CS4125 - System Analysis & Design

Group Project Report

Team Members:

16172639: Harry deLongue

19269749: Oliver Nagy

18238637: Emmet Browne

16189973: Ryan O'Keeffe

https://github.com/deLongueHarry/CS4125Proj22

Table Of Contents

1. Narrative Description

For our project we chose to work on a Gym booking/management system The system works as follows:

The system records reservations and bookings made by users as well as user information. Additionally, it would give management and admins the ability to change user information, memberships, and activity and class kinds.

There is a login/signup page where a user can login or if they haven’t registered, they can register . This login page would also serve as a login page for admins/staff with special logins.

There is a login/signup page where users can create a new account, or login with existing credentials. To create a new account, a user will have to supply the following information:

- Full name
- Mobile number
- Email address
- New password
- Student ID

This data will help us organise users within our database.

Admins will also be able to login with special credentials, where they can access an admin table. From here they can add/remove users or edit information.

User: Once the user signed in/ registered successfully and their membership is active, they will be directed to the home page where they will have a number of options including booking classes , managing membership or editing their profile.

Booking classes: Users will be able to book slots throughout the week for the gym/pool facilities. This will show what time slots are unavailable.

Managing membership: From here, the user can change payment details, change tier level or cancel membership.

Edit Profile: This will allow the user to change personal details, such as email address, mobile number etc.

Management/Admin: Once the admin has signed in they will be shown a home page with the following options to choose from : Manage members, manage classes and activities.

We will also have a tiering system available for our members. Members can choose between a bronze, silver and gold membership. This will give different options to members, for example gold members will have full access to gym / pool / classes.

2. Software Lifecycle Model

For our project we examined a number of different software lifecycle models.

2.1: Waterfall

The waterfall model, also known as the linear sequential lifecycle model, is one of the earliest SLC's to be used in software development. It is of sequential nature, meaning that new development phases begin after the previous has been completed. Due to this, no working software is produced until late into the life cycle for the project. It is for this reason that we chose not to follow this SLC.

2.2 V-Shaped Model

We next looked at the V-model. In the V-model, the development and testing phases run in parallel.The V-model has two sides, the Validation phase and Verification phase, which are connected together by the coding phase. The model's simplicity offers an advantage, as it is easy to understand and apply. However, each phase is completed iteratively. Because of this we chose not to use this SLC for our project.

2.3: Iterative Model

In the iterative model, the project begins with a small set of system requirements that are iteratively expanded upon until the project's completion. With each iteration, new functional requirements are added with new versions of the software being produced.

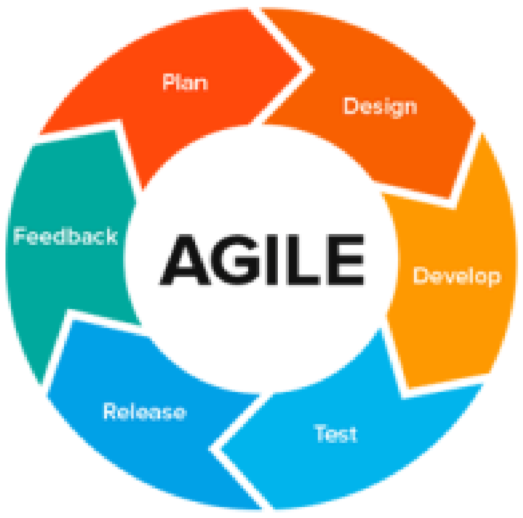
With this model, working software can be produced early in the project's life cycle. It's also easier to measure progress as testing/debugging is easier to implement with smaller batches of code.

However, it has its drawbacks. System Architecture issues may arise due to a lack of requirements gathered from the beginning of the project's life cycle. This model is also not well suited for making any changes to requirements should changes be necessary. Finally, designing each iteration may result in the need for a complete definition of all requirements, essentially going against the definition of the iterative model.

It is for these reasons that we chose not to pursue this software lifecycle model for our project.

2.4: Agile Model

We chose to go with the Agile model for our project due to the flexibility offered. In Agile, the project is split into several builds, or sprints. Each sprint is incremented in terms of features. This means that in the first sprint, basic functionality is implemented. Future sprints then add more complex functionality and so on.
With this model, software is implemented early on in the project's life cycle, which is exactly what we want due to time constraints.
It is for these reasons that we decided following an Agile development cycle would be best suited for our project.



"In software development the primary measure of progress has to be working software that meets the needs of the end users."
— Larry Apke, Understanding The Agile Manifesto: A Brief & Bold Guide to Agile

3. Project Plan

3.1 Planning Summary

We have arranged to meet 2 times per week, and set our several short term plans, using the project plan timetable shown in the specification.

We have created a GitHub repository and we plan to communicate through our discord group chat. This will help us to make sure any commits made to the master branch are approved by the other members.

We will be using VsCode as our IDE, as we are all equally familiar with it.

3.2 Project Roles

	Role	**Description**	**Designated Team Member**
1	Project Manager	Sets up group meetings, gets agreement on the project plan, and tracks progress.	**Emmet**
2	Documentation Manager	Responsible for sourcing relevant supporting documentation from each team member and composing it in the report.	**Harry**
3	Business Analyst / Reqs. Engineer	Responsible for section 4 - Requirements.	**Emmet**
4	Architect	Defines system architecture.	**Oliver**
5	System Analyst	Creates conceptual class models.	**Ryan**
6	Designer	Responsible for recovering design time blueprints from implementation.	**Ryan**
7	Technical Lead	Leads the implementation effort.	**Harry**
8	Programmers	Each team member develops at least 1 package in the architecture..	**ALL**
9	Tester	Coding of automated test cases.	**Oliver**
10	Dev Ops	Must ensure that each team member is competent with development infrastructure, e.g. GitHub, etc.	**Oliver**

3.2 Project Plan: Timetable

Week	Workflow
3	Setup team roles, agree on scenario, setup GitHub repository, research on existing projects
4	Requirements
5	Analysis
6	High level architecture
7	Coding Iteration 1: Basic infrastructure and 2 key use cases.
8	Coding Iteration 2: 2 more use cases.
9	Coding Iteration 3: Another use case and MVC (GUI)
10	Coding Iteration 4: Another use case and added value.
11	Overrun
12	Architecture and design recovery

3.2 Co-op Experiences

Names	Co-op Job	Experience
Harry Delongue	IT Support / Management	System Admin / Security
Ryan O’Keeffe	3D Printing Lab Manager	Arduino
Oliver Nagy	ITD Learning Centre	System Admin / Lab work
Emmet Browne	Intern Software Develop	Java, Agile

4. Software Requirements

Functional Requirements

The Customer must be able to:

- Create an Account
- Log into the account
- Delete their account
- Make a booking
- View their bookings
- Change membership tier type

The Admin must be able to:

- View customer accounts
- modify/delete customer accounts
- Change tier options

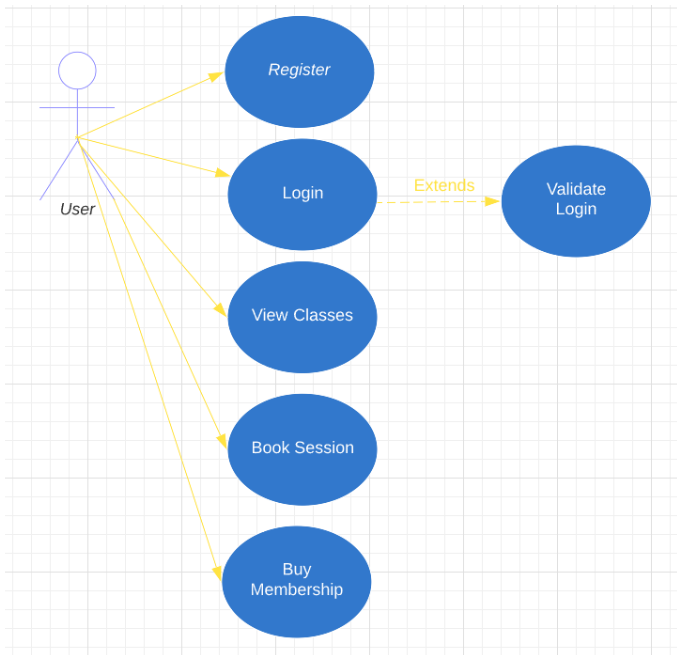
Non-Functional Requirements

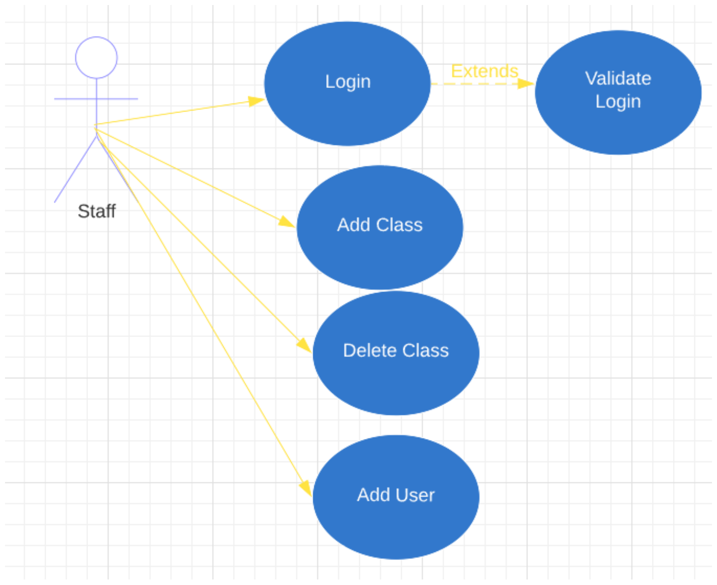
- System should be easy to navigate

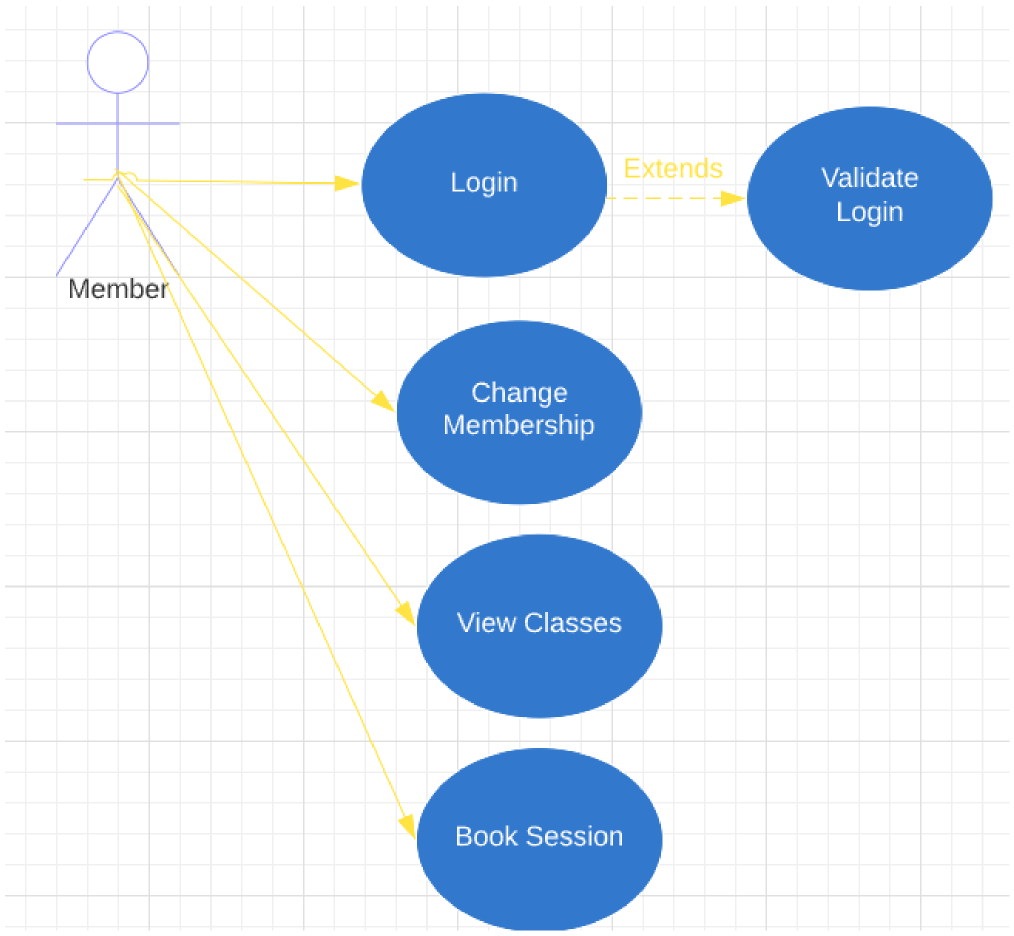
- System should be responsive

- System should be reliable/consistent

Use Case Diagram






Member
Login
Extends
Validate Login
Change Membership
View Classes
Book Session

USE CASE	***REGISTER USER***
goal	For a member of staff to add a new member to the system
scope/level	Company
Success end condition	New user added to database
Fail end condition	Staff cannot add new user
Primary, Secondary Actors	Staff, User
Trigger	New customer wants to register
Normal flow	1.New customer requests to be added to system 2.Staff records all of the users details 3.Staff assigns user new memberID 4.Staff adds member to database
variations	User wishes to close account
priority	top
performance	10 mins

USE CASE	***UPGRADE MEMBERSHIP***
goal	For a user to upgrade their membership
scope/level	Company
Success end condition	User membership is changed so prices lowered
Fail end condition	User can’t successfully upgrade membership
Primary, Secondary Actors	User
Trigger	User selects upgrade membership
Normal flow	1. User goes to view memberships 2. Selects membership tier they would like to purchase 3. They click purchase membership 4. Enter Details/Use Saved Company successfully receives payment 5. User membership tier is changed and prices effected
variations	User wishes to cancel membership
priority	top
performance	5 mins

USE CASE	***ADD CLASS***
goal	For a member of staff to list a new class on the system
scope/level	Company
Success end condition	New class added
Fail end condition	Staff cannot add new class
Primary, Secondary Actors	Staff, User
Trigger	Member of staff chooses option to add new class
Normal flow	1.Staff has a new class to be listed on the system 2.Staff chooses add new class option through UI 3.Staff fills in class attributes such as location, capacity and cost 4.New class is added to the database and viewable to users and members
variations	Staff tries to add a class that conflicts with another
priority	top
performance	10 mins

Tactics to support handling Quality Attributes

We decided to develop our system using Java for multiple reasons. It is a highly portable language, which allows for moving across multiple systems. It is also a modern object-oriented language, which will facilitate the implementation of many of the design patterns that are found within the System Analysis and Design module. Finally, the team is most familiar with Java across the board and having multiple online resources to aid us was seen as an advantage when deciding on which language to select.

Extensibility:

The use of design patterns will help to facilitate the extensibility of our system. Implementation of the Model-View-Controller design pattern (MVC) will allow for future views to be introduced while reusing the existing models and controllers. The factory and state design patterns also help to facilitate extensibility within our system.

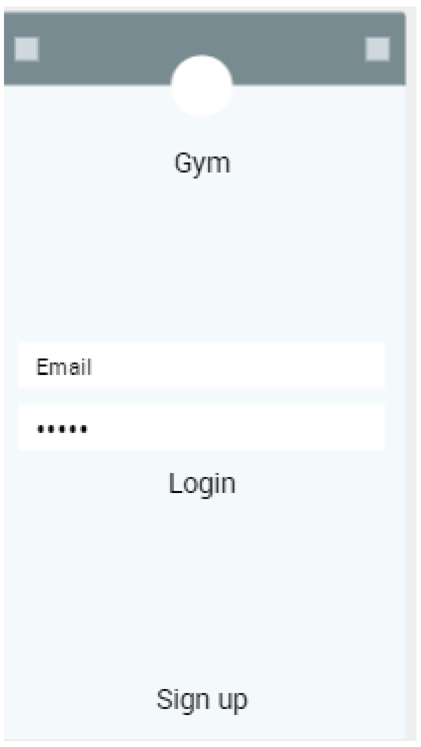
Testability:

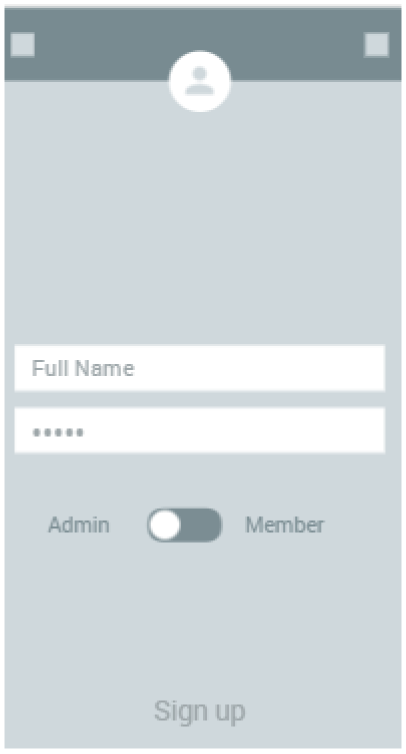
The ability to test the system for bugs is an important feature within System Design, so consciously adding test cases throughout implementation of the system is imperative. This will allow us to ensure that the system functions as desired.

Security:

Ensuring that the system has security elements is imperative to design. To provide some security to the system, adding a database and restricting access to the admin user type only will allow for security within the system.

GUI Prototypes





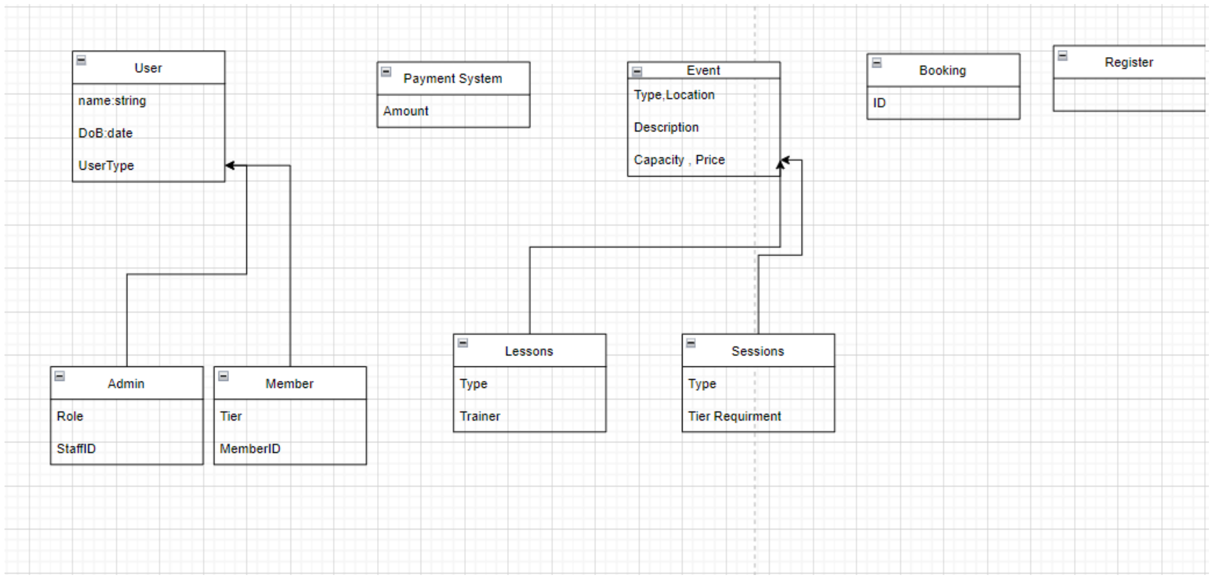
5. System Architecture

For our implementation, we divided the system into three sub-systems: user interface, business, and database. As a result of each subsystem operating independently of the others, the overall system is as maintainable and portable as possible.

6. **Analysis**

Our gym booking system will interface with customers and staff through a UI which they will access through their phone
Staff will be able to login and add new classes that customers can then book.
Staff can also add and remove users as well as edit their membership level.
Members have a unique name, dob, member ID and one of 3 membership tiers on their profile. Members are able to edit their profile, purchase new memberships and Book in for gym or pool sessions.

Class Diagram



7. Transparency and Traceability

Code Breakdown

Written by Oliver

Class Authors

Package	Class	Authors	Lines of Code
BOOKING SYSTEM	Class.java	Emmet & Oliver	54
	Payment.java	Emmet & Oliver	19
	BookingSystem.java	Emmet & Oliver	20
DATABASE	ClassSearchDTO.java	Oliver	58
	UserSearchDTO.java		69
	VerifyLoginDTO.java		40
	WirterDTO.java		32
GUI.Factory	AddClasses.java	Emmet	162
	AdminUI.java	Emmet	197
	BookClassUI.java	Oliver	140
	GUI.java	Oliver	9
	GUIFactory.java	Oliver	32
	MemberShipUI.java	Emmet	185
	MenuUI.java	Emmet	252
	OfferWindow.java	Oliver	81
	RegUI.java	Emmet	209
	UserUI.java	Emmet	150
GUI.STATEDESIGN	GUIContext.java	Oliver	29
	StartAddClass.java	Oliver	23
	StartAdminGUI.java	Oliver	24
	StartBookClass.java	Oliver	23
	StartLoginGUI.java	Oliver	20

	StartMemberGUI.java	Emmet	22
	StartMemberShipGUI.java	Oliver	21
	StartOfferGUI.java	Oliver	22
	StartRegUI.java	Oliver	20
	State.java		12
TESTING	LoginTest.java	Oliver	33
	AddClassTest.java		32
	PaymentHandler.java		26
USER.Decorator	BasicMemberShip.java	Oliver	41
	Benefit.java		40
	GoldMemberShip.java		34
	TierDecorator.java		36
	TierMembership.java		44
			23
USER.Observer	Channels.java	Emmet	64
	Subscribers.java		56
USER	Admin.java	Ryan	15
	Member.java	Ryan	36
	User.java	Ryan	81

Total Code Developed

~2486 lines of code total.

Team Member Contributions

Team Member	***Lines Contributed***
Oliver	***1,157***
Emmet	***1,178***

8. Code Snippets and Descriptions

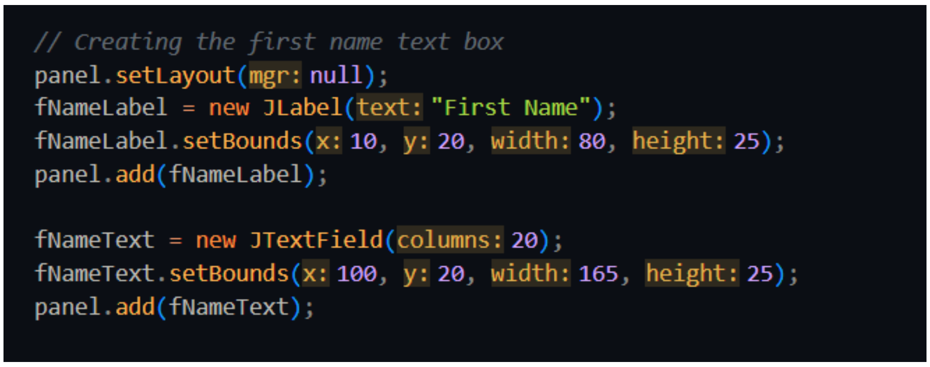
8.1 Model View Controller

Written by Emmet

In developing our Gym Booking System, we implemented a Model View Controller (MVC) architectural pattern. We decided to use Java swing to develop our UI, which allowed us to successfully achieve the MVC structure. This pattern allows for a clear separation of concerns between the user interface, the data that it must display and the business logic that operates on that data. The MVC consists of three main components.

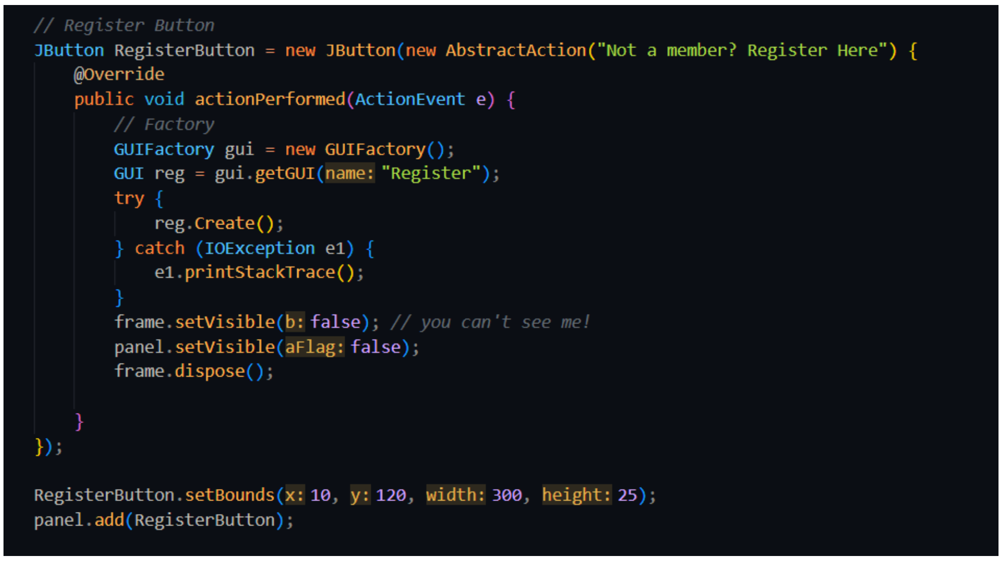
Model:

The model encompasses the state data for each component. For example, in our register page (RegUI.java) we create multiple text boxes that the user can enter their details into. Here you can see the 'first name text box' being initiated.



```java
// Creating the first name text box
panel.setLayout(mgr: null);
fNameLabel = new JLabel(text: "First Name");
fNameLabel.setBounds(x: 10, y: 20, width: 80, height: 25);
panel.add(fNameLabel);

fNameText = new JTextField(columns: 20);
fNameText.setBounds(x: 100, y: 20, width: 165, height: 25);
panel.add(fNameText);
```

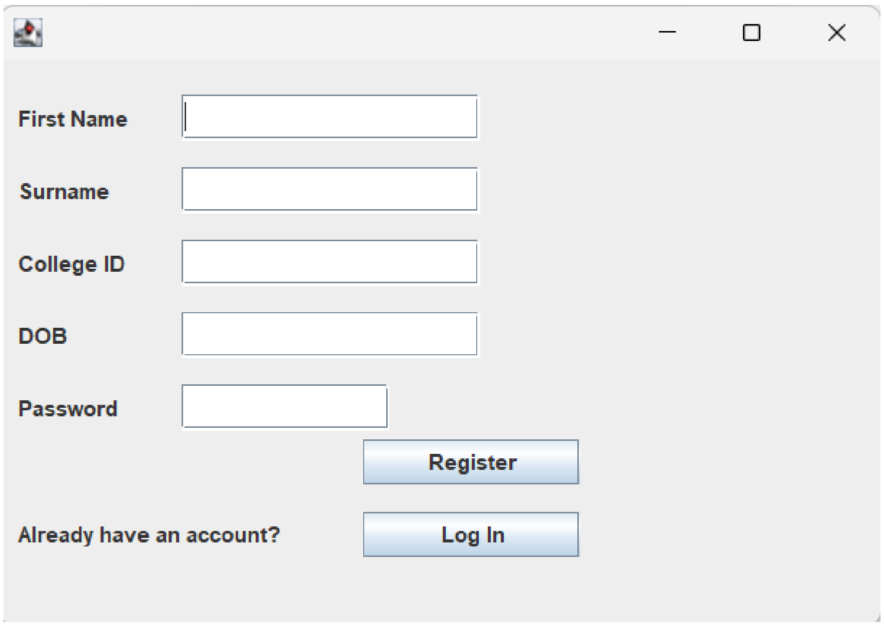
Another example of this would be creating buttons in the UI, that the user can press on to perform different activities. In this example, we look at the login class (LoginUI.java) where the user logs in. Here, the user has the choice to login, or register as a new user.



```java
// Register Button
JButton RegisterButton = new JButton(new AbstractAction("Not a member? Register Here") {
    @Override
    public void actionPerformed(ActionEvent e) {
        // Factory
        GUIFactory gui = new GUIFactory();
        GUI reg = gui.getGUI(name: "Register");
        try {
            reg.Create();
        } catch (IOException e1) {
            e1.printStackTrace();
        }
        frame.setVisible(b: false); // you can't see me!
        panel.setVisible(aFlag: false);
        frame.dispose();

    }
});

RegisterButton.setBounds(x: 10, y: 120, width: 300, height: 25);
panel.add(RegisterButton);
```

View:

The view refers to how you see the component on the screen. The aforementioned text boxes or buttons are shown on the UI. For example, this is the Register UI. We can see the text boxes are shown, and also allow for the user to type text into them.

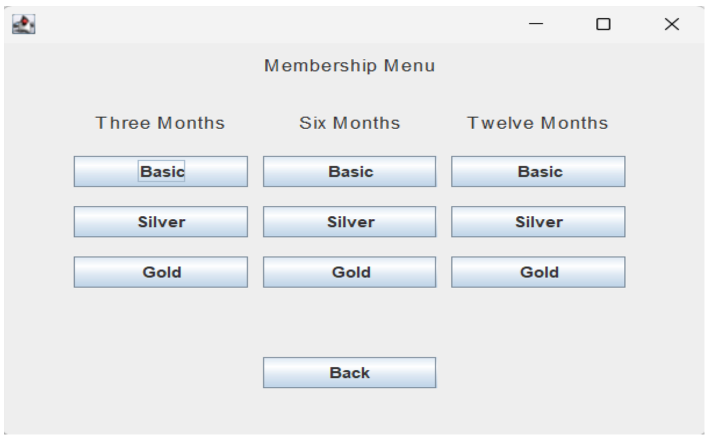


The view that Java Swing creates, allows you to adjust the size of the window, and also comes with minimise, maximise and a close button on the top right corner. I was able to add multiple text boxes, with labels beside them to inform the user what details to input. We can also see a good example of buttons here with ‘Register’ and ‘Login’ which would allow the user to either save their account, or login in with pre-existing details.

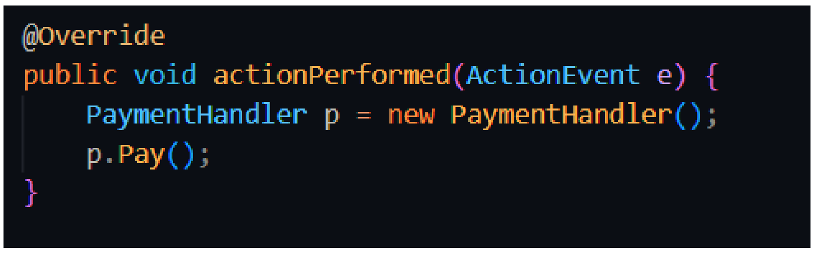
<u>**Controller:**</u>

<u>Written by Emmet</u>

The controller is the portion of the user interface that dictates how the component interacts with events. As this UI is generally simple, our main examples of this were mouse clicks on buttons, or entering text into text boxes. For example in our ‘Membership Menu’ (Membership.java) we have buttons that allow the user to buy a membership for our gym. You can choose any tier of membership, for different times (3 months, 6 months, 12 months).

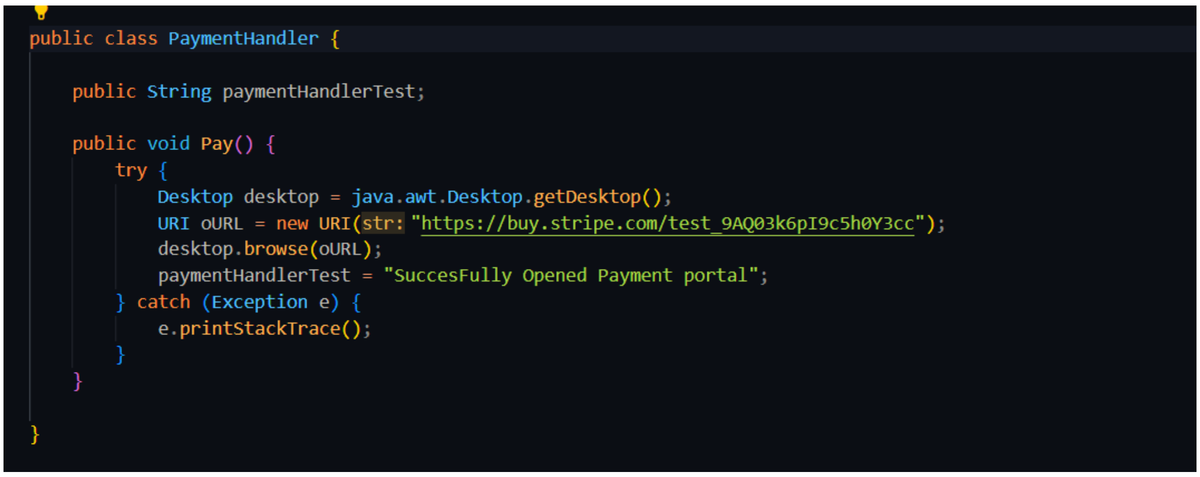


Clicking any of these buttons will trigger an event. In this example, these buttons will open a Stripe payment page, where the user can buy a gym membership.

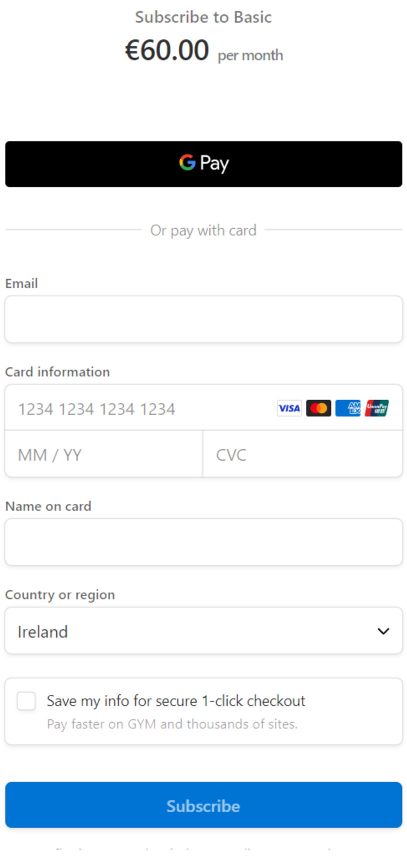


```
@Override
public void actionPerformed(ActionEvent e) {
    PaymentHandler p = new PaymentHandler();
    p.Pay();
}
```

We can see here that the event is handled once the button is pressed, a new PaymentHandler is created, and then the pay method is called. Pressing this button will bring you to the online payment screen.



```
public class PaymentHandler {

    public String paymentHandlerTest;

    public void Pay() {
        try {
            Desktop desktop = java.awt.Desktop.getDesktop();
            URI oURL = new URI(str: "https://buy.stripe.com/test_9AQ03k6pI9c5h0Y3cc");
            desktop.browse(oURL);
            paymentHandlerTest = "SuccesFully Opened Payment portal";
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

}
```



Subscribe to Basic

€60.00 per month

Email

Card information

1234 1234 1234 1234

MM / YY CVC

Name on card

Country or region

Ireland

Save my info for secure 1-click checkout

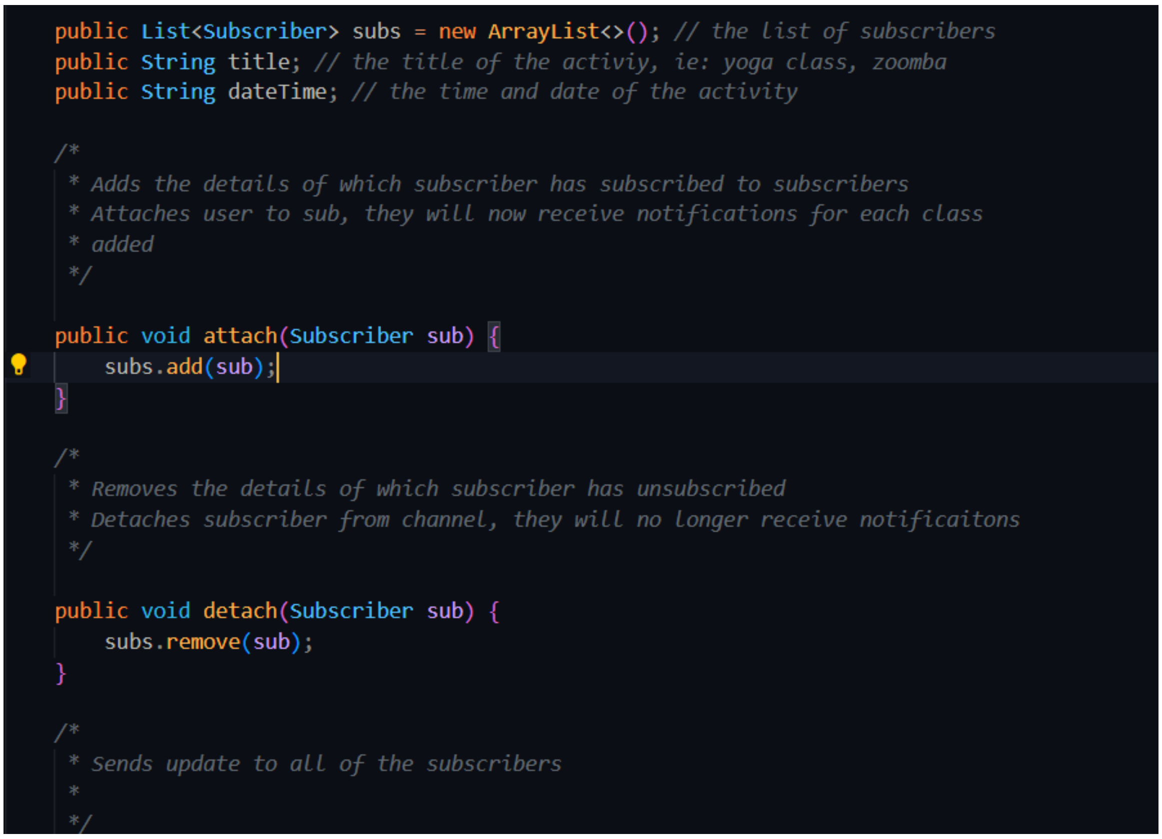
Pay faster on GYM and thousands of sites.

Observer Design Pattern:

Written by Emmet

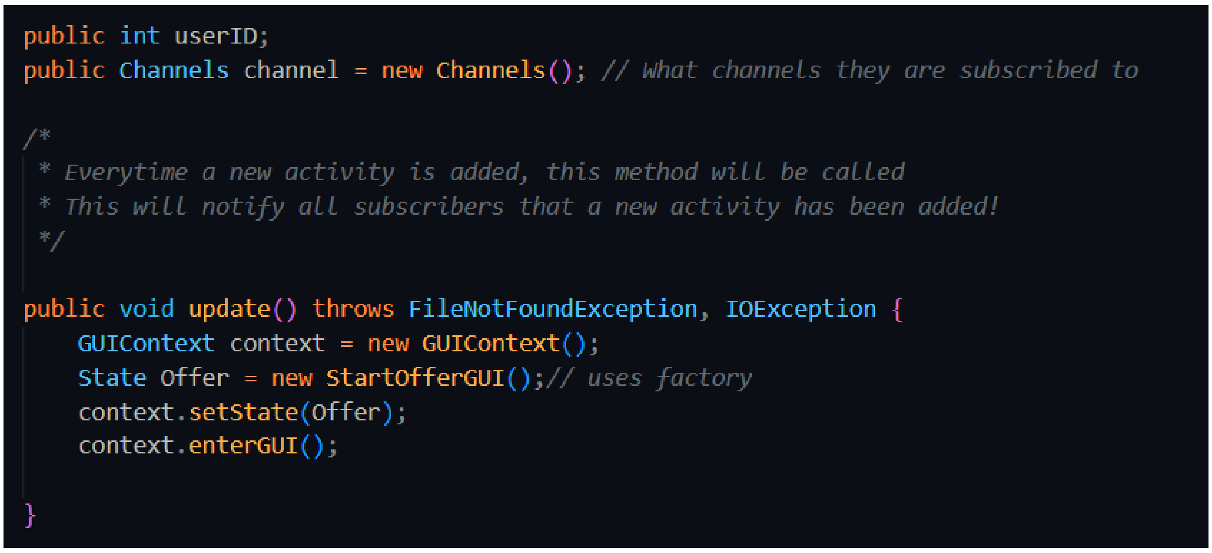
The Observer design pattern suggests that you add a subscription mechanism to your system so that objects can individually subscribe to or unsubscribe from a ‘channel’ of events. To implement this, I decided on a ‘gym offers’ window, where users could subscribe to a ‘channel’ that would inform them of gym offers.

In our example we automatically subscribed (attached) every user to this channel, this means every user will get notified about our special offers. If we had more time, we definitely would have implemented a system where users can subscribe / unsubscribe from this, or add some more functionality such as ‘Gym News’.



```
public List<Subscriber> subs = new ArrayList<>(); // the list of subscribers
public String title; // the title of the activiy, ie: yoga class, zoomba
public String dateTime; // the time and date of the activity

/*
 * Adds the details of which subscriber has subscribed to subscribers
 * Attaches user to sub, they will now receive notifications for each class
 * added
 */

public void attach(Subscriber sub) {
    subs.add(sub);
}

/*
 * Removes the details of which subscriber has unsubscribed
 * Detaches subscriber from channel, they will no longer receive notificaitons
 */

public void detach(Subscriber sub) {
    subs.remove(sub);
}

/*
 * Sends update to all of the subscribers
 *
 */
```

Here you can see I created a list of Subscribers, where all of our users that are subscribed are stored. I also created attach and detach methods, which are called when the user logs in, and logs out.



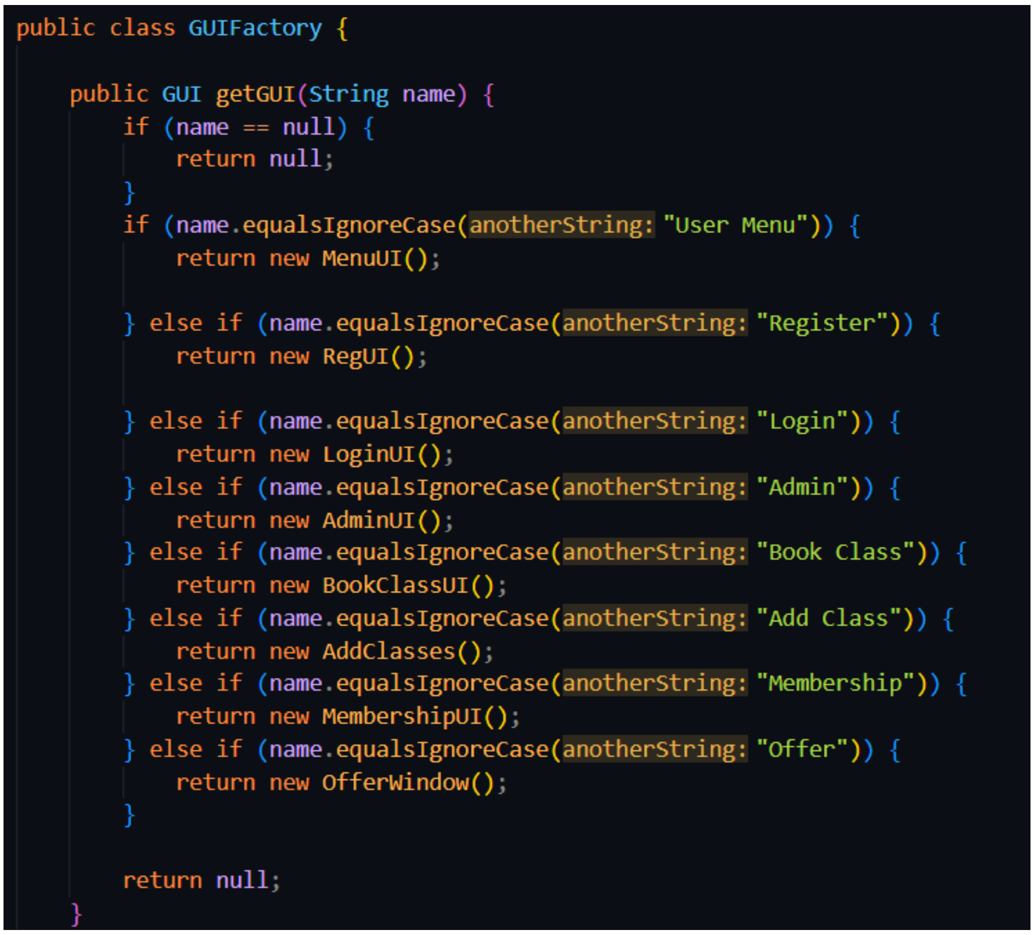
```java
public int userID;
public Channels channel = new Channels(); // What channels they are subscribed to

/*
 * Everytime a new activity is added, this method will be called
 * This will notify all subscribers that a new activity has been added!
 */

public void update() throws FileNotFoundException, IOException {
    GUIContext context = new GUIContext();
    State Offer = new StartOfferGUI();// uses factory
    context.setState(Offer);
    context.enterGUI();

}
```

When this ‘update’ method is called, it creates a new offer object, and also creates a new UI window for the offer. Currently we only have one offer, but this allows for more to be added in the future.

Factory Design Pattern:

Written by Emmet

The Factory Design is a creational design pattern that allows for the creation of objects. This is used to instantiate different interfaces, from the same place.



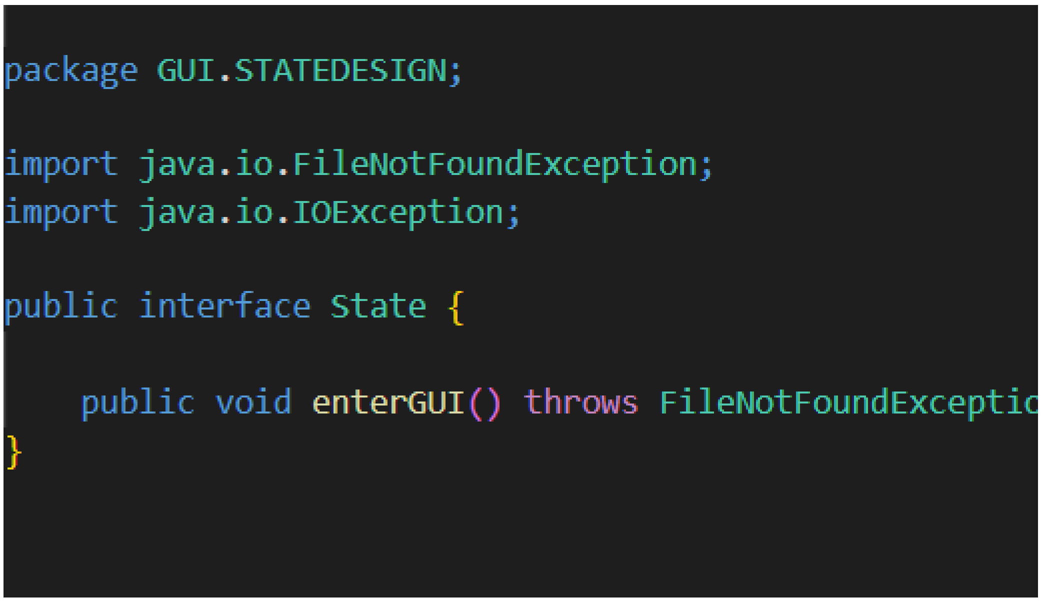
```java
public class GUIFactory {

    public GUI getGUI(String name) {
        if (name == null) {
            return null;
        }
        if (name.equalsIgnoreCase(anotherString: "User Menu")) {
            return new MenuUI();

        } else if (name.equalsIgnoreCase(anotherString: "Register")) {
            return new RegUI();

        } else if (name.equalsIgnoreCase(anotherString: "Login")) {
            return new LoginUI();
        } else if (name.equalsIgnoreCase(anotherString: "Admin")) {
            return new AdminUI();
        } else if (name.equalsIgnoreCase(anotherString: "Book Class")) {
            return new BookClassUI();
        } else if (name.equalsIgnoreCase(anotherString: "Add Class")) {
            return new AddClasses();
        } else if (name.equalsIgnoreCase(anotherString: "Membership")) {
            return new MembershipUI();
        } else if (name.equalsIgnoreCase(anotherString: "Offer")) {
            return new OfferWindow();
        }

        return null;
    }
```

Our system uses Factory methods to get references to GUI instances. The reason we do this is to handle potentially changing requirements in the future. If there are future changes and something other than these GUI’s need to be returned, we only need to make changes in the Factory class. This design pattern also supports the single responsibility principle as classes don’t need to be concerned with the instantiation of their objects, and can be focused on the task at hand.

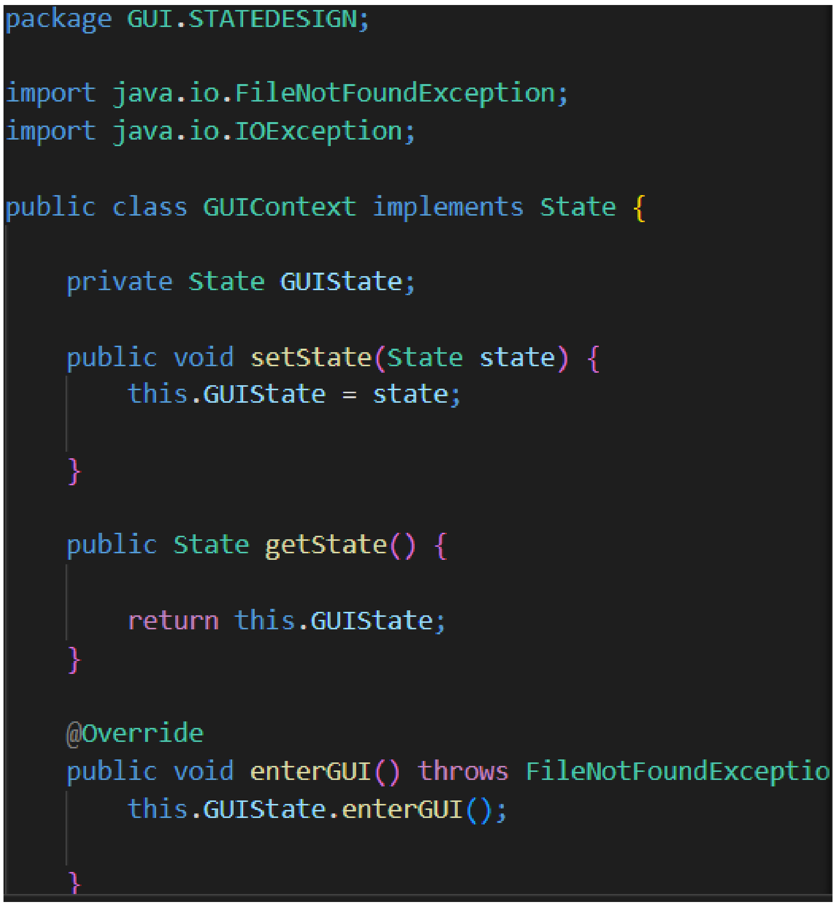
State Design Pattern:

Written by Oliver

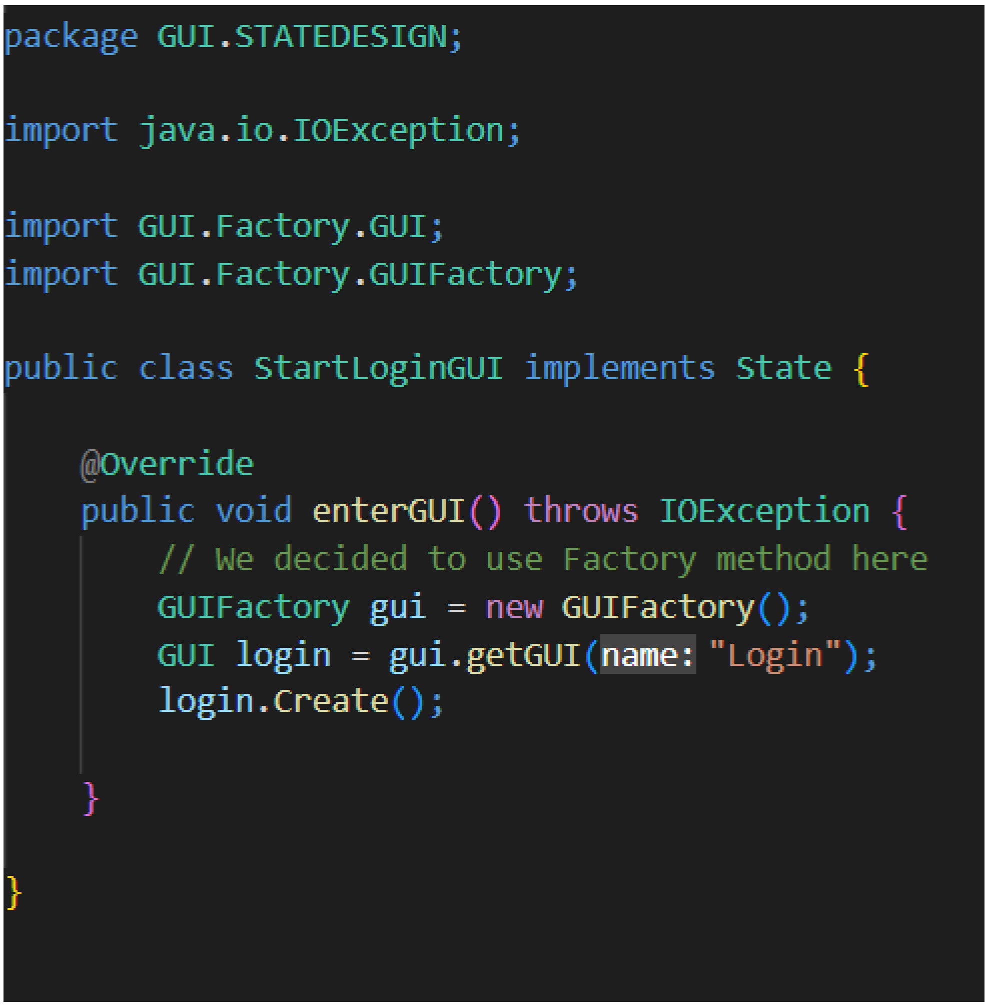
We implemented the State Design pattern in order to handle creation of our User interfaces. Our GUIContext base object contains a GUIState base object , indicating which GUI state is currently implemented. The State interface serves to define the basic characteristics of the different state objects. 8 classes in total implement the State object , one for each UI we have. Each of the 8 classes implement their own logic using the Factory method. Which will affect the GUIContext instance applied to the particular object. This allows the GUIContext to adjust its behaviour based on the particular state being applied.

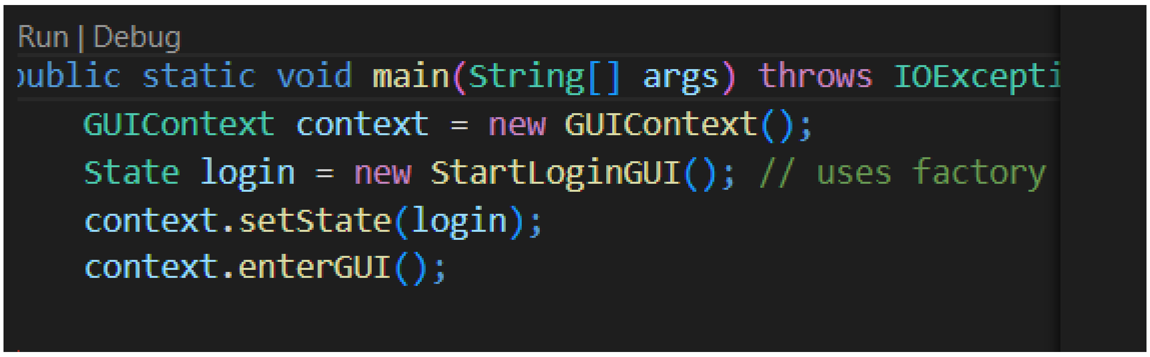


```
package GUI.STATEDESIGN;

import java.io.FileNotFoundException;
import java.io.IOException;

public interface State {

    public void enterGUI() throws FileNotFoundExceptio
}
```



```java
package GUI.STATEDESIGN;

import java.io.FileNotFoundException;
import java.io.IOException;

public class GUIContext implements State {

    private State GUIState;

    public void setState(State state) {
        this.GUIState = state;

    }

    public State getState() {

        return this.GUIState;
    }

    @Override
    public void enterGUI() throws FileNotFoundExceptio
        this.GUIState.enterGUI();

    }
```

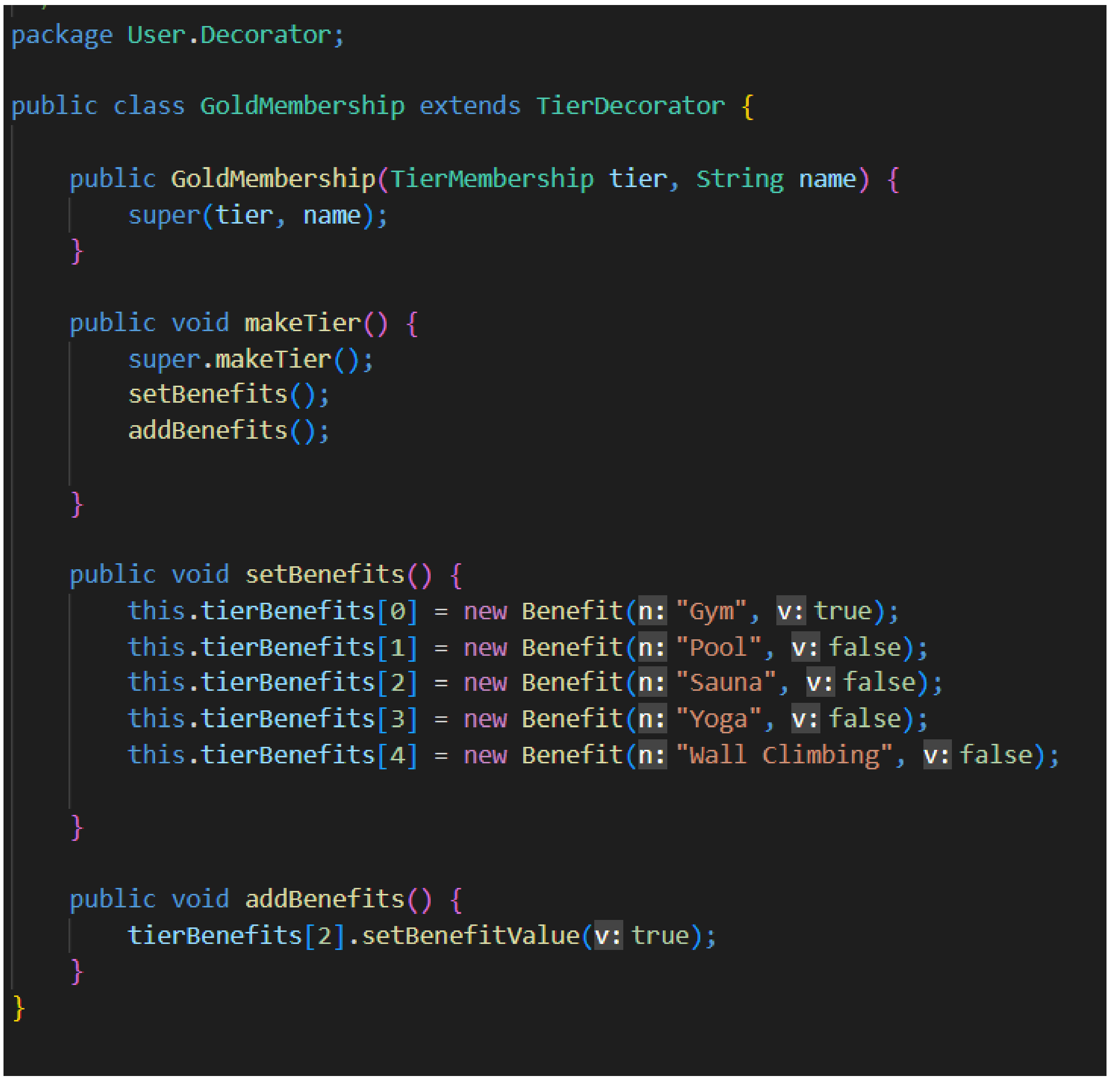

```java
package GUI.STATEDESIGN;

import java.io.IOException;

import GUI.Factory.GUI;
import GUI.Factory.GUIFactory;

public class StartLoginGUI implements State {

    @Override
    public void enterGUI() throws IOException {
        // We decided to use Factory method here
        GUIFactory gui = new GUIFactory();
        GUI login = gui.getGUI(name: "Login");
        login.Create();

    }

}
```


```java
Run | Debug
ublic static void main(String[] args) throws IOExcepti
    GUIContext context = new GUIContext();
    State login = new StartLoginGUI(); // uses factory
    context.setState(login);
    context.enterGUI();
```

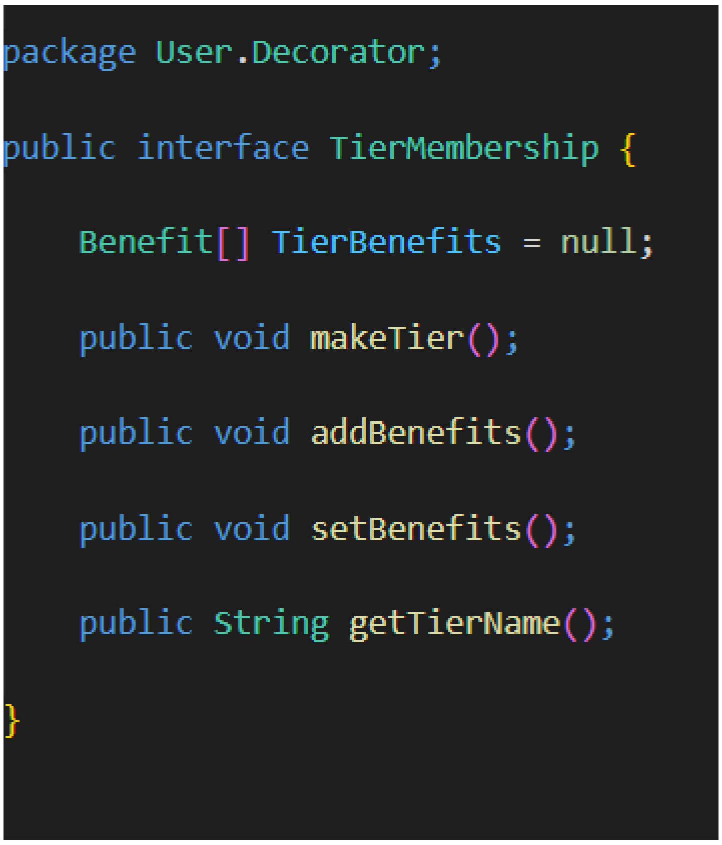
Decorator:

Written by Oliver

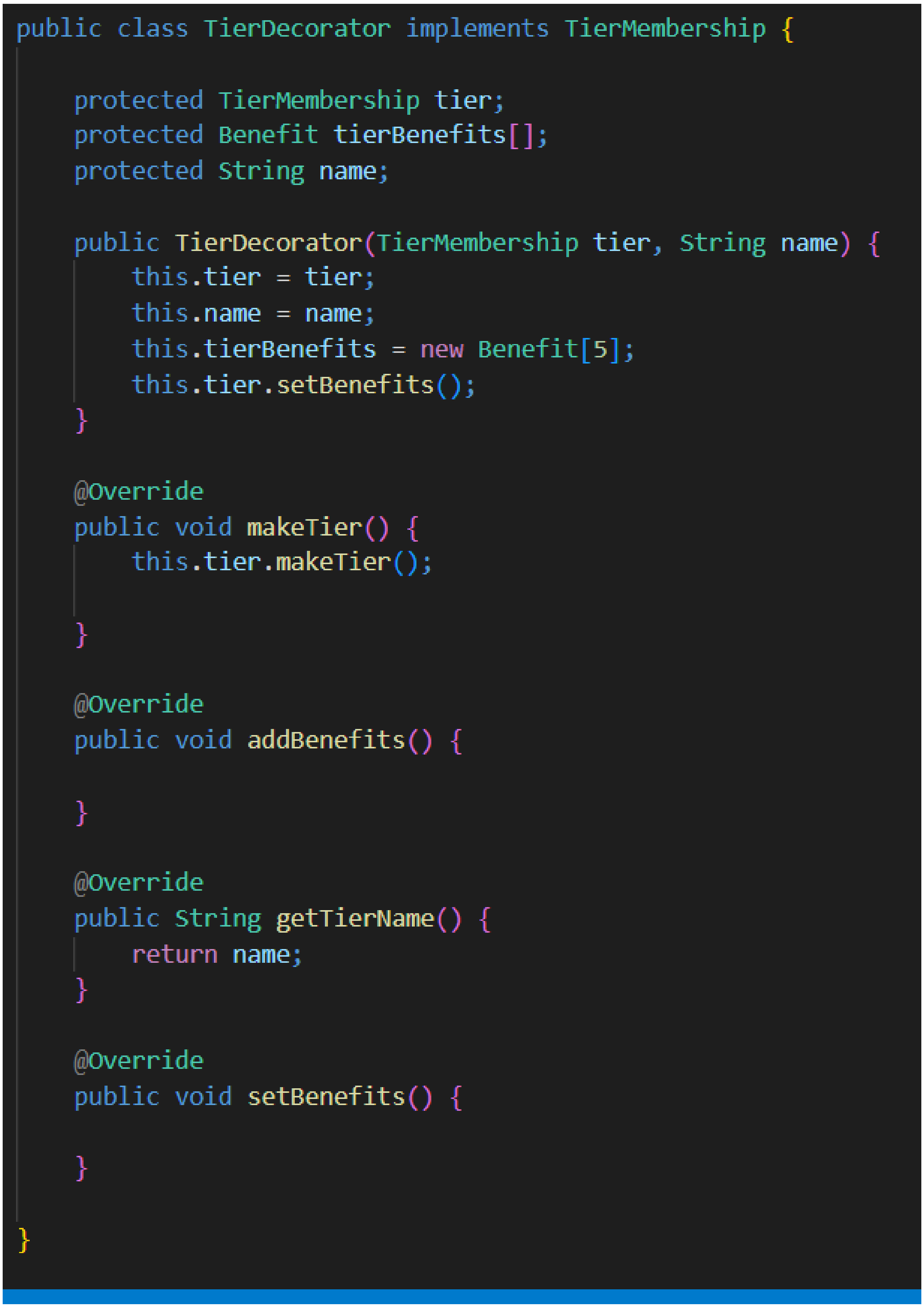
In order to support additional benefits for certain types of Membership tiers we have taken advantage of the decorator pattern. This structural pattern takes an existing object at run time and binds additional functionality to that object. The decorator pattern can be used when a subclassing approach results in a large number of subclasses. In this case it will allow us to wrap our component in whatever combination of decorator classes that we require rather than creating an entire new subclass for that specific combination of functionality. In our case this means that each of our subclasses e.g Membership Tiers will have varying functionality with their setBenefits() methods. To start with we have an interface TierMembership. Then we have our TierDecorator which implements our interface. And finally we have our concrete decorators SilverMemberShip and GoldMemberShip which extend Tier Decorator.



```java
package User.Decorator;

public class GoldMembership extends TierDecorator {

    public GoldMembership(TierMembership tier, String name) {
        super(tier, name);
    }

    public void makeTier() {
        super.makeTier();
        setBenefits();
        addBenefits();

    }

    public void setBenefits() {
        this.tierBenefits[0] = new Benefit(n: "Gym", v: true);
        this.tierBenefits[1] = new Benefit(n: "Pool", v: false);
        this.tierBenefits[2] = new Benefit(n: "Sauna", v: false);
        this.tierBenefits[3] = new Benefit(n: "Yoga", v: false);
        this.tierBenefits[4] = new Benefit(n: "Wall Climbing", v: false);

    }

    public void addBenefits() {
        tierBenefits[2].setBenefitValue(v: true);
    }
}
```



```
package User.Decorator;

public interface TierMembership {

    Benefit[] TierBenefits = null;

    public void makeTier();

    public void addBenefits();

    public void setBenefits();

    public String getTierName();

}
```


```java
public class TierDecorator implements TierMembership {

    protected TierMembership tier;
    protected Benefit tierBenefits[];
    protected String name;

    public TierDecorator(TierMembership tier, String name) {
        this.tier = tier;
        this.name = name;
        this.tierBenefits = new Benefit[5];
        this.tier.setBenefits();
    }

    @Override
    public void makeTier() {
        this.tier.makeTier();

    }

    @Override
    public void addBenefits() {

    }

    @Override
    public String getTierName() {
        return name;
    }

    @Override
    public void setBenefits() {

    }

}
```

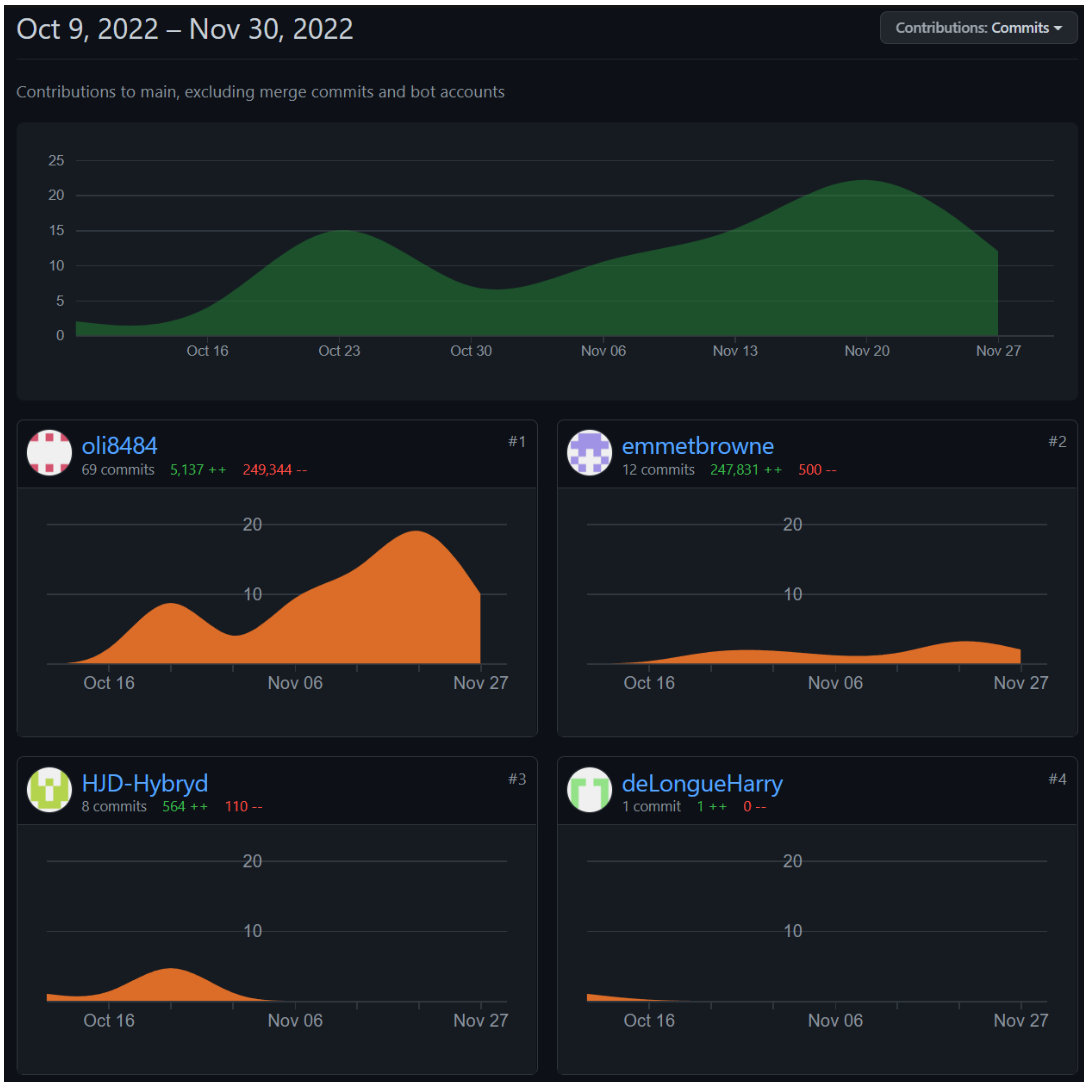
8.5 Visualisations of Version Control

Written by Emmet

https://github.com/deLongueHarry/CS4125Proj22
We used Github as our form of version control as all members of the team were familiar with it. From the beginning, we decided the general architecture of our project by making skeleton / pseudocode classes. From here, we decided that for each change made to the code, we would create a new branch. We named each branch as follows: 'Example_Implementation_*TeamMemberName*'. This allowed for us to easily see what changes had been made to the code, and who had implemented them.

To keep all code updated, we would only create branches from the most recently updated branch. If I was to do this project again, I would merge each branch into main so that we would have a designated 'final' version of our code with each update. However, the system we used allowed for efficient development, and ensured that each team member could keep their code up to date.

We also took part in a lot of paired programming, especially when someone was finished working on a branch. We would get on a call and run through the changes that were made to ensure everyone was happy.


Oct 9, 2022 – Nov 30, 2022

Contributions to main, excluding merge commits and bot accounts

oli8484 #1
69 commits 5,137 ++ 249,344 --

emmetbrowne #2
12 commits 247,831 ++ 500 --

HJD-Hybryd #3
8 commits 564 ++ 110 --

deLongueHarry #4
1 commit 1 ++ 0 --

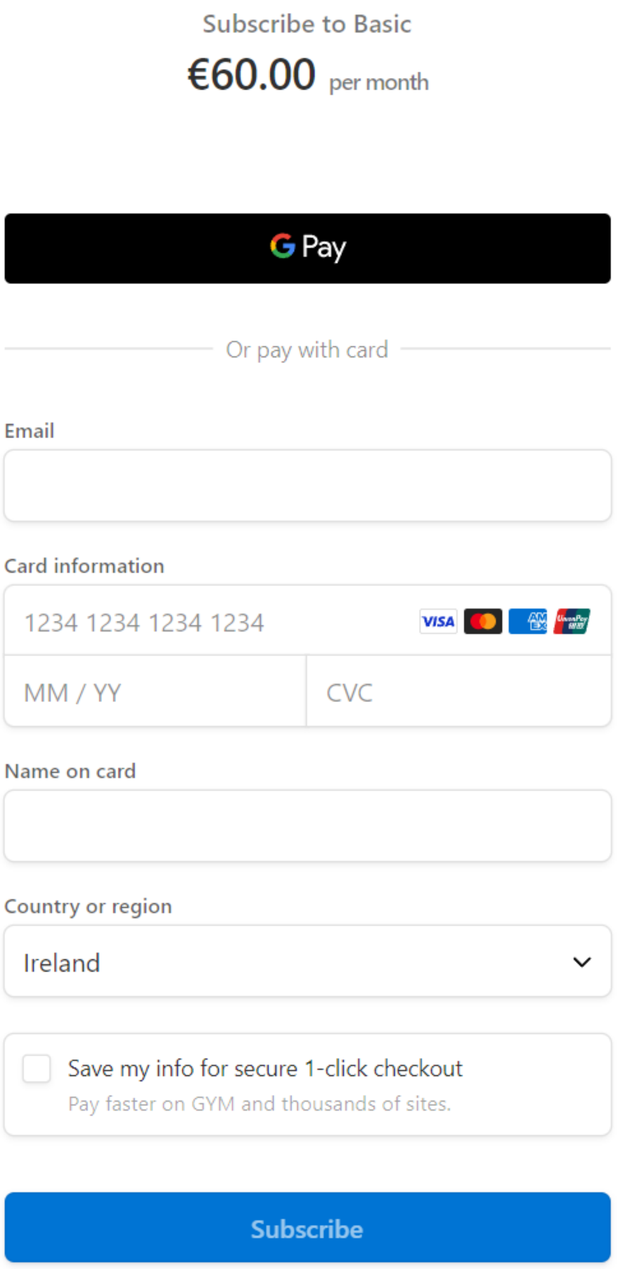
8.6 Added Value

Written by Emmet

Upon completion of the main code for this project, we wanted to add some extra functionality as added value.

Stripe

One of the features we added was a Stripe payment gate. If we were fully developing a Gym Booking System with full functionality, a payment gate would be necessary so we decided to add this as proof of concept. It was also something that Oliver and I had never done before so we learned a lot from it.


Subscribe to Basic

€60.00 per month

Email

Card information

1234 1234 1234 1234

MM / YY

CVC

Name on card

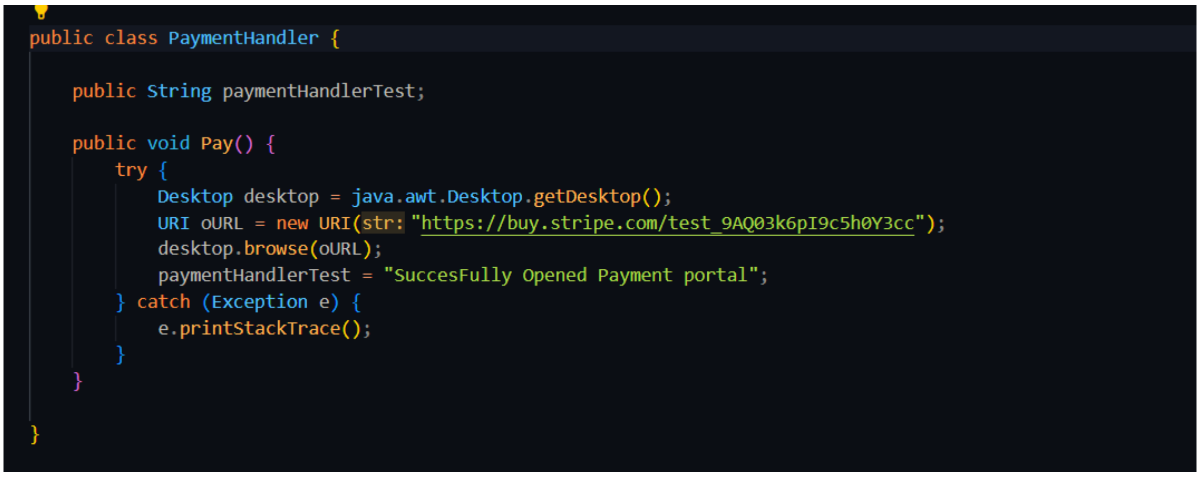
Country or region

Ireland

Save my info for secure 1-click checkout

Pay faster on GYM and thousands of sites.

Implementing Stripe
Firstly, I had to create an account with Stripe to have access to their API dashboard. From here, I was able to create test cards, and create a simple paygate. This website was opened when you pressed one of the ‘Buy Membership’ buttons.



```java
public class PaymentHandler {

    public String paymentHandlerTest;

    public void Pay() {
        try {
            Desktop desktop = java.awt.Desktop.getDesktop();
            URI oURL = new URI(str: "https://buy.stripe.com/test_9AQ03k6pI9c5h0Y3cc");
            desktop.browse(oURL);
            paymentHandlerTest = "SuccesFully Opened Payment portal";
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

}
```

EsLint

We also chose to add EsLint to our project as added value. I had previously worked with Lint during my Co-operative placement so I was happy to implement it. Once I had installed it to our project, I had to feed it some information about our code to choose the best rules.

We found that once Lint was ran, the main issues were as follows:

- Unused imports
- Empty catch statement (Try - Catch)
- Incorrect naming conventions for classes
- Unnecessary comment lines
- Incorrect static instantiation

Once we had got rid of all of these issues, the code felt more complete and polished. If I were to do this again, I would possibly have implemented EsLint earlier in the development, so that we could run it at different stages.

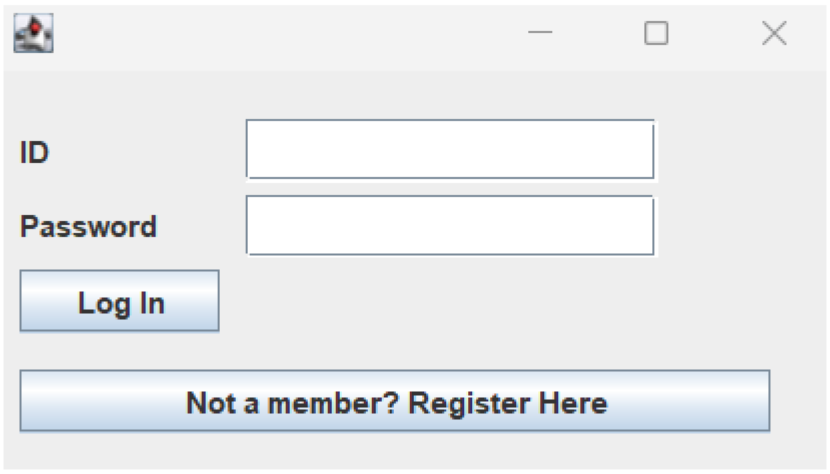
Key Use Cases:
<u>Written by Oliver</u>

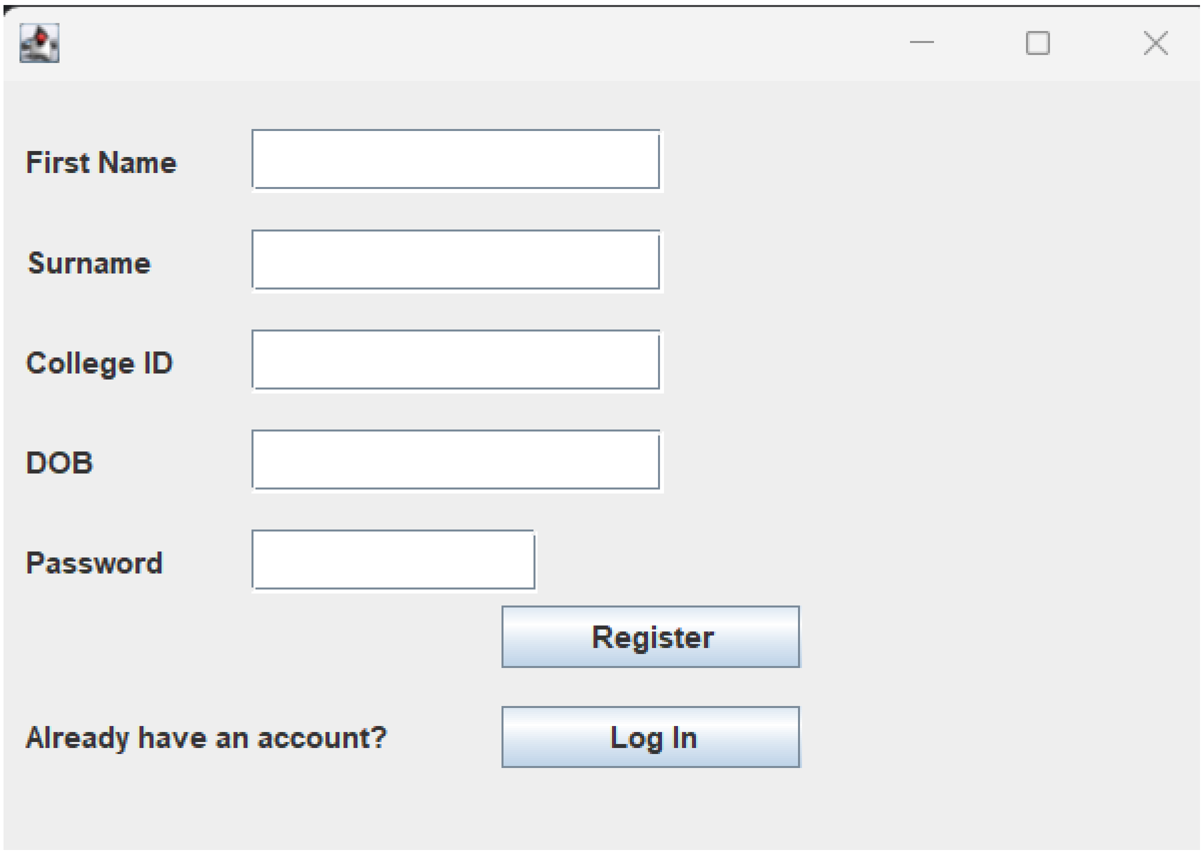
1. Registering and Logging In

   One of our use cases was registering and logging in for a member and admin.

```
public void RegisterUser() {
        DateTimeFormatter formatter =
DateTimeFormatter.ofPattern("d/MM/yyyy");

        firstName = fNameText.getText();
        surName = sNameText.getText();
        DOB = DOBText.getText();
        ID = collegeIDText.getText();
        password = passwordText.getText();
        LocalDate dob = LocalDate.parse(DOB, formatter);
        LocalDate Today = LocalDate.now();

        User member = new User(ID, firstName + " " + surName,
Today, Today, dob, "Admin");
        Subscriber sub = new Subscriber(ID, firstName + " " +
surName, Today, Today, dob, "Member");

        LocalDate dateOfB = member.getDOB();
        LocalDate start = member.getstartDate();
        LocalDate end = member.getendDate();

        String[][] Subscriber = {
                { sub.getUserID(), sub.getUserName(),
dateOfB.toString(), start.toString(), end.toString(),
                        sub.getUserType() }

        };

        String[][] User = {
                { member.getUserID(), member.getUserName(),
dateOfB.toString(), start.toString(), end.toString(),
                        member.getUserType() }
        };

        String[][] userAndPassword = {
                { member.getUserID(), password }
        };
        WriterDTO wr3 = new WriterDTO();
        try {
            wr3.Write(Subscriber, "SUBSCRIBERS");
        } catch (IOException e2) {
            e2.printStackTrace();
        }

        WriterDTO wr2 = new WriterDTO();

        try {
            wr2.Write(userAndPassword, "LOGIN_DETAILS");
        } catch (IOException e1) {

            e1.printStackTrace();
        }

        WriterDTO wr = new WriterDTO();

        try {

            wr.Write(User, "USERS");
        } catch (IOException e1) {

            e1.printStackTrace();
        }

    }

}
```

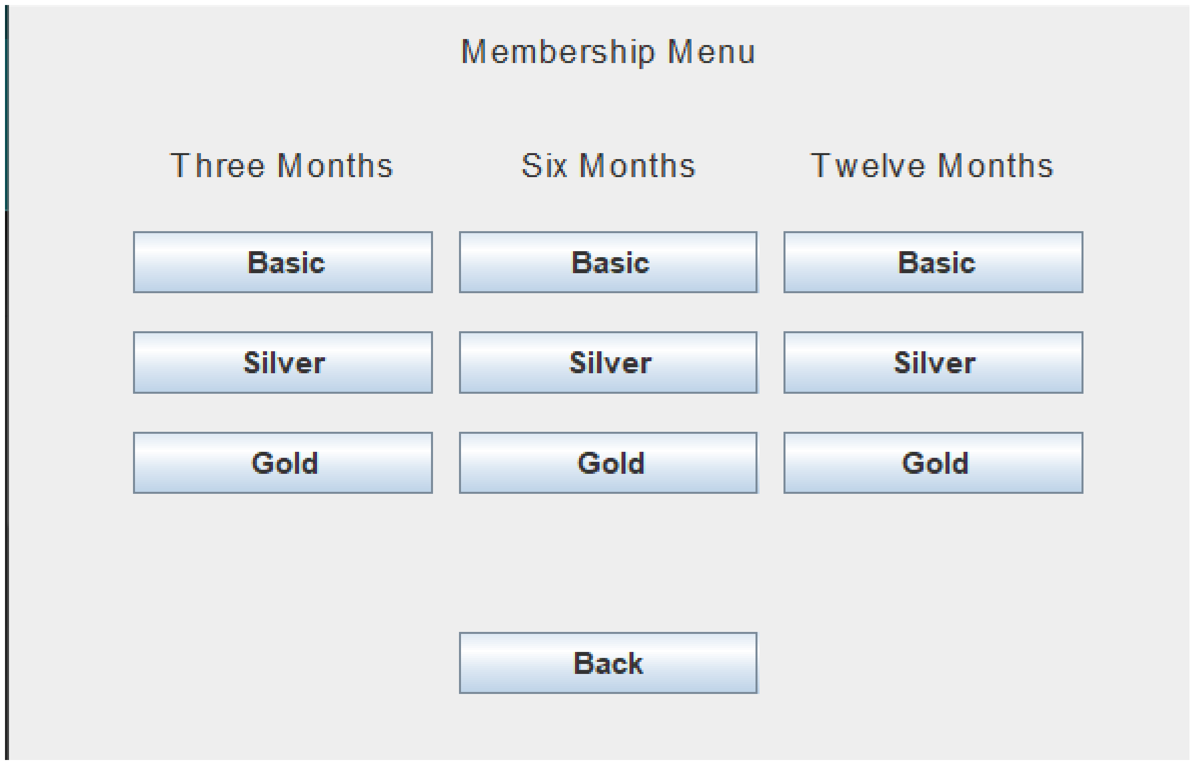
```
public void LoginUser() {

        userName = userText.getText();
        password = passwordText.getText();

        VerifyLoginDTO rd = new VerifyLoginDTO();
        try {
            rd.Verify("LOGIN_DETAILS", password, userName);
            if (rd.found == true) {

                MenuUI.SetID(userName);
                AdminUI.SetID(userName);
                if (userName.equals("Admin")) {

                    startAdmin = new StartAdminGUI();// uses
factory

                    context.setState(startAdmin);
                    context.enterGUI();

                } else {

                    startUser = new StartMemberGUI();// uses
factory

                    context.setState(startUser);
                    context.enterGUI();
                }

                frame.setVisible(false); // you can't see me!
                panel.setVisible(false);
                frame.dispose();

                testLoginMessage = "Succesfull Login";

            } else {
                testLoginMessage = "Unsuccessfull Login";

            }
        } catch (IOException e1) {

            e1.printStackTrace();
        }
    }


}
```

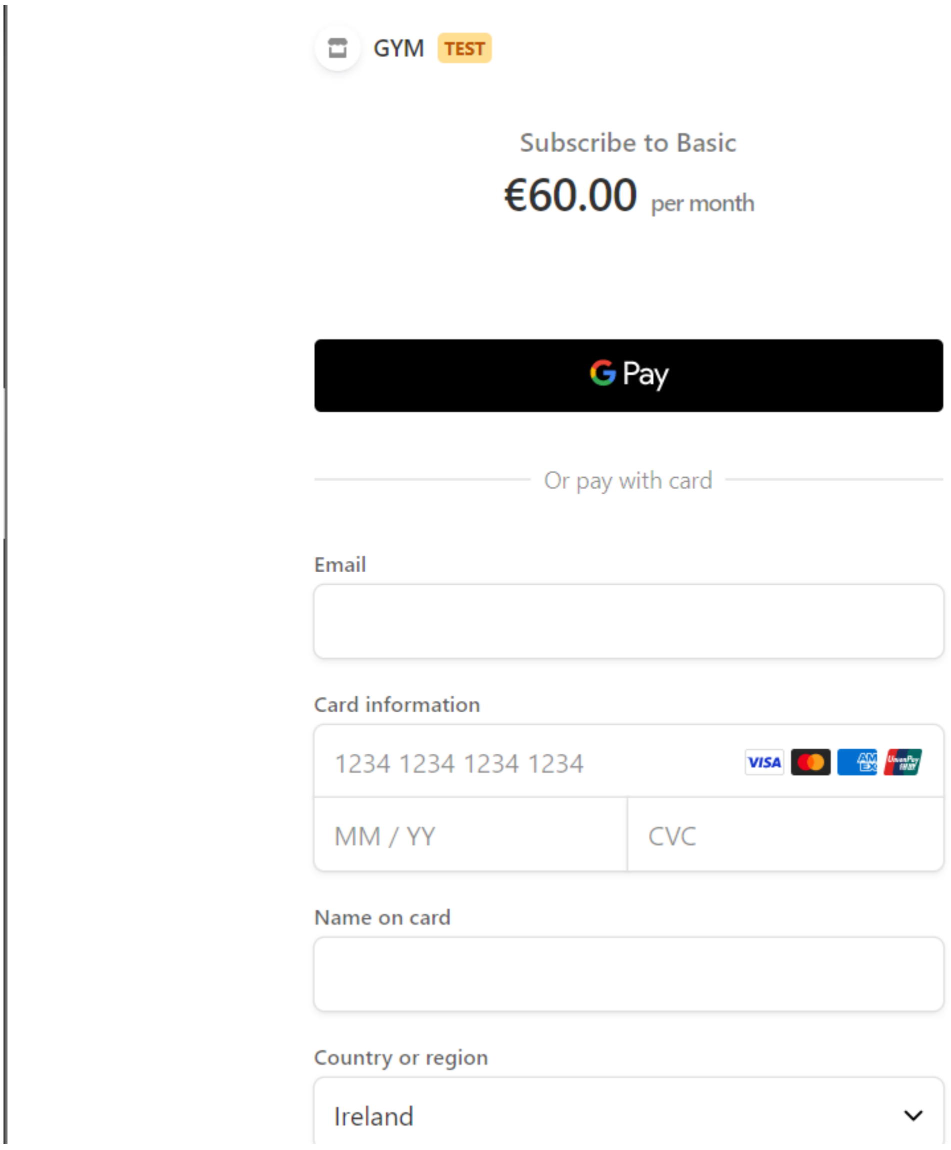

ID
Password
Log In
Not a member? Register Here


First Name
Surname
College ID
DOB
Password
Register
Already have an account?
Log In

2. Buying Membership

A member could purchase a membership online using an online gateway managed by Stripe.




GYM TEST

Subscribe to Basic

€60.00 per month

G Pay

Or pay with card

Email

Card information

1234 1234 1234 1234

MM / YY

CVC

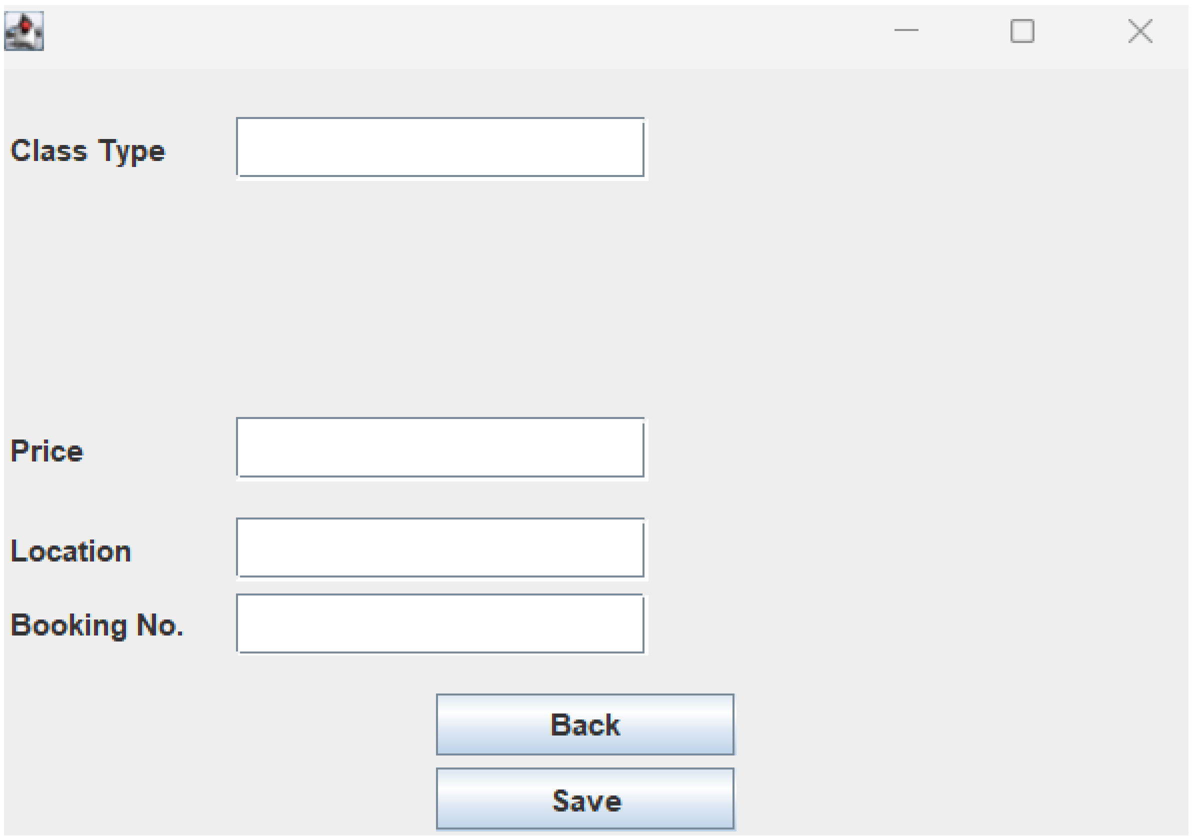
Name on card

Country or region

Ireland

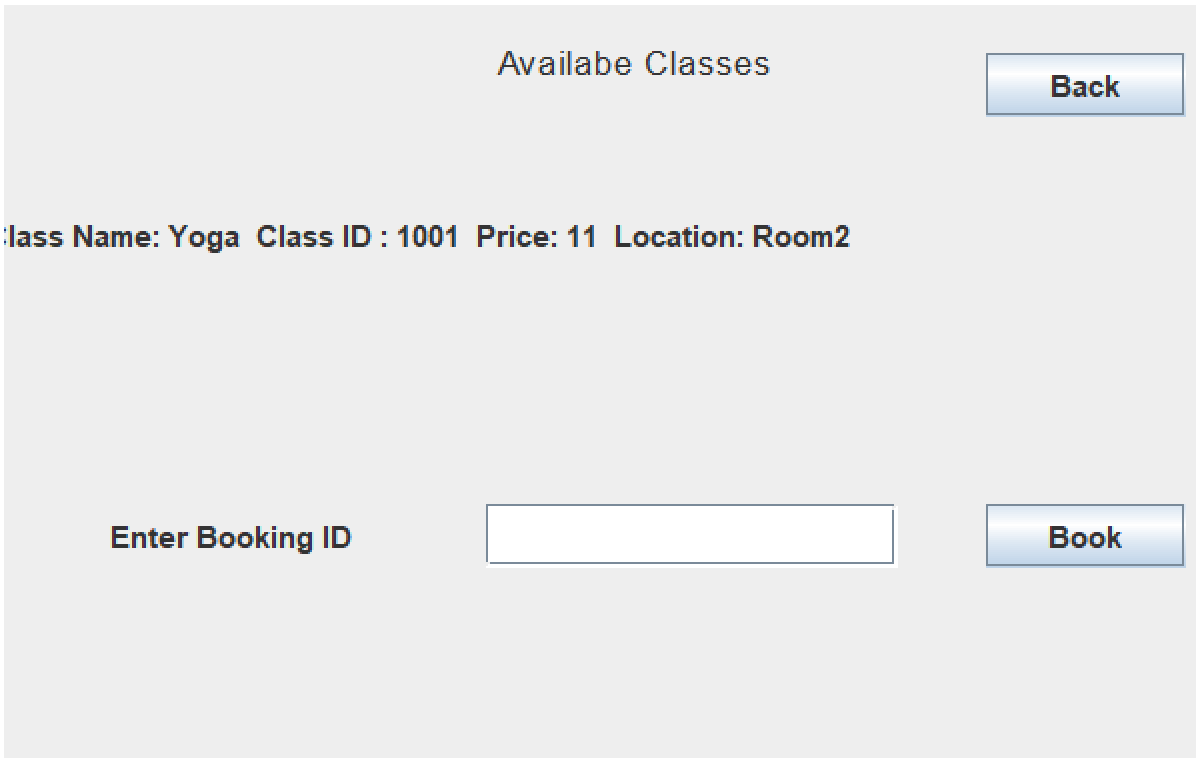
3. Creating Classes

Another use case we had was Creating Classes . This can only be done if you are signed in as an Admin.



4. Booking Classes

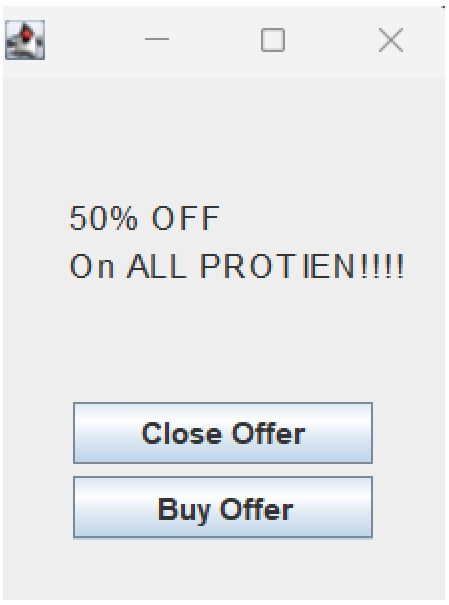
Another key use case is booking classes. This can only be done if you have a valid Member account.



lass Name: Yoga Class ID : 1001 Price: 11 Location: Room2

5. Buying Offers

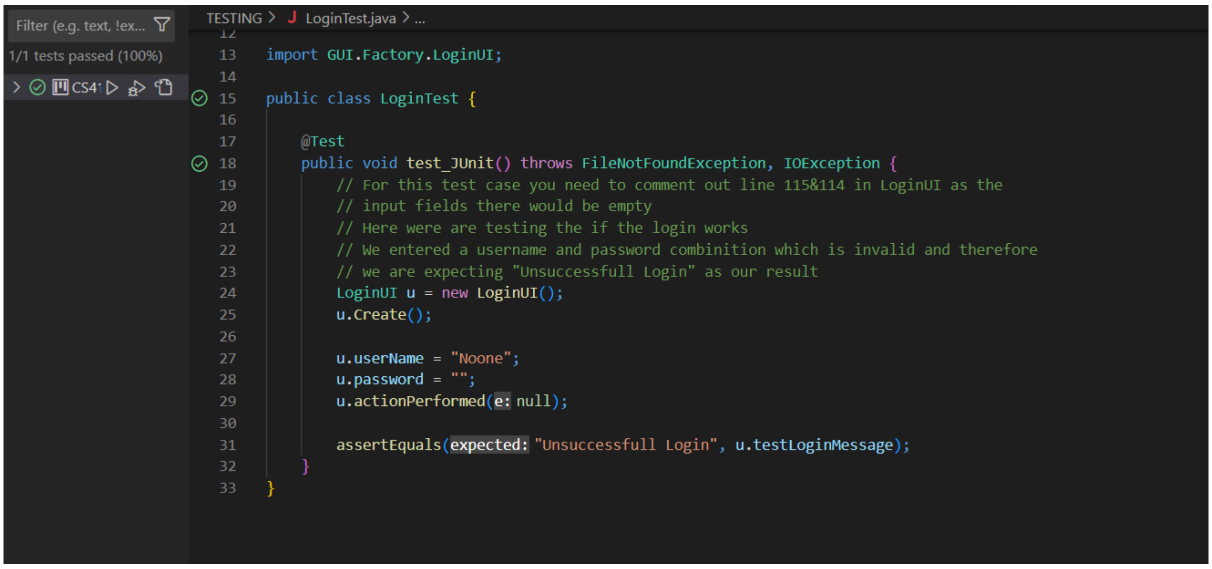
Our last key use case was buying offers . These special offers would be displayed at each login to all subscribed users.



Automated Testing Junit:
<u>Written by Oliver</u>

Test 1:

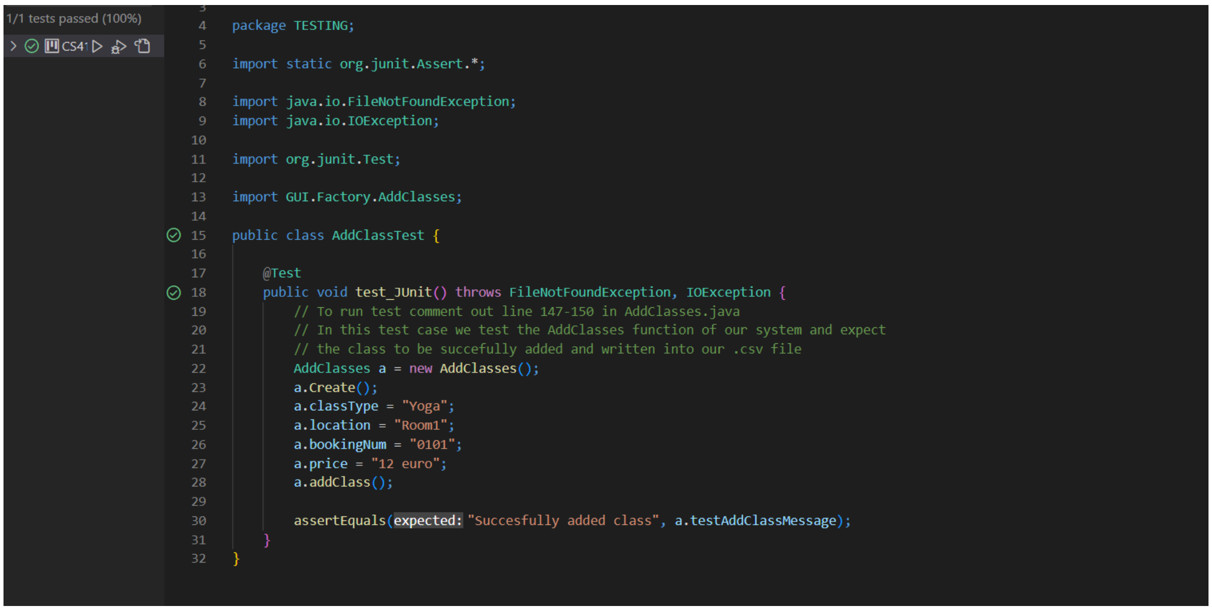
For our testing we decided to use Junit. For the first test case we tested our Login to see if it was functional. For this test we edited our code so that on successful logins the testLoginMessage would read “Successful” and on unsuccessful logins “Unsuccessful” Then we entered a username and password combination which was invalid and not stored in our system/database. Our test was successful.



```java
import GUI.Factory.LoginUI;

public class LoginTest {

    @Test
    public void test_JUnit() throws FileNotFoundException, IOException {
        // For this test case you need to comment out line 115&114 in LoginUI as the
        // input fields there would be empty
        // Here were are testing the if the login works
        // We entered a username and password combinition which is invalid and therefore
        // we are expecting "Unsuccessfull Login" as our result
        LoginUI u = new LoginUI();
        u.Create();

        u.userName = "Noone";
        u.password = "";
        u.actionPerformed(e: null);

        assertEquals(expected: "Unsuccessfull Login", u.testLoginMessage);
    }
}
```

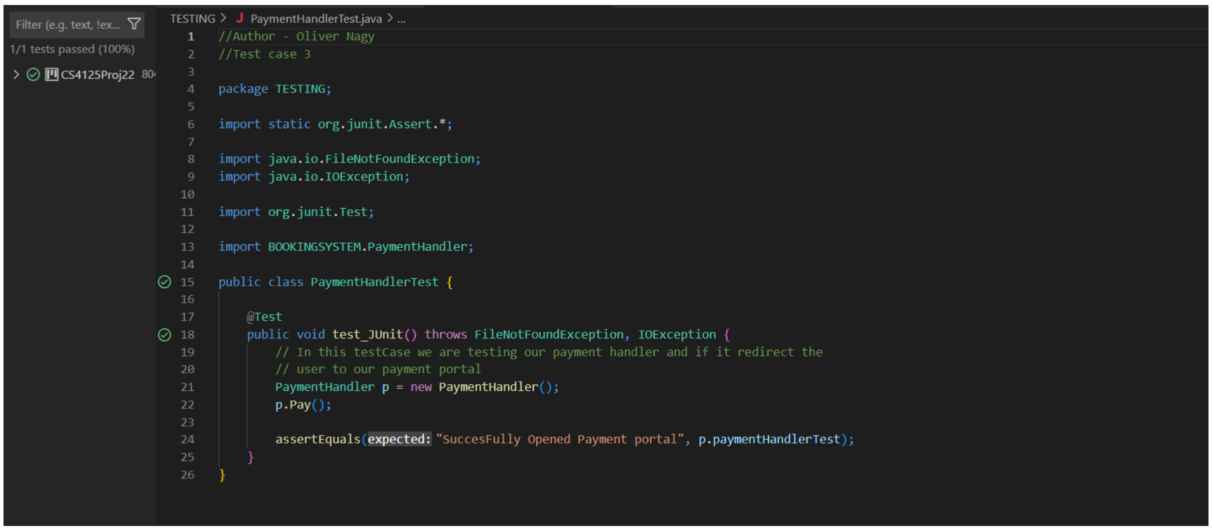
Test 2:

For our second test we tested our Adding Classes Functionality. In this test we also created a String which will store our test message in this case if our class has been created/added successfully. This test was also successful.



```java
package TESTING;

import static org.junit.Assert.*;

import java.io.FileNotFoundException;
import java.io.IOException;

import org.junit.Test;

import GUI.Factory.AddClasses;

public class AddClassTest {

    @Test
    public void test_JUnit() throws FileNotFoundException, IOException {
        // To run test comment out line 147-150 in AddClasses.java
        // In this test case we test the AddClasses function of our system and expect
        // the class to be succefully added and written into our .csv file
        AddClasses a = new AddClasses();
        a.Create();
        a.classType = "Yoga";
        a.location = "Room1";
        a.bookingNum = "0101";
        a.price = "12 euro";
        a.addClass();

        assertEquals(expected: "Succesfully added class", a.testAddClassMessage);
    }
}
```

Test 3:
Our last test case was testing the Payment Handler redirect link to our portal. In this test we were checking if it successfully opened the payment portal.The test was successful.



Filter (e.g. text, !ex...
1/1 tests passed (100%)
CS4125Proj22

TESTING > PaymentHandlerTest.java > ...

```java
//Author - Oliver Nagy
//Test case 3

package TESTING;

import static org.junit.Assert.*;

import java.io.FileNotFoundException;
import java.io.IOException;

import org.junit.Test;

import BOOKINGSYSTEM.PaymentHandler;

public class PaymentHandlerTest {

    @Test
    public void test_JUnit() throws FileNotFoundException, IOException {
        // In this testCase we are testing our payment handler and if it redirect the
        // user to our payment portal
        PaymentHandler p = new PaymentHandler();
        p.Pay();

        assertEquals(expected: "SuccesFully Opened Payment portal", p.paymentHandlerTest);
    }
}
```

10 Architecture and design blueprints

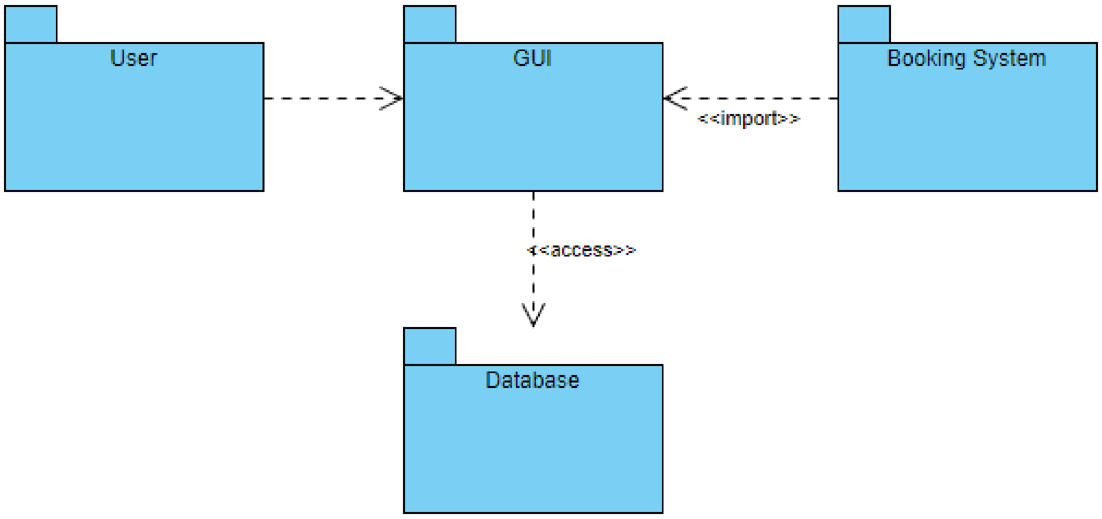


Figure 10.1 Design-Time package Diagram

Class Diagram

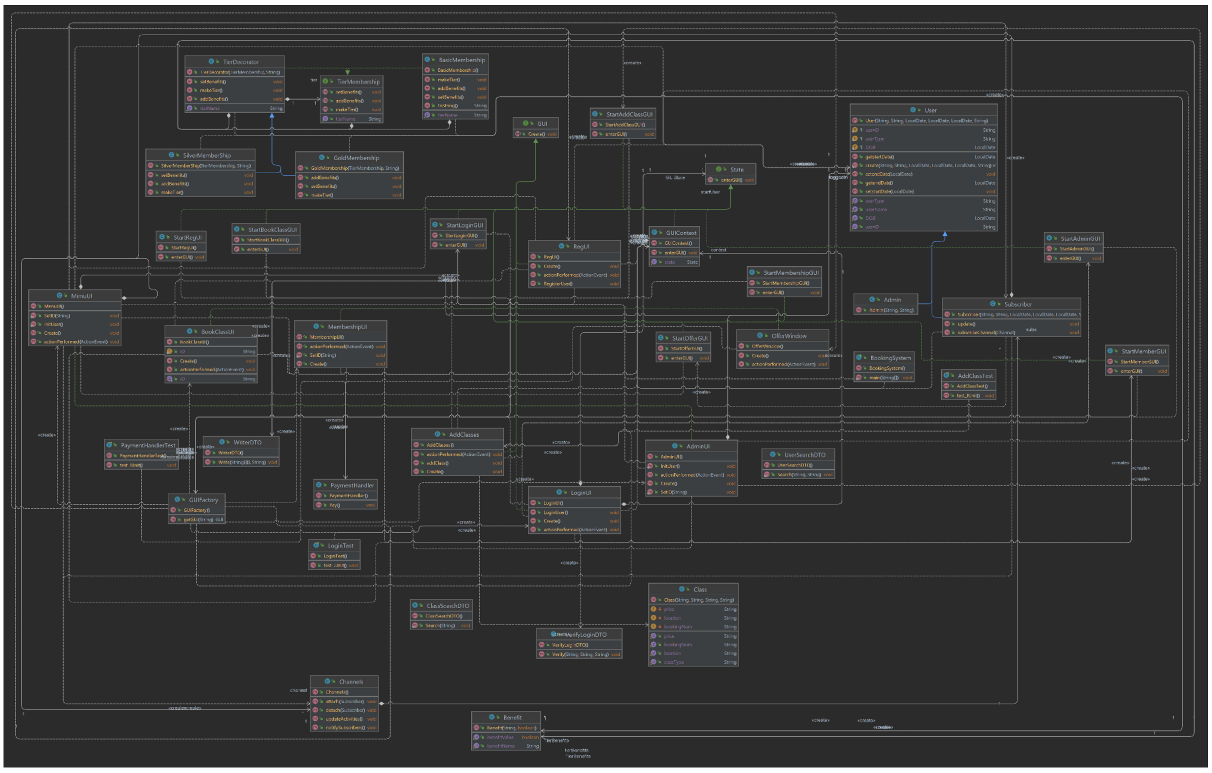


Figure 11.1 Design-Time Class Diagram

Sequence Diagram

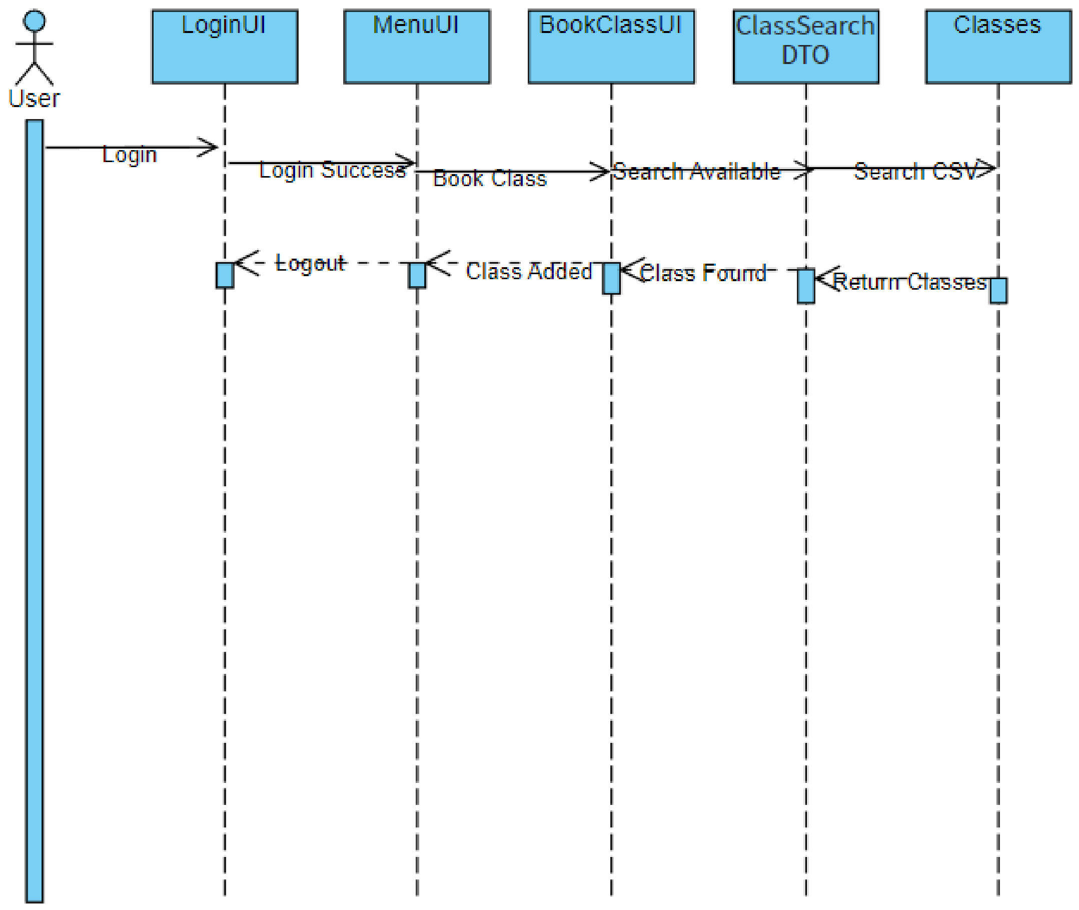


Figure 10.1 Design-Time SequenceDiagram

11 Critique

Throughout the duration of the project many challenges were faced. The main critique throughout the duration of the project was the time constraint. From week 4 to week 12, the analysis requirements along with the implementation proved very time consuming.

Another challenge faced in the project was the implementation of sufficient design patterns to adhere to the specification laid out in week 4. It proved a challenge finding areas of the implementation where adding design patterns would be useful for the code to satisfy requirements and non-functional requirements.

The design of diagrams to aid in the implementation effort proved useful as it laid out a structure for the team to follow.
All team members contributed equally at the beginning of the project's life cycle but ultimately did not contribute equally to the final submission.

The Diagrams that we initially drafted proved to be very different from our finished product. I think in retrospect that we should have spent more time at the design state really nailing down which classes we would need, the structure we would implement and how classes would interact with each other. Instead of which we ended up changing the application structure throughout development. I think if we were to redo the design stage with the knowledge we have now, we could reduce the difference in design and implementation significantly.