these changes and discusses the benefits and drawbacks of the implementation

## 2. Extensibility

As a result of the previous design we are able to reuse and extend the existing system with ease. Things such as the TestingSite singleton were used as a means of displaying the testing sites from the booking memento applied. In addition, since all classes had single responsibility principle in mind it was easier to find and connect different parts of the encapsulated objects. Using the response adapter it was also possible to easily convert data to its textual representation which was used to work with API and was made more efficient as a result. Lastly, using the frames as a means of holding all business logic made it easier to segregate its functionalities into its respective model and controller making it easier to reduce the dependencies and increase cohesion by simplify applying refactoring methods.

## 3. Design Principles

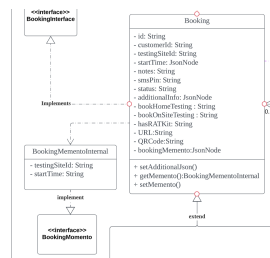
### 2.1 Single Responsibility Principle (SRP) [Reference: FIT 3077 Week 3 Lecture Slides]

SRP is applied to classes that should have responsibility over a single part of the software's functionality, and make that responsibility entirely encapsulated by the class. The class should have one and only one reason to change.



For example, the RunMessage class, it stores all the functionalities that are related to messages. So that if in the future, we get new features that are related to Message which need to be added or changed, we can easily make those changes to them in the RunMessage class as it stores all the information about the message functionality. This helps to make the system easier to maintain and reduces coupling between responsibilities.

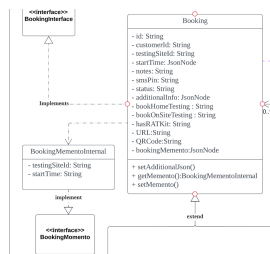
## 2.1 Open-Closed Principle (OCP) [Reference: FIT 3077 Week 3 Lecture Slides]



As a memento is applied within the booking package, open closed principle was kept in mind to offer a closed modification and open to extension property. As a BookingMementoInterface is applied we can simply implement a new memento in order to restore different aspects of the booking class as currently only testingSiteId and the timing are restored. This is beneficial as if we want other parts of the system to restore other properties it is possible to simply create a memento that will do so and pass it into the setMemento function. Finally there is a drawback as if multiple mementos are added it might be difficult to ensure the correctness of functionality and may require us to perform extensive testing thus it would be a good idea to simply leave the memento classes to a minimum.

## 4. Design Patterns

### 3.1 Memento [Reference: <https://github.com/aaleti/java-design-patterns/tree/master/memento>]



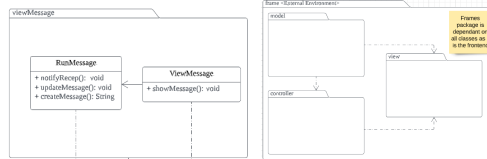
```
public BookingMemento getMemento() {
    BookingMementoInternal state = new BookingMementoInternal();
    state.setStartTime(startTime);
    state.setTestingSiteId(this.getTestingSiteId().getId().asText());
    return state;
}

public void setMemento(BookingMemento memento) {
    BookingMementoInternal state = (BookingMementoInternal) memento;
    this.startTime = state.getStartTime();
    this.testingSiteId = state.getTestingSiteId();
}
```

Mementos are a behavioural design pattern that lets you save and restore the previous state of an object without revealing the details of its implementation. In the case of the system a booking memento was implemented as a means of restoring the bookings when a customer or operator wishes to do so. A benefit of using this design pattern is that the user can easily restore booking data to previous states with ease and helps the administrator keep track of the lifecycle of the object all whilst maintaining encapsulation. A drawback of this is that if the originator object is too big then it may prove to be more expensive in terms of computational cost whilst restoring and saving data. This is however counteracted by providing the least amount of data to restore the date and testing site.

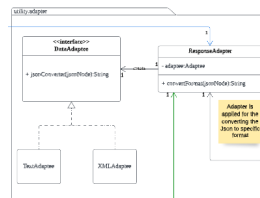
## 5. Package Principle

### 4.1 Common Closure Principle (CCP) [Reference: FIT 3077 Week 3 Lecture Slides]



In the purpose of increasing the cohesion, package viewMessage and frame is created, which these packages contain all the information about the Message and frame(MVC). This follows the Common Closure Principle. This approach is able to increase the maintainability as when there's a change to the requirement, we do not need to do a lot of re-validation and redeploying. All the classes in each package are closed together for the purpose of function. For example, if there is a function for delete messages in the future, all the classes in the viewMessages will only be affected. This is the same to the frame when a new model is added.

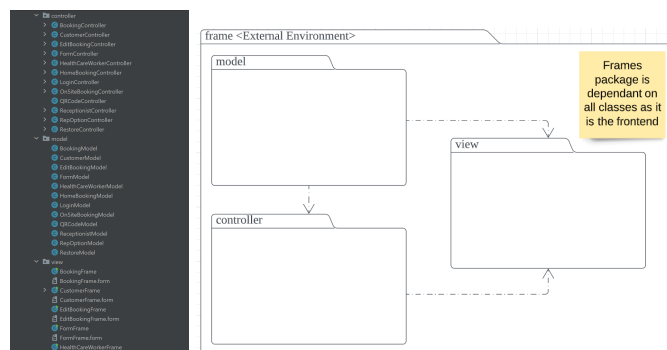
### 4.2 Common Reuse Principle (CRP) [Reference: FIT 3077 Week 3 Lecture Slides]



CRP has applied in utility.adapter. CRP states that classes that tend to be reused together belong in the same component. Classes are seldom reused in isolation. All the classes in the adapter are reused together. If we reuse one of the classes in the adapter, we reuse them all. All the classes that make up the adapter are tightly coupled and depend heavily on each other. CRP is applied as it is hard to separate them easily. Classes that are not tightly bound to each other are not in the same components.

## 6. Architecture Patterns

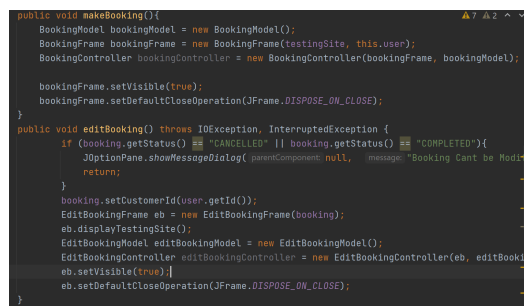
### 5.1 Model-view-controller [Reference: <https://github.com/aaleti/java-design-patterns/tree/master/model-view-controller>]



Model-View-Controller or MVC is an architectural pattern that separates an application into three logical components: the model, view and controller as depicted in the name. Our system includes this by segregating each view to have its own model and respective controller. This makes it easier to understand the difference between the business logic and the end visual representation of the system. An advantage of utilising this method is that it coherently organises the application as aforementioned through separation of the three components and also makes the system easily modifiable as a single section is already independent of the other ones. A disadvantage is that it is not quite suitable for small applications and might actually be harder for other programmers to understand the entire model. As a result there is a lot of emphasis placed on documentation and commenting to ensure the code inside the MVC is detailed and accounted for.

## 7. Refactoring

### 6.1 Move Methods [Reference: <https://refactoring.guru/move-method>]



```
public void makeBooking(){
    BookingModel bookingModel = new BookingModel();
    BookingFrame bookingFrame = new BookingFrame(testingSite, this.user);
    BookingController bookingController = new BookingController(bookingFrame, bookingModel);

    bookingFrame.setVisible(true);
    bookingFrame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
}

public void editBooking() throws IOException, InterruptedException {
    if (booking.getStatus() == "CANCELLED" || booking.getStatus() == "COMPLETED"){
        JOptionPane.showMessageDialog(parentComponent, null, "message: Booking Can't be Modified",
        JOptionPane.ERROR_MESSAGE);
        return;
    }
    booking.setCustomerId(user.getId());
    EditBookingFrame eb = new EditBookingFrame(booking);
    eb.displayTestingSite();
    EditBookingModel editBookingModel = new EditBookingModel();
    EditBookingController editBookingController = new EditBookingController(eb, editBookingModel);
    eb.setVisible(true);
    eb.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
}
```

Move methods is a refactoring technique in order to relocate methods into distinct classes to isolate the behaviour. This was especially useful in the implementation of the MVC as frame class methods could be used as a model to reduce dependencies between the model and the view. A prominent example of this is visible within the Customer where the majority of its business logic was extracted from the customer frame which dramatically reduced the dependency between the view and the model as a controller would manipulate them within itself. Isolating the behaviour might reduce the dependencies and increase cohesiveness, however it also drastically increases the number of classes as this has been applied to every frame thus it may become more tedious to locate connecting parts from subsystems.

### 6.2 Move Classes [Reference: <https://refactoring.guru/extract-class>]

As a means of ensuring encapsulation and making sure that classes belong in appropriate packages we can move classes to other packages where it is actually being used more often. In the case of our system the login class which belonged to utility has been refactored to be renamed as login model and moved to the model package for the MVC. This was done as it is more relevant under the model as it is being used by the LoginController solely. An advantage of this is that we can easily find the classes around the package that it is being used in. However the hierarchy level must be chosen appropriately or it may prove more difficult to locate it if chosen incorrectly.

[Reference: <https://refactoring.guru/extract-method>]

```

for (User user : userList) {
    if (user.getAccountId() != null) {
        if (user.getAccountId().equals(accountId)) {
            this.updateMessage(accountId, user, message);
        }
        if (user.getAccountId().equals(reviewId)) {
            this.updateMessage(reviewId, user, message);
        }
    }
}
}

//update a message
public void updateMessage(String testingId, User user, String message) throws Exception, InterruptedException {
    String message = user.getId() + " " + message;
    this.sendMessage(message);
    this.sendMessage(message);
    String msg = "OK" +
        "\n" + "testingId:" + testingId + " " + message +
        "\n" + "reviewId:" + this.getReviewId() + " " + "OK" +
        "\n";
    ObjectMapper objectMapper = new ObjectMapper();
    JsonNode jsonNode = objectMapper.readValue(msg, JsonNode.class);
    User updateUser = new User(user, (JsonNode)jsonNode);
    api.put("api/users/" + updateUser.getId(), updateUser);
    this.sendMessage(updateUser.getId() + " " + "OK");
}

```

Extract Methods is used as a means of extracting very long methods and separating them into a method of its own to isolate and model the behaviour more accurately. In the case of our system the most prominent example is in the `RunMessage` class whereby the method has been extracted into its own `updateMessage` method responsible for solely updating the messages. The benefit of this is that we are able to easily separate the code into smaller sections to reduce the amount of code smells. However it may introduce unwanted dependencies to the parameters when trying to extend the system and should be used minimally.

[Reference: <https://refactoring.guru/preserve-whole-object>]

```

795
    * Constructor of Booking which create for patching the data
    * @param booking Booking object
    * @param additionalInfo Booking additional Information
    *
    * @return Booking
    */
    public Booking(Booking booking, JsonNode additionalInfo) {
        this.id = booking.getId();
        this.customerId = booking.getCustomer().get("id").textValue();
        this.testingSiteId = booking.getTestingSite().get("id").textValue();
        this.startTime = booking.getStartTime();
        this.notes = booking.getNotes();
        this.status = booking.getStatus();
        this.additionalInfo = additionalInfo;
    }
}

```

Preserve Whole Object is a refactoring technique that helps to treat the code smell of Long Parameter List. Instead of passing a group of data received by another object as parameters, we pass the object itself to the method by using this refactoring technique. This is very helpful for a Booking Registration System which the Booking class is mainly used in the system and will have many improvements or add new attributes to it in the future. So in the future, if the method needs more data from the object, we don't need to rewrite all the places where the method is used - merely inside the method itself.