**City University of Hong Kong**

**CS3343 Software Engineering Practice**

**Analysis and Design Report**

PROJECT TITLE: CITYU STUDY ROOM SCHEDULER

**Group 10**

|  |  |
| --- | --- |
| **Student Name** | **SID** |
| Nurdaulet TAUMERGENOV | 56679550 |
| Aliya OSPANOVA | 56842241 |
| Alibi ZHENIS | 57065469 |
| Anton SHATOKHIN | 56865310 |
| Beket YERMEKOV | 56679630 |
| Nur ALDEKEN | 56679433 |
| Dilnaz AUSHAKHIMOVA | 57012060 |

## 

## **Table of Contents**

[**Table of Contents 2**](#_1gie1ppt53ne)

[**Introduction 3**](#_qtzazl2butk4)

[**Design Constraints 4**](#_yreovvkthomu)

[Use Case Diagram 5](#_seovmmebkkyb)

[Use Case Specifications 6](#_tbypdgjfsyj8)

[Main GUI 6](#_vos1acvoewqx)

[Delete GUI 8](#_ni2ipz133ylx)

[Change GUI 10](#_7n9xi3j6ridw)

[Book GUI 12](#_yi1n0sf4v4am)

[Schedule GUI 14](#_mvrrp5fbnlf8)

[**Class Diagram 15**](#_3907mjhcgj1d)

[**Design Principles and Patterns 16**](#_cxi83p7zavhs)

[Single Responsibility Principle (SRP) 16](#_77oncw9iyxrq)

[Law of Demeter (LoD) 16](#_9c649nszevs0)

[Model-View-Controller (MVC) pattern 17](#_2k7snvw1ystg)

[Separation of Concerns principle (SoC) 18](#_jff56l8r5g9j)

[Singleton Pattern 19](#_h4eq1ehpnfqw)

[**Program Flow 20**](#_gjdgxs)

[ShowSchedule flow 20](#_j6e6vw11h82r)

[BookStudyRoom flow 22](#_hgg01uyg354h)

[DeleteBooking flow 24](#_9zu2xaml6n02)

[ChangeBooking flow 26](#_te0b9ugvymlq)

[Persistence flow 28](#_7tr11zbgbxp)

[**Testing Report 34**](#_8g9cznjdk19p)

[Module Organization 34](#_8vx2itnlalx)

[Testing Strategy 36](#_30j0zll)

[Branch Coverage 37](#_1fob9te)

[Test Cases 38](#_3znysh7)

[**Code refactoring 42**](#_6iqlnjdpk4l)

## **Introduction**

The goal of the Study Room Scheduler App is to provide City University students with a practical and efficient way to reserve study spaces. Students may maximize their study time by using the application's user-friendly design, making booking study rooms easier. The app solves the typical problems that students run into while trying to reserve study spaces. It's simple for users to browse and reserve available study spaces, make changes to their reservations as needed, and even postpone or cancel their study sessions. With the Study Room Scheduler App, users can locate and book the best study room available, whether they need a peaceful place for self-study or a cooperative area for group work.

Furthermore, the Study Room Scheduler App prioritizes user convenience and reliability. It undergoes rigorous compatibility testing to ensure seamless operation across different operating systems and web browsers.

With its user-centered design, robust functionality, and seamless integration with existing university systems, the Study Room Scheduler App is poised to become an indispensable tool for students of the college community.

## 

## 

## **Design Constraints**

During the design phase of the Study Room Scheduling App, several limitations and considerations were identified. These constraints include:

1. Persistence of Schedule:

* Constraint: The current implementation of the application lacks a centralized database, which limits the ability to handle multiple user accesses from different machines.
* Impact: Without a database, the application cannot support simultaneous data entry and scheduling from multiple users using different machines.
* Considerations: Enhancements need to be made to the application to incorporate a centralized database that can handle concurrent data entry and scheduling from multiple users. This would enable users to access and modify the schedule from different machines, ensuring data consistency and avoiding conflicts. Implementing database technologies suitable for multi-user environments and considering efficient data synchronization and conflict resolution mechanisms will be crucial in achieving this functionality.

2. Limited Scalability:

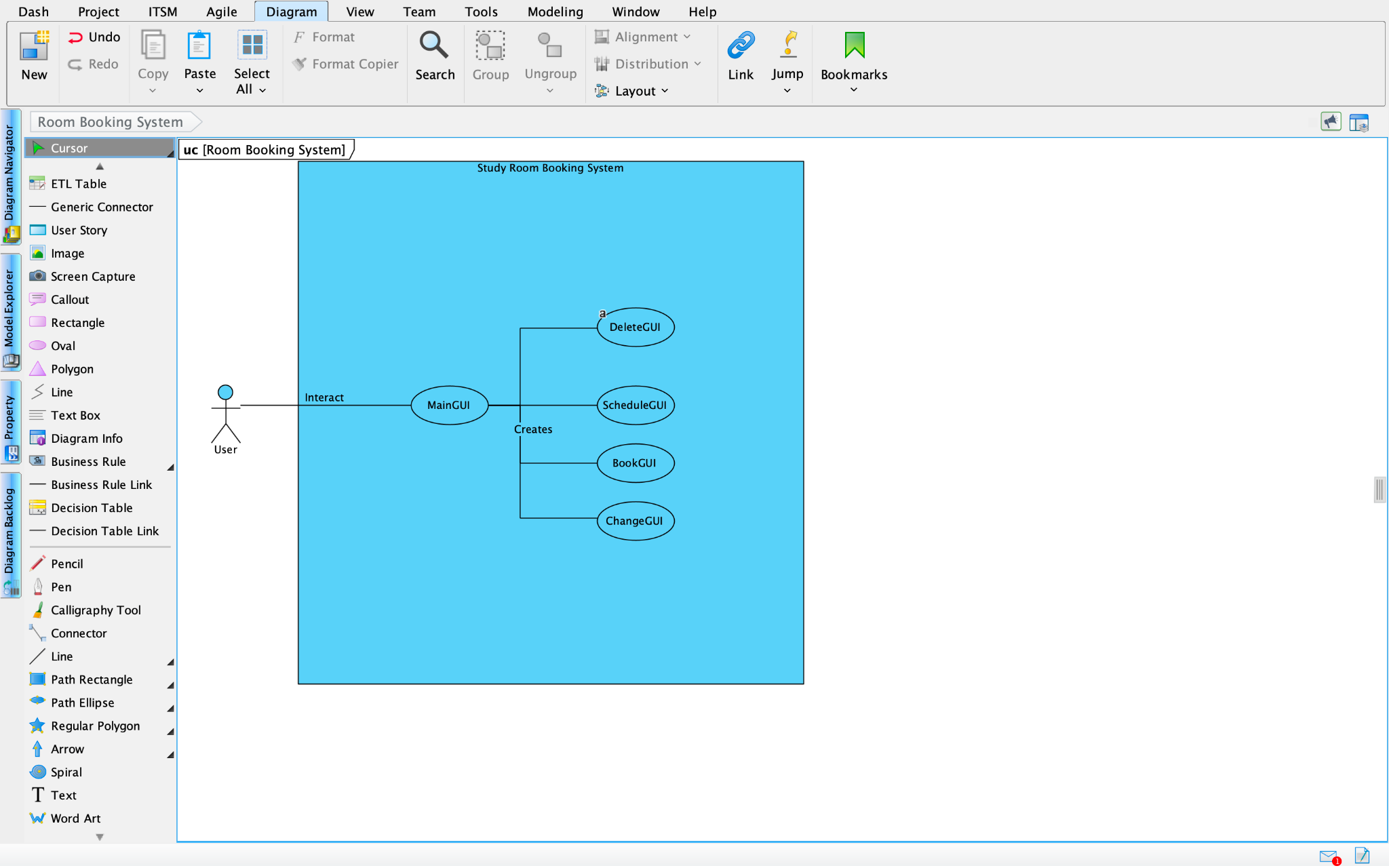
* Constraint: The system's capacity to handle a large number of concurrent users is limited.
* Impact: When a substantial volume of users access the system simultaneously, it may lead to performance degradation or system instability.
* Considerations: Load testing and performance optimization techniques should be implemented to ensure the system operates smoothly under reasonable user loads.

3. Unlimited Bookings per Individual:

* Constraint: The system does not impose any restrictions on the number of bookings an individual can make.
* Impact: This lack of limitations may result in some users monopolizing study rooms, potentially disadvantaging others.
* Considerations: Implementing a fair booking policy, such as capping the number of concurrent bookings or setting limits on the frequency of bookings, can promote equitable access to study rooms.

By recognizing and addressing these design constraints, the Study Room Booking System can be developed with a comprehensive understanding of the limitations and considerations involved. These constraints serve as important guidelines for designing and implementing the system, accounting for the absence of a dedicated database, scalability limitations, unrestricted bookings, and the need for schedule persistence.

## **Use Case Diagram**



## **Use Case Specifications**

### *Main GUI*

|  |  |  |
| --- | --- | --- |
| **Use Case Name:** | **MainGUI** | |
| **Actor(s):** | **User** | |
| **Description:** | This use case describes the interaction of the user with the Main GUI, which serves as the main entry point to the system. The Main GUI facilitates the access to different functionalities by creating and managing various GUI classes. | |
| **Typical course of events:** | **Actor Action** | **System Response** |
|  | **Step 1:** The user launches the application and the Main GUI is displayed.  **Step 2:** The user selects a specific functionality from the available options, such as:  - Deletion: The user selects the option to access the Delete GUI for deleting a room.  - Scheduling: The user selects the option to access the Schedule GUI for scheduling a room.  - Booking: The user selects the option to access the Book GUI for booking a room.  - Changing: The user selects the option to access the Change GUI for changing the details of a room. | **Step 3:** The System creates the corresponding GUI class based on the selected functionality.  **Step 4:** The created GUI class is displayed to the user, providing access to the specific functionality. |
| **Alternative course of events:** | ***N/A*** | |
| **Precondition:** | The system is running and the user has launched the application. | |
| **Postcondition:** | The user has accessed the desired functionality through the Main GUI. | |

### 

### *Delete GUI*

|  |  |  |
| --- | --- | --- |
| **Use Case Name:** | **DeleteGUI** | |
| **Actor(s):** | **User** | |
| **Description:** | This use case describes the interaction of the user with the Delete GUI, which enables the deletion of a room from the system. | |
| **Typical course of events:** | **Actor Action** | **System Response** |
|  | **Step 1:** The Main GUI creates the Delete GUI class.  **Step 2:** The user selects a specific room ("X100," "X200," "X300," or "X400") from the list of available rooms displayed in the Main GUI.  **Step 4:** The user selects the booking time slot to be deleted from the list.  **Step 6:** The user confirms the deletion. | **Step 3:** The Delete GUI displays the selected room number and a list of available booking time slots for deletion.  **Step 5:** The Delete GUI prompts the user to confirm the deletion of the selected booking.  **Step 7:** The Delete GUI sends a request to the system to delete the selected booking for the specific room.  **Step 8:** The system verifies the deletion request and removes the selected booking. |
| **Alternative course of events:** | **Step 3a:** If there are no available booking time slots for deletion, the Delete GUI won’t be interactable as there are no bookings to delete for the selected room. The use case terminates.  **Step 5a**: If the user decides not to proceed with the deletion, they can cancel the operation. In this case, the Delete GUI returns to the list of available booking time slots without deleting any booking. | |
| **Precondition:** | The user has accessed the Main GUI and selected the option to delete a room. | |
| **Postcondition:** | The selected room is deleted from the system. | |
| **Notes and Issues:** | - The Delete GUI is created and managed by the Main GUI.  - The system performs the actual deletion of the booking based on the request from the Delete GUI. | |

### *Change GUI*

### 

|  |  |  |
| --- | --- | --- |
| **Use Case Name:** | **Change GUI** | |
| **Actor(s):** | **User** | |
| **Description:** | This use case describes the interaction of the user with the Change GUI, which enables the modification of an existing booking in the system. | |
| **Typical course of events:** | **Actor Action** | **System Response** |
|  | **Step 1:** The Main GUI creates the Change GUI class.  **Step 2:** The user selects a specific room ("X100," "X200," "X300," or "X400") from the list of available rooms displayed in the Main GUI.  **Step 4:** The user selects the booking time slot to be changed from the list.  **Step 6:** The user confirms the modification. | **Step 3:** The Change GUI displays the selected room number and a list of available booking time slots for change.  **Step 5:** The Change GUI prompts the user to confirm the modification of the selected booking.  **Step 7:** The Change GUI sends a request to the system to change the selected booking for the specific room.  **Step 8:** The system verifies the request and changes the selected time slot. |
| **Alternative course of events:** | **Step 5a:** If the user decides not to proceed with the modification, they can cancel the operation. In this case, the Change GUI returns to the list of available booking time slots without making any modifications.  **Step 3a:** If there are no available booking time slots for modification, the Change GUI displays a message indicating that there are no bookings to modify for the selected room. The use case terminates. | |
| **Precondition:** | The user has accessed the Main GUI and selected the option to modify a booking. | |
| **Postcondition:** | The selected booking is modified in the system. | |

### 

### *Book GUI*

|  |  |  |
| --- | --- | --- |
| **Use Case Name:** | **Book GUI** | |
| **Actor(s):** | **User** | |
| **Description:** | This use case describes the interaction of the user with the Book GUI, enabling them to make a time slot booking for a study room in the Study Room Booking System. | |
| **Typical course of events:** | **Actor Action** | **System Response** |
|  | **Step 1:** The Main GUI creates the Book GUI class.  **Step 2:** The user selects a specific room ("X100," "X200," "X300," or "X400") from the list of available rooms displayed in the Main GUI.  **Step 4:** Selects a desired time slot from the available options. | **Step 3:** The Book GUI displays the selected room number and a list of available booking time slots.  **Step 5:** Checks the availability of the selected time slot for the chosen study room.  **Step 6:** The Book GUI sends a request to the system to reserve the selected booking for the specific room.  **Step 7:** Records the booking information, updates the study room availability, and notifies the user of the successful booking. |
| **Alternative course of events:** | **Step 5a:** If the selected time slot is not available, the Book GUI notifies the user and prompts them to choose another time slot or select a different study room. | |
| **Precondition:** | The user has accessed the Book GUI interface. | |
| **Postcondition:** | The user successfully books a time slot for the selected study room. The system updates the study room availability and records the booking information. | |

### 

### *Schedule GUI*

|  |  |  |
| --- | --- | --- |
| **Use Case Name:** | **Schedule GUI** | |
| **Actor(s):** | **User** | |
| **Description:** | This use case describes the interaction of the user with the Schedule GUI, allowing them to view their schedule of booked time slots in the Study Room Booking System. | |
| **Typical course of events:** | **Actor Action** | **System Response** |
|  | **Step 1:** The Main GUI creates the Schedule GUI class.  **Step 2:** The user selects a specific room ("X100," "X200," "X300," or "X400") from the list of available rooms displayed in the Main GUI. | **Step 3:** Schedule GUI retrieves and displays the booked time and available time slots for the selected room number. |
| **Alternative course of events:** | N/A | |
| **Precondition:** | The user has accessed the Schedule GUI interface. | |
| **Postcondition:** | The user can view the schedule of booked time slots for the selected room number. | |

### 

# **Class Diagram**

# 

# 

# 

# 

# 

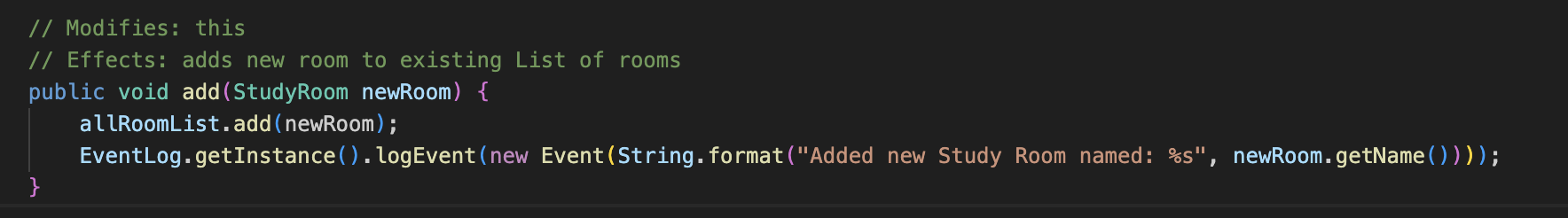
# **Design Principles and Patterns**

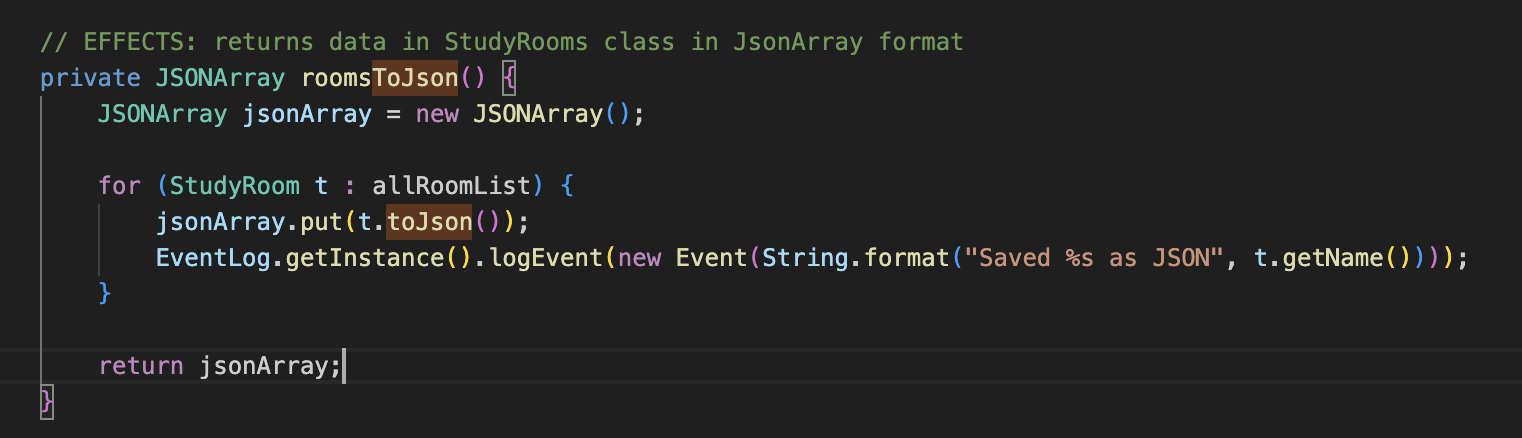
### **Single Responsibility Principle (SRP)**

All classes in our project including UI related and functionality related are following this principle, as long as they perform exactly one functionality and for each of them there is only one reason they can be changed. For example, ListRooms class has many getters, addition of the new study room and returning a JSON version of data asked, therefore, all functionality is related only to room listing and it’s methods are not performing any other side tasks except one explained and this principle is the same for all other classes in project. It also has huge advantages related to bugs localization and testing.

### **Law of Demeter (LoD)**

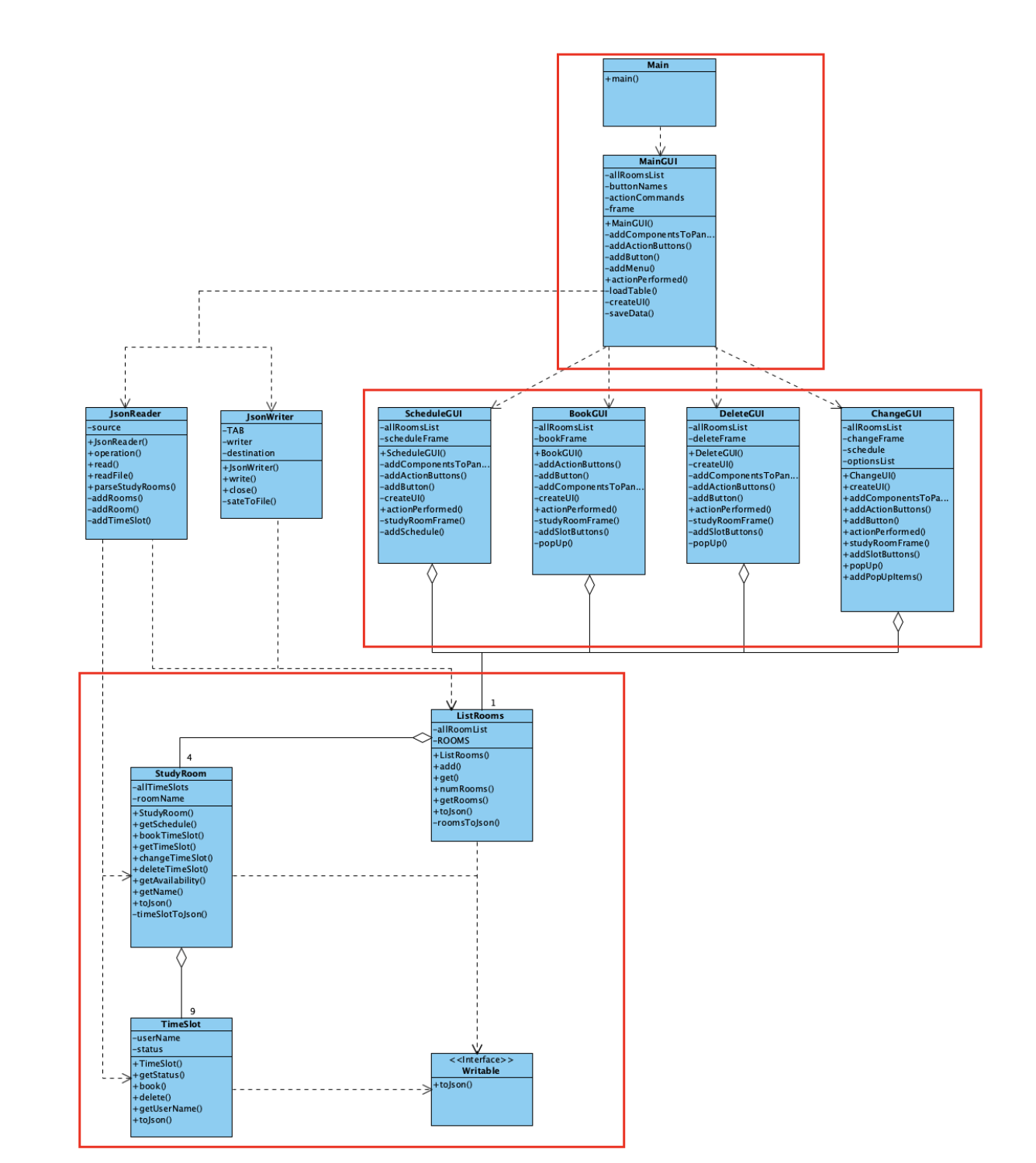
ListRooms class maintains StudyRoom objects but doesn’t expose internal state of StudyRoom object to external classes as long as it uses StudyRoom’s public methods to interact with it like getName() and toJson(). ListRooms class doesn’t have a lot of knowledge about how the methods and internal operations of StudyRoom are done therefore StudyRoom class internal changes will not significantly affect ListRooms class.





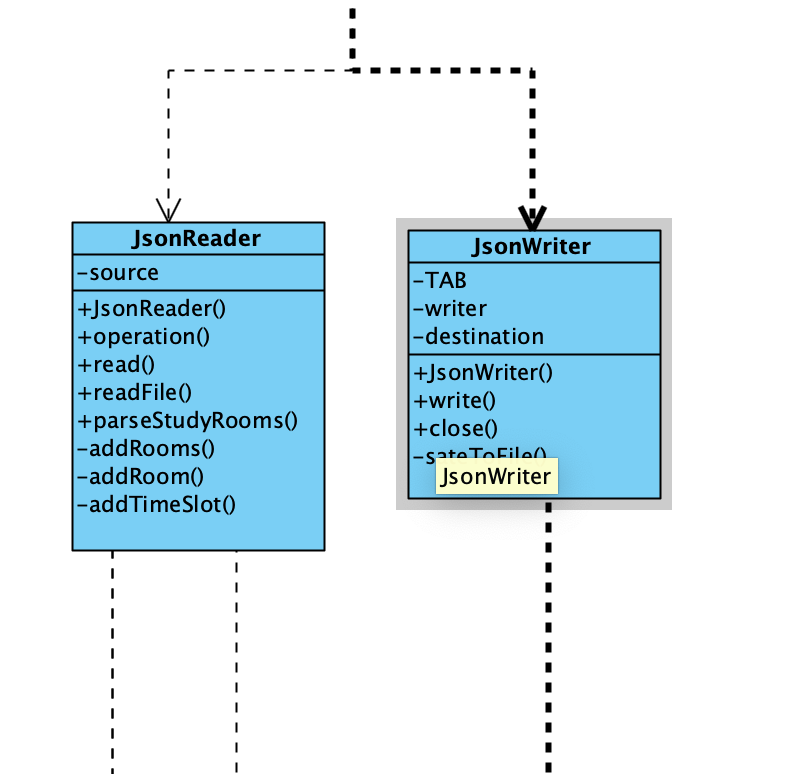
### **Model-View-Controller (MVC) pattern**

Our project utilizes MVC as long as it includes UI and user’s interaction with it. In particular all GUI related components (MainGUI, BookGUI, ChangeGUI, DeleteGUI, ScheduleGUI) apply this pattern. For examples, MainGUI. It works as a Controller, it handles all user inputs, updates Model component (ListRooms) and manipulates View components calling other GUI classes like BookGUI. That helps to separate modules responsibilities and perform easier problem localisation.



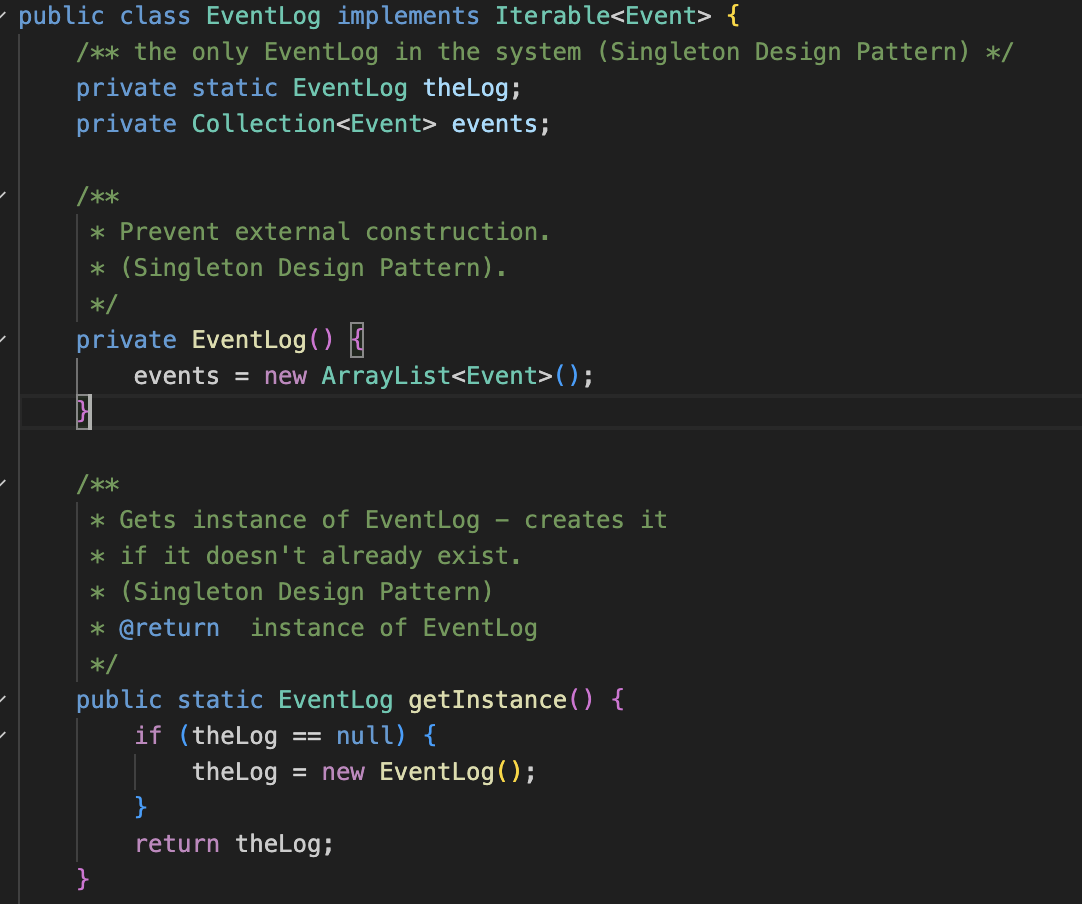
### **Separation of Concerns principle (SoC)**

It is almost the same as SRP but we want to point out that classes JsonReader and JsonWriter are good examples of reading and writing decoupling. That increases testability because of code modularity and encapsulation because we don’t store all information about reads and writes within one module.



### **Singleton Pattern**

EventLog class is the only class with this principle as long as we want to have unified logger for the whole application that is why we need only one instance to be created.



## 

## 

## **Program Flow**

Our program efficiently uses ArrayLists and different classes, which separate the workflow into many subfunctions. In this section of the documentation, we will present the sequence diagrams for all main functions and explain how multiple classes interact.

### **ShowSchedule flow**

A screenshot of a computer

Description automatically generated

As we can see from this sequence diagram, many classes and ArrayLists divide the processes for this function. First, the Menu object gets the input of the room and requests the ListRooms object to get the needed StudyRoom object. ListRooms, in its turn, receives this object from its ArrayList allRoomList.

Then, the Menu object queries StudyRoom to get its schedule. The StudyRoom object then gets all the TimeSlot things from its ArrayList and requests them in the loop to receive their status. If the TimeSlot is taken, StudyRoom gets the name of the user who booked the room for that time slot. In the end, it returns the output with the full schedule to the Menu object.

This is what the code looks like:

Menu.java: A computer screen with colorful text

Description automatically generated

StudyRoom.java:

A computer screen shot of a program code

Description automatically generated

### **BookStudyRoom flow**

A screenshot of a computer

Description automatically generated

As we can see, the same classes interact with each other to give the desired result.

First, the Menu object receives the room and timeslot as input, which the user wants to book. Then, it requests the ListRooms object to get the object of the StudyRoom class. For that, the ListRooms object uses its ArrayList.

Then, the Menu object calls the getAvailability(timeslot) function to check whether the StudyRoom is available for that time slot. The StudyRoom object gets the desired time slot from its ArrayList and receives the status from the TimeSlot object.

In the end, if the time slot in that room is available, the Menu object receives the name of the user as input and books the room for the time slot for that Username.

This is how the process looks in the code:

Menu.java:

A computer screen shot of a program code

Description automatically generated

StudyRoom.java:

A screen shot of a computer code

Description automatically generated

A screen shot of a computer program

Description automatically generated

As it can be seen from the code, we also make use of the EventLog class to log all the events that have happened in the current session. In the end, we could see the events the user called during his session. It can be helpful in monitoring, debugging and understanding the flow.

### **DeleteBooking flow**

A screenshot of a computer

Description automatically generated

This flow works similarly to the bookingRoom flow.

First, the Menu object shows the schedule to the user.

The Menu object receives the room and timeslot from the user input.

Then, it gets the desired StudyRoom object from the ListRooms object and calls the deleteTimeSlot(timeslot) method of the StudyRoom class.

In turn, StudyRoom receives the desired TimeSlot object from its ArrayList and calls the delete() method on that object.

This is the code for this function:

Menu.java:

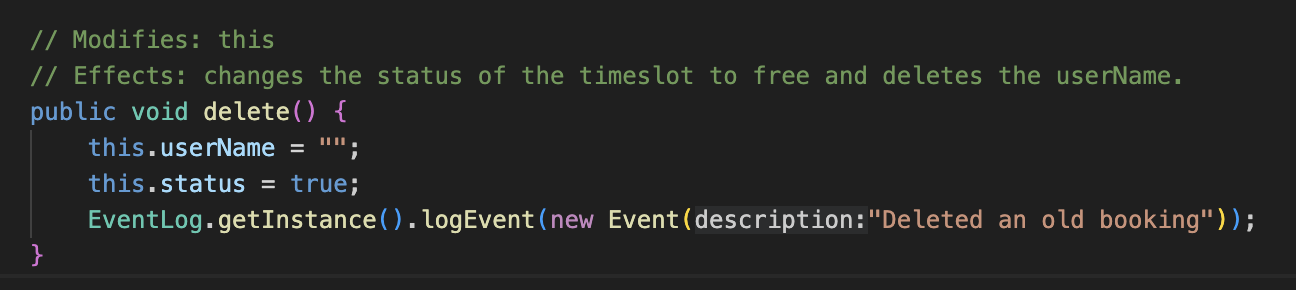
A computer screen shot of a program code

Description automatically generated

StudyRoom.java:

A screen shot of a computer code

Description automatically generated

TimeSlot.java: 

As we can see, to “delete” the booking, we set the username for the time slot as an empty string and set its status to true. It would show that the time slot is free in the future. We also use event logging, just like in the previous flow.

### **ChangeBooking flow**

A screenshot of a computer

Description automatically generated

In this flow, the first steps look just like in the previous flow. However, the difference starts from the 10th step.

As we can see, in the 4th step, the Menu object gets the oldTimeSlot from the user input. Then, in the 10th step, it gets the name of the user, who booked that time slot. After getting the username, it proceeds to delete the time slot as in the previous flow.  
  
After that, the Menu object receives the new time slot from the user and proceeds to book the time slot. For the name attribute, it uses the name that it received from the oldTimeSlot. This flow prevents any unneeded and redundant user input.

This is the code:

Menu.java



This code doesn’t need any further explanations, as it makes use of previous flows.

### **Persistence flow**

For persistence purposes (the booked rooms appear booked when the program launches the next time), we make effective use of JSON objects. The idea was to save all the data as a local JSON object and then retrieve the data on the launch.  
  
A screenshot of a computer

Description automatically generated

As we could see from the previous flows, there was a function saveData() that was called in the end of the functions. This function saves the modified data into a JSON object.   
  
To do that, we use JsonWriter and PrintWriter classes. First, we open the file, where we want to save the data. Then, we write to this file the allRoomList ArrayList. For this reason, we came up with the specific structure of the JSON object. Then we use the PrintWriter class instance to write the JSON object into the file.

When we want to load the data, we call the read() function. JsonReader class is designed to read the file and return allRoomList ArrayList to the Menu object. Reading flow is very complex, as we divide the JSON object into different parts, such as StudyRooms and TimeSlots to correctly retrieve the data. This process is done in the loop.

This process becomes clearer with the code:

Menu.java

A screen shot of a computer program

Description automatically generated

JsonWriter.java:

A screen shot of a computer program

Description automatically generated

First, we start with converting ListRooms into JSON objects, which then will call functions of other classes that also convert that class objects into JSON objects.

ListRooms.java:

A computer screen shot of a program code

Description automatically generated

StudyRoom.java:

A computer screen shot of a program code

Description automatically generated

TimeSlot.java:

A screen shot of a computer code

Description automatically generated

We successfully converted the objects of all classes into JSON objects and saved into the file. Now, we try to load the data from the file.

Menu.java:

A screen shot of a computer program

Description automatically generated

JsonReader.java:

A computer screen shot of a program code

Description automatically generated

ListRooms.java:

A computer screen with text on it

Description automatically generated

A screen shot of a computer program

Description automatically generated

StudyRoom.java:

A screen shot of a computer program

Description automatically generated

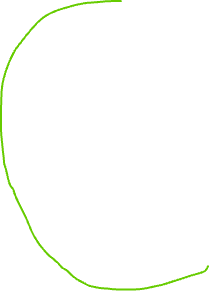
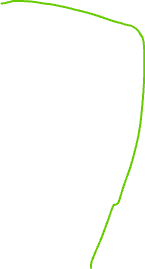
As we can see, the process is divided between many objects in order to correctly convert the session data into JSON objects and make it easily retrievable.

# 

# **Testing Report**

### **Module Organization**

A diagram of a software company

Description automatically generated

The above diagram describes the module structure of the system.

EventLog and Event are 2 lowest-level modules that are responsible for logging.

TimeSlot is the smallest unit managed by the system. Whereas StudyRoom and ListRooms are used to manage it.

JsonWriter and JsonReader are part of the persistence package, which is responsible for saving and managing booking data in json files.

The app uses Swing GUI extensively in the form of MainGUI, BookGUI, ScheduleGUI, ChangeGUI, and DeleteGUI.

However, testing automation for GUI is out of scope of the course. Therefore, we will focus only on testing the highlighted modules using Junit.

This organization allows us to develop a comprehensive testing plan and determine necessary modules to and branches to be tested.

### **Testing Strategy**

We have decided to use Bottom-up approach as our testing strategy.

A diagram of a study room

Description automatically generated

Since the hierarchy is pretty complex, we only have 1 unit test:

* Event

After this unit testing, we started different integration tests, which meticulously test each module, as well as their respective methods.

* Event + EventLog
* Event + EventLog + TimeSlot
* Event + EventLog + TimeSlot + StudyRoom
* Event + EventLog + TimeSlot + StudyRoom + ListRooms
* Event + EventLog + TimeSlot + StudyRoom + ListRooms + JsonWriter
* Event + EventLog + TimeSlot + StudyRoom + ListRooms + JsonWriter + JsonReader

Finally, the system test was performed manually, as it involves GUI.

### **Branch Coverage**

In the testing process, we will focus on 2 types of control-flow testing – statement and branch coverage. Statement coverage considers only whether each line was executed. On the other hand, branch coverage is more complex and checks whether every branch has been executed. Here are examples in the class Event.

Test Function: equals

|  |
| --- |
| Program Source:  public boolean equals(Object other) {  if (other == null) {  return false;  }  if (other.getClass() != this.getClass()) {  return false;  }  Event otherEvent = (Event) other;   return (this.dateLogged.equals(otherEvent.dateLogged)  && this.description.equals(otherEvent.description)); } |
| Truth table for the test cases T2-T4 from top to bottom:   |  |  |  |  |  | | --- | --- | --- | --- | --- | | other | this | other == null | Other.getClass() != this.getClass() | return | | null | Event(“Sensor open at door”) | true | N/A (isn’t executed) | false | | “Not an event”: String | Event(“Sensor open at door”) | false | true | false | | Event(“Sensor open at door”) | Event(“Sensor open at door”) | false | false | true | |

### **Test Cases**

We only tested functionality without GUI, which is Persistence and Model. That is why, we will provide coverage for them.

Statement Coverage:



Branch Coverage:  


|  |  |  |  |
| --- | --- | --- | --- |
| **Event.java** | | | |
| **Test case ID** | **Input Description** | **Expected Result** | **Actual Output** |
| T001 | Get the event description | Returns description “Sensor open at door” | As expected |
| T002 | Get the event date | Returns current date | As expected |
| T003 | Check whether event is equal null | Returns false | As expected |
| T004 | Check whether event is equal to string “Not an event” | Returns false | As expected |
| T005 | Check whether event is equal to Event(“Sensor open at door”) | Returns  true | As expected |
| T006 | Get the event hashcode | Returns  Correct hashcode | As expected |
| T007 | Get the event toString | Returns  Correct string representation | As expected |
| T008 | Test setter function for description by getting after set | Returns correct description | As expected |
| T009 | Test setter function for Date by getting after set | Returns correct time | As expected |

|  |  |  |  |
| --- | --- | --- | --- |
| **EventLog.java** | | | |
| **Test case ID** | **Input Description** | **Expected Result** | **Actual Output** |
| T010 | Add Event(“A1”), Event(“A2”), Event(“A3”), and then retrieve | Returns  The correct events in correct order | As expected |
| T011 | Clear the log | EventLog becomes empty | As expected |
| T012 | Test if the instance of event log changes by comparing instances | Instances are equal | As expected |
| T013 | Test iterator of the event log by creating and removing the event and checking the event pointed by iterator | Correct event | As expected |

|  |  |  |  |
| --- | --- | --- | --- |
| **ListRooms.java** | | | |
| **Test case ID** | **Input Description** | **Expected Result** | **Actual Output** |
| T014 | Add StudyRoom(“test”) and retrieve | Adds the study room in the container | As expected |
| T015 | Add StudyRoom(“test”), StudyRoom(“test”) and get size | Returns the correct size of the listRooms (2) | As expected |
| T016 | Add StudyRoom(“test”), StudyRoom(“test”) and retrieve the list | Returns  the correct list of rooms | As expected |
| T017 | Test constructor by getting number of rooms from the list | number == 4 | As expected |
| T018 | Add new StudyRoom, check number of rooms and retrieve new room | Room is added and number is incremented | As expected |
| T019 | Test getRoom by getting room by index 2 | Retrieved room’s name is “X300” | As expected |
| T020 | Test toJson by creating JSON, getting jsonArray “data” and checking its length | Length of JSON Array is 4 | As expected |
| T021 | Try getting room of invalid index and see if exception is thrown | IndexOutOfBoundsException is thrown | As expected |
| T022 | Test adding room at specified position | Returns correct list | As expected |

|  |  |  |  |
| --- | --- | --- | --- |
| **StudyRoom.java** | | | |
| **Test case ID** | **Input Description** | **Expected Result** | **Actual Output** |
| T023 | Get the list of empty schedule | Returns  A list of “Free” rooms | As expected |
| T024 | Book (9, “User1”); (17, “User2”) and get the list of schedule | Returns  A list of “Free” rooms except 2 | As expected |
| T025 | Book (9, “User”) and see availability at time 9 | Returns  false | As expected |
| T026 | Book (17, “User”) and then delete the booking and check availability | Returns  true | As expected |
| T027 | Test getName by getting room’s name | Returns “Test Room” | As expected |
| T028 | Test change Time slot by creating a new timeslot, changing timeslot at 0 to new and getting timeslot at 0 | Returns new time slot | As expected |
| T029 | Test getTimeSlotUser of the unbooked timeslot | Returns “” | As expected |
| T030 | Get availability of unbooked timeslot | Returns true | As expected |
| T031 | Test toJson by booking timeslots and checking json elements | Returns correct JSON Object | As expected |
| T032 | Book already booked timeslot and verify that the booker remains the same | Returns initial name | As expected |
| T033 | Delete already free timeslot to verify its username and status don’t change | username == “” and status = True | As expected |
| T034 | Change timeslot with invalid index >11 and see if exception is thrown | IndexOutOfBoundsException is thrown | As expected |
| T035 | Test toJson with no booked timeslots and check Json Object | Returns correct Json Object | As expected |

|  |  |  |  |
| --- | --- | --- | --- |
| **TimeSlot.java** | | | |
| **Test case ID** | **Input Description** | **Expected Result** | **Actual Output** |
| T036 | Book a slot and get the user | Returns  “user” | As expected |
| T037 | Book a slot, delete the book and get the user | Returns  “” | As expected |
| T038 | Test constructor by getting username and status | Returns username == “” and status is True | As expected |
| T039 | Test parameterized constructor by giving name and status | Returns username == “John” and status is False | As expected |
| T040 | Test toJson by booking timeslot and checking Username attribute | Returns correct Json Object | As expected |
| T041 | Test getStatus by getting it, then booking and getting it again, then deleting booking and getting it again | Returns True, False, True | As expected |
| T042 | Test getUserName of free and booked timeslot | Returns “” and “User” | As expected |
| T043 | Test if timeslot can be booked by empty string for username | username == “” and status = false | As expected |

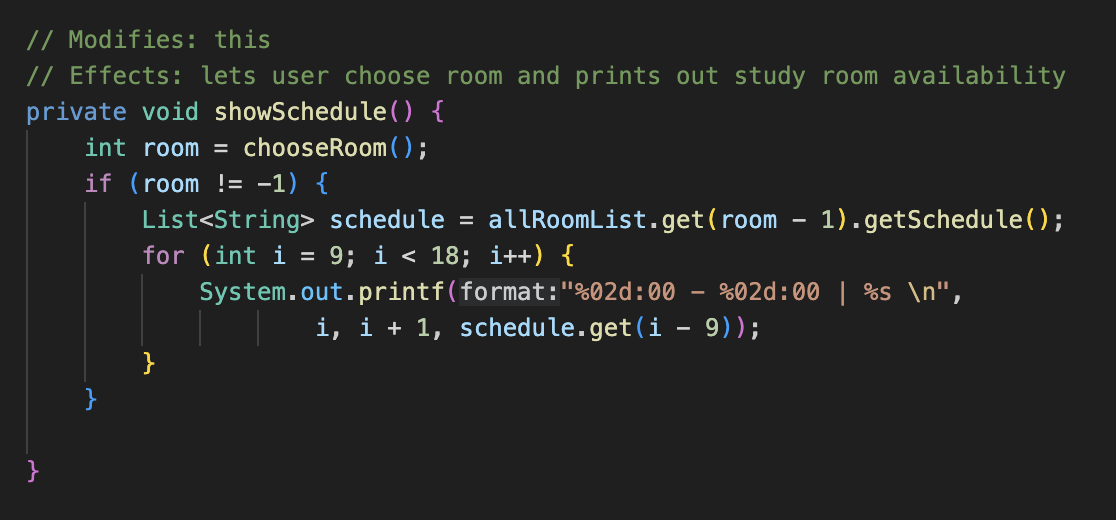
|  |  |  |  |
| --- | --- | --- | --- |
| **JsonReader.java** | | | |
| **Test case ID** | **Input Description** | **Expected Result** | **Actual Output** |
| T044 | Read a non-existent file | Raises IOException | As expected |
| T045 | Read an empty file and get number of rooms | Returns 0 | As expected |
| T046 | Read a file and retrieve rooms | Returns  A list of 2 rooms | As expected |

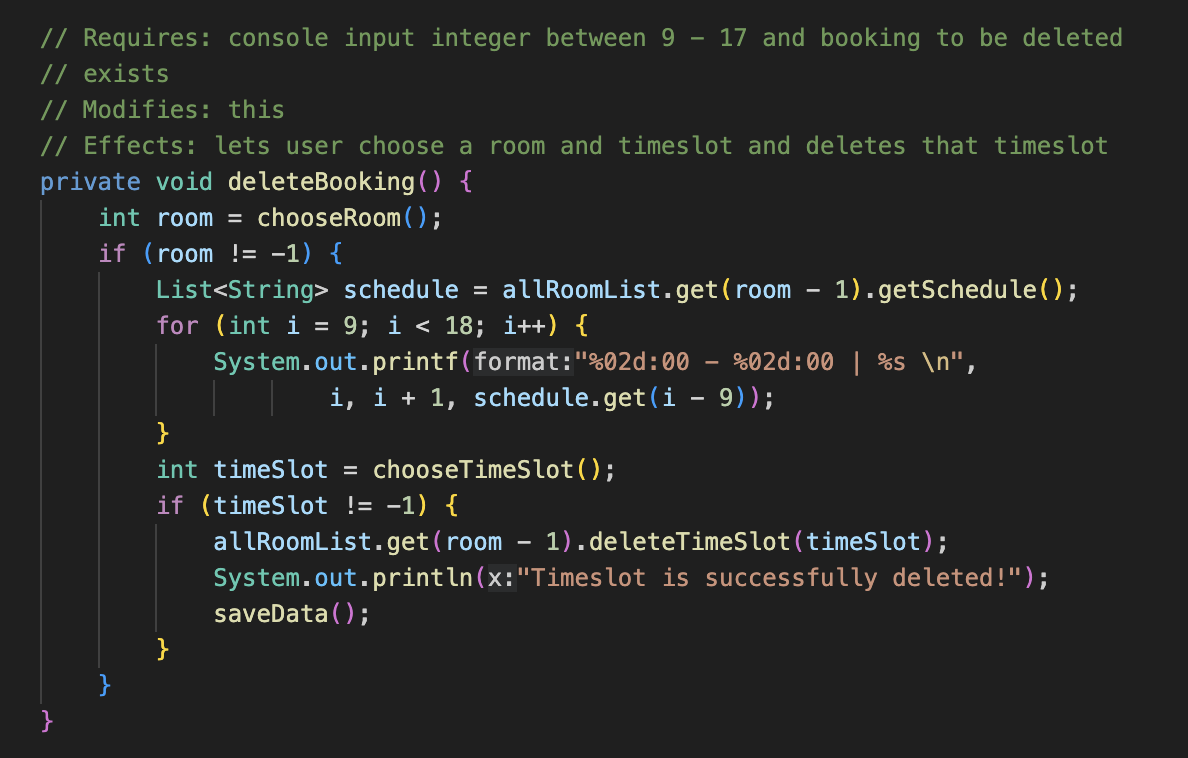
|  |  |  |  |
| --- | --- | --- | --- |
| **JsonWriter.java** | | | |
| **Test case ID** | **Input Description** | **Expected Result** | **Actual Output** |
| T047 | Write a ListRooms to a file, retrieve it back and check size | Returns 0 | As expected |
| T048 | Write a ListRooms to a file, retrieve it back and check size | Returns 0 | As expected |
| T049 | Write to a file with an illegal name | Raises IOException | As expected |
| T050 | Write a ListRooms to a file, retrieve it back and check size | Returns 0 | As expected |
| T051 | Writes a ListRoom of 2 StudyRooms, retrieves it back and gets Rooms | Returns a list of 2 StudyRooms | As expected |

# **Code refactoring**

After each update to the code, we did code refactoring. We fixed the variable and methods’ naming, divided complex logic and made efficient use of separating functions. As changing the naming is straightforward, we will give an example of the latter.

Code before refactoring:



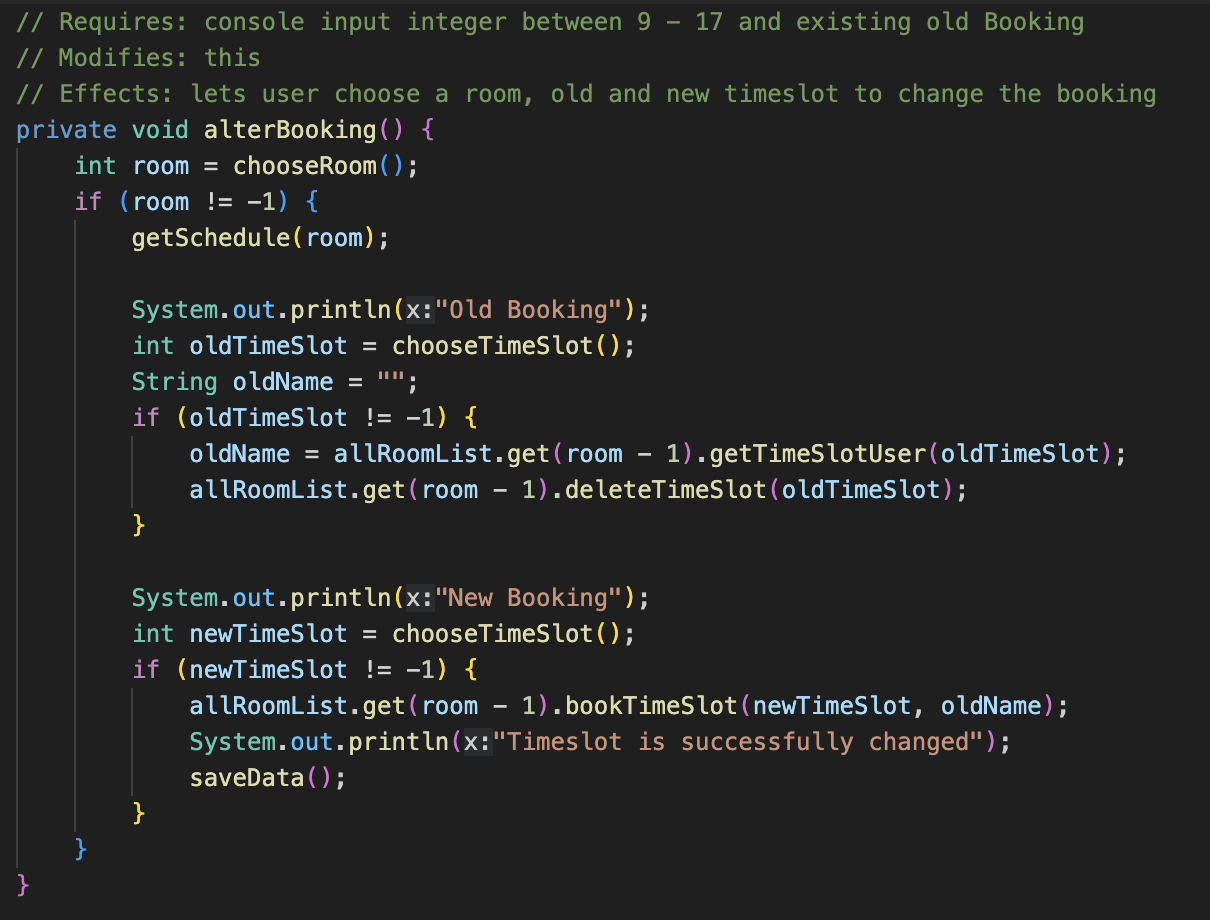
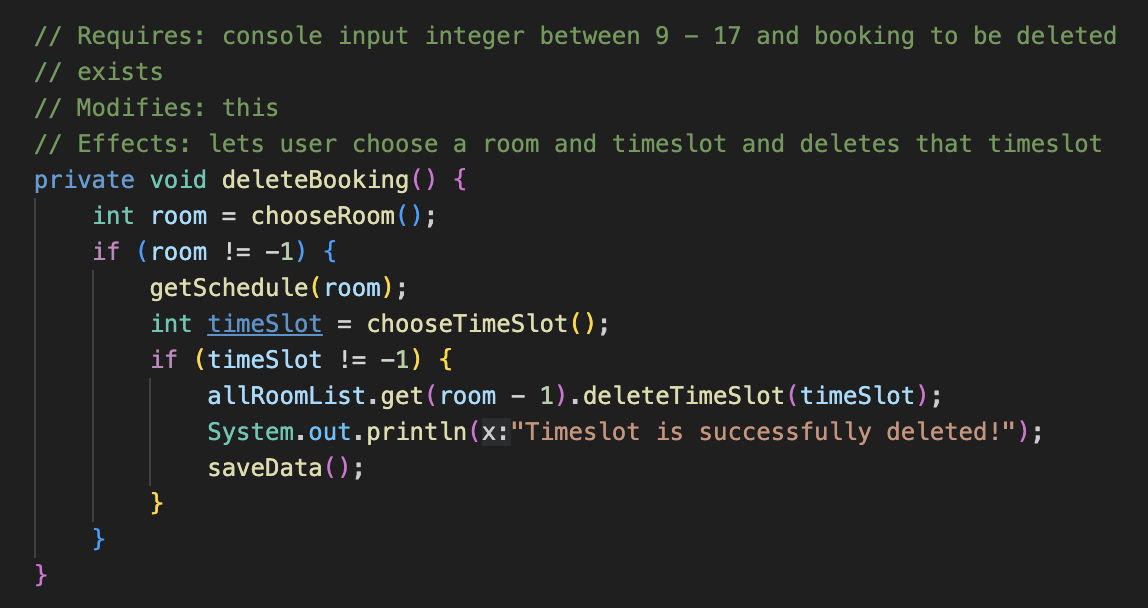
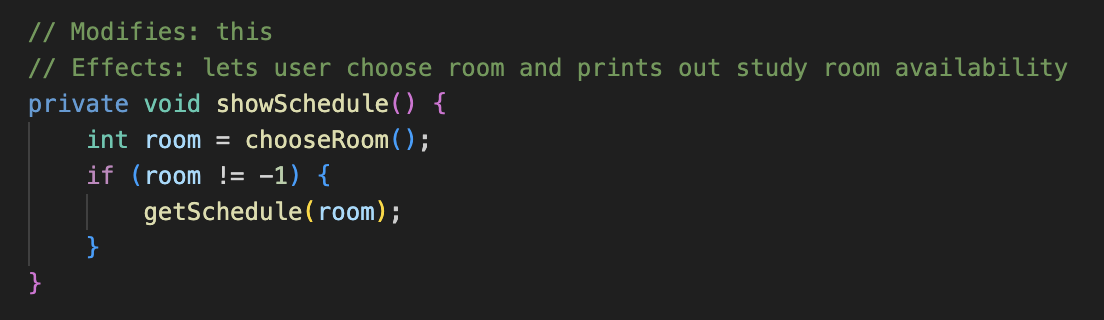




As you can see, all three functions use the same code for getting the schedule. Therefore, we decided to extract it into a separate function and call this function.

Code after refactoring:





Now, these functions call another method called “getSchedule”, which takes room as an argument and does the same, what was done before.

Importantly, thanks to test-driven approach, we could always maintain code and prevent it from malfunctioning.