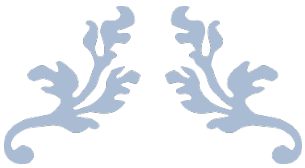


CS 5004 Final Project: Mindful Mastery

Xinyan Liu

APRIL 25, 2023

Java Programming Language Concept Map

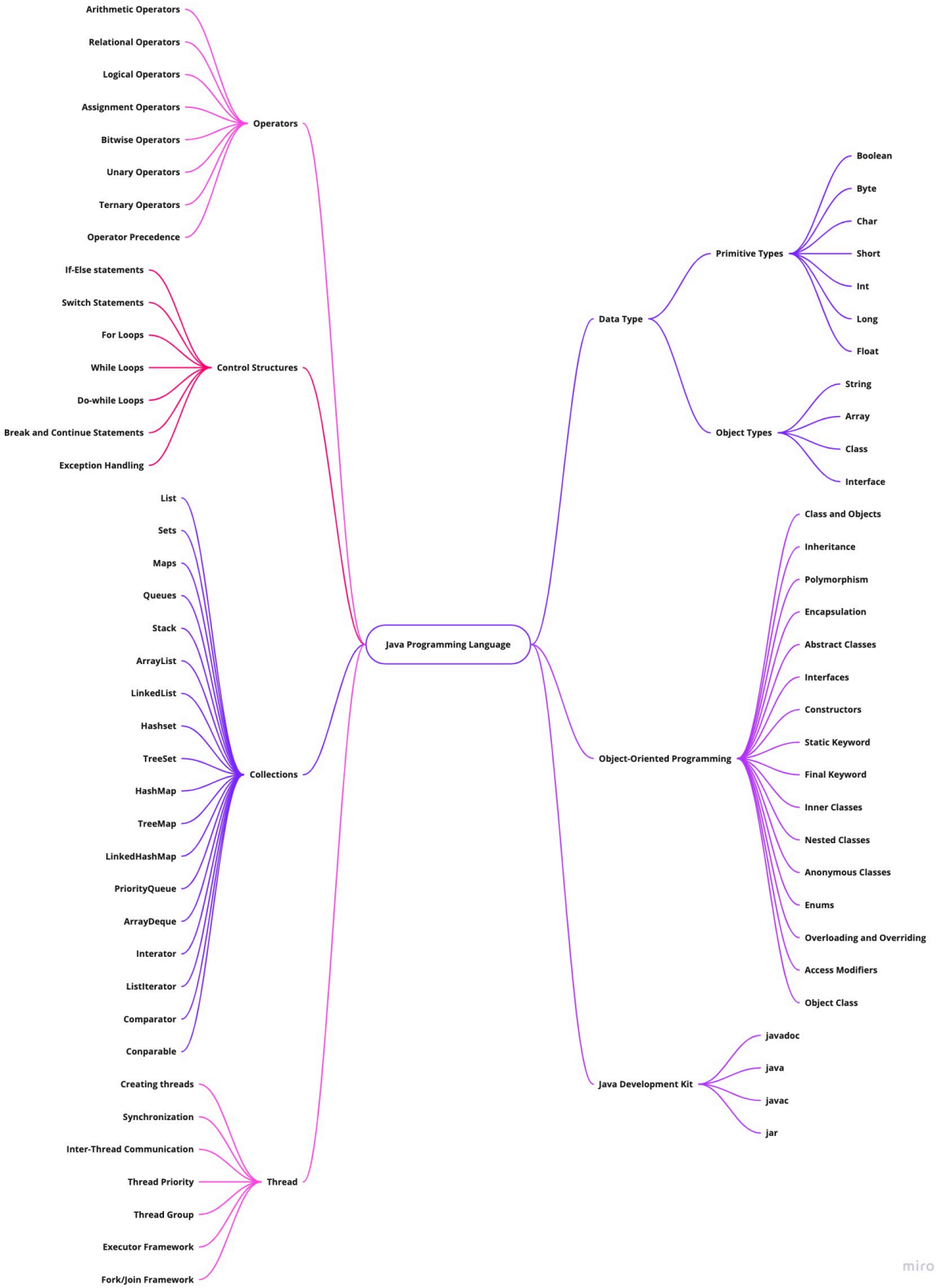


Table of Contents

Topic Introduction

Meditation has been used for thousands of years to help people calm down, lessen their stress levels, and gain focus. Meditation's popularity has skyrocketed in recent years as more individuals look for ways to boost their health and happiness. Meditation apps have become increasingly popular because they make it simple to access guided meditations, mindfulness exercises, and other resources that can help with meditation.

As part of this Java programming project, we will create a basic application for meditation. Users can sign up for an account, log in, and keep tracks on their meditation progress with the app. Guided meditations will be available for users to select from, along with the ability to record their emotional status after each meditation session and track their progress. This program will also have tools for handling subscriptions and informing users of courses updates. We will learn about object-oriented programming, Java data structures, and user interface design while working on this project.

Assignment Goals

The goal of this project is to create a basic meditation application that allows users to keep track of their meditation activities and receive notifications about new courses and sessions. This application is built using the Model-View-Controller (MVC) design pattern and implementing the Observer pattern. This program includes several classes and interfaces to represent different entities such as Users, Courses, Sessions, and Records. This application also features a notification system using observer design pattern for notifying users for updated courses. The overall goal of this work is to provide a comprehensive solution for users to manage their meditation activities in an organized way.

Table: concepts covered by this project and where to find it

Concept	Where to find it	How it was demonstrated
Recursion in Practice (Intermediate level)	"module" class's hierarchy structure	Recursion is used to traverse this tree structure in a depth-first manner, where each module is treated as the root node of a sub-tree. Also used in the ExerciseRecordList class to implement the size() and filter() methods.
Abstract Classes and Interfaces (Intermediate level)	"ExerciseInterface" interface; "Meditation" class extends the "Exercise" class and implement the "ExerciseInterface"; abstract class "ExerciseRecord" abstract class; "ExerciseRecordList" "Observer" interface; "User" class implements the "Observer" interface;	Please see the UML diagrams in chapter 2

Abstracted Linked Lists (Intermediate level)	ExerciseRecordList class and its subclasses	In the ExerciseRecordList class, the linked list is defined with a generic type T, which is constrained to be a subtype of ExerciseRecord. This allows for the creation of a linked list that can store any type of ExerciseRecord.
Higher order functions map, filter, and fold (Intermediate level)	The “ExerciseRecordList” class implements the filter method; “Module” class contains “getTotalCredits()” method uses the “mapToInt()” and “sum()” functions; “getCourseNames()” method uses the “map()” function; “getRequiredCourses(Predicate< Course> predicate)” method uses the “filter()” function;	Higher-order function that takes a predicate and returns a filtered list of ExerciseRecord objects; "getTotalCredits()" method uses the "mapToInt()" and "sum()" functions to calculate the total number of credits for all courses in the module; "map()" function maps the list of Course objects to a list of their names;
Hierarchical Data Representation (Intermediate level)	“Module” class; “Meditation” class; “Session” class; “NotificationSystem” class;	"Module" class represents a hierarchy of a meditation program; "Meditation" class demonstrates hierarchical data representation by containing a list of "MeditationRecord" objects; "Session" class demonstrates hierarchical data representation by containing a list of "SessionRecord" objects; "NotificationSystem" class demonstrates hierarchical data representation by containing a list of "Observer" objects.
MVC Design (Intermediate level)	“User” model; “UserController”; “UserView”;	Create an instance of the User model class, an instance of the UserView class that displays the model, and an instance of the UserController class that connects the two. Use JFrame and add the UserView object to its content pane, and make the frame visible.
SOLID Principles (Intermediate level)	Please see descriptions in Chapter 7	Please see descriptions in Chapter 7
Observer Pattern (Intermediate level)	“Observer” interface; “NotificationSystem” class;	Use an Observer design pattern to notify users about new courses, sessions, or exercises that are added to the system.

Project Description

This project designed a system for a meditation app that includes classes for users, courses, instructors, and exercises.
Here are the brief descriptions for each class and interface we need for this project:

- Course:

Represents a course that can be taken by a user. It contains information such as the course name, description, instructor, and sessions. Also has a method for retrieving required courses based on a certain criterion.

- Exercise:

An abstract class that represents an exercise in a meditation program. It contains basic information about the exercise, such as its name, description, and duration, and a list of records associated with the exercise.

- ExerciseInterface:

An interface that provides method to control the execution of an exercise, and it includes methods for starting, pausing, resuming and stopping the exercise.

- ExerciseRecord:

An abstract class that represents a record of an exercise. It stores information about when the exercise was performed.

- ExerciseRecordList:

An abstract class that represents a list of exercise records. It provides methods for adding records, filtering records based on a condition, and sorting records.

- Instructor:

Represents an instructor who can teach a course or lead a meditation exercise. Contains information such as the instructor's name and bio.

- Meditation:

Represents a meditation exercise that can be included in a session. Contains information such as the meditation name, description, duration, and instructor.

- MeditationRecord:

A class that extends ExerciseRecord, specifically for meditation exercises. Represents a record of a meditation exercise performed by a user. Contains information such as the start time, duration, associated meditation exercise, and emotional state of the user after the exercise.

- MeditationRecordList:

A class that extends ExerciseRecordList for meditation records. It provides methods for filtering and sorting meditation records.

- Session:

Represents a meditation session that can be included in a course, which contains a name, description, duration, and a list of exercises.

- SessionRecord:

A class that extends ExerciseRecord for session records. It stores the start and end times of a session.

- SessionRecordList:

A class that extends ExerciseRecordList for session records. It provides methods for filtering and sorting session records.

- Module:

Module class represents a module within the meditation app, which has a name, description, and a list of courses. Each course can have a list of sessions, and each session can have a list of exercises. This can be useful for organizing and representing large amounts of data in a structured and hierarchical manner.

- NotificationSystem:

Represents a notification system that can send notifications to registered users about new courses or updates to existing courses. It provides methods include addCourse, removeObserver, and notifyObservers to manage the list of observers.

- Observer:

Represents an interface that defines the contract for objects that need to be notified when the state of another object changes. The user is notified when the subject it is registered with undergoes a change.

- Subject:

An interface contains methods to register, remove, and notify observers. It is being observed by one or more observers, and it maintains a list of observers that are interested in being notified of changes. The “Exercise” class, “Session” class, and “Course” class implement the “Subject” interface to allow observers to subscribe and receive notifications.

- Payment:

Represents a payment made by a user for a course or subscription. Contains information such as the payment amount, date, and payment method.

- Subscription:

Represents a subscription to a course or module. Contains information such as the subscription start date, end date, and subscription level.

- User:

Represents a user of the meditation app. Contains information such as the user's name, email, password, and meditation/session history. Also represents as a model in the MVC design pattern.

- UserController:

A controller acts as the controller in the MVC design pattern for handling user data and updates.

- UserView:

A view class acts as a graphical user interface (GUI) class that displays the information of a user. It extends JPanel and includes text fields for user's name, email, and password, as well as JLists for the user's meditation and session history.

- Driver:

The Driver class is the main class that runs the meditation application. It creates and manages various objects such as instructors, meditation exercises, sessions, courses, modules, users, and payment and subscription information. It also interacts with users by displaying information and prompting for input via the console and GUI view window. Additionally, it demonstrates the functionality of various classes in the application such as MeditationRecord, SessionRecord, and ExerciseRecordList by creating instances and printing their details. Finally, it also showcases the notification system by registering and unregistering users as observers and notifying them of updates to courses.

Chapter 1: Recursion in Practice

This project applies the concept of recursion with the module hierarchy structure. In this program, the module hierarchy is implemented as a tree structure, where each module can contain a list of courses. Each course can have a list of sessions, and each session can have a list of exercises. This can be useful for organizing and representing large amounts of data in a structured and hierarchical manner. Recursion is used to traverse this tree structure in a depth-first manner, where each module is treated as the root node of a sub-tree.

For instance, the application will recursively traverse the tree to show all the courses and sessions within the selected module when the user clicks on a module. A module is passed as an input to a method, which then calls itself with each of the module's sub-modules until it reaches the root of the hierarchy.

Let's see the constructors for the hierarchy structure of module, courses, sessions, exercises classes:
Each module contains a list of courses:

```
/*

* Constructor constructs a module with the specified name, description and courses

* @param name the name of the module

* @param description the description of the module

* @param courses the list of courses in the module

*/

public Module(String name, String description, List<Course> courses) {

this.name = name;

this.description = description;

this.courses = courses;

}
```

Each course has a list of sessions:

```
/*

* Constructer constructs a new Course object with the given name, description, sessions, instructor, credits, and required status.
```

```
* @param name the name of the course

* @param description the description of the course

* @param sessions the list of sessions for the course

* @param instructor the instructor for the course

* @param credits the number of credits for the course

* @param required whether the course is required or not

*/

public Course(String name, String description, List<Session> sessions,Instructor instructor, int credits,
boolean required) {

this.name = name;

this.description = description;

this.sessions = sessions;

this.instructor = instructor;

this.credits = credits;

this.required = required;

observers = new ArrayList<Observer>();

}
```

Each session has a list of exercises:

```
/*

* Constructor for the Session class

* @param name the name of the session

* @param description the description of the session

* @param duration the duration of the session

* @param exercises the exercises in the session
```

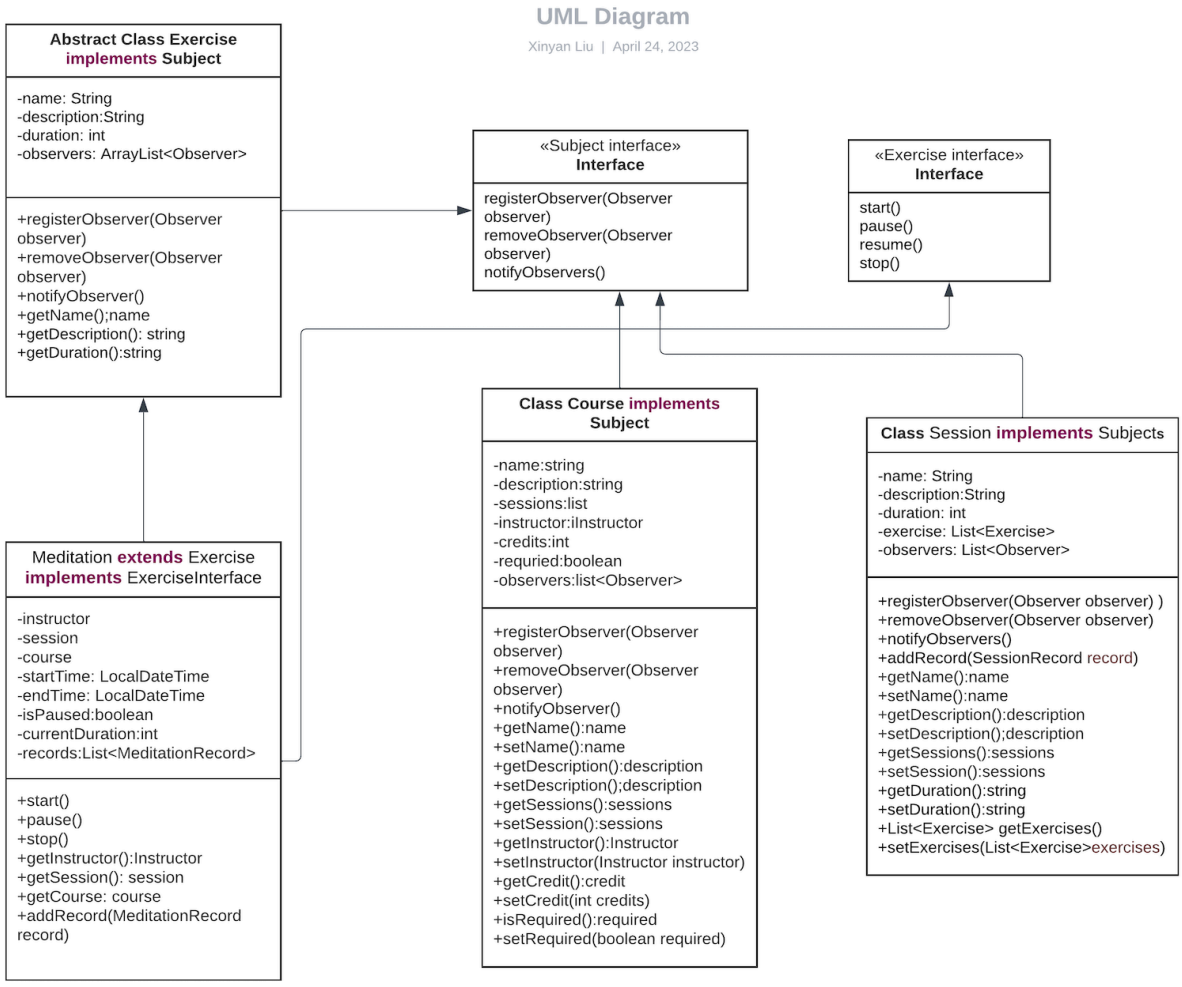
```
*/

public Session(String name, String description, int duration, List<Exercise> exercises) {

this.name = name;

this.description = description;

this.duration = duration;

this.exercises = exercises;

observers = new ArrayList<Observer>();

}
```

This approach allows for a flexible hierarchy structure, where modules, courses, and sessions can be added or removed without changing the traversal logic. We can create a printModule method to print the module hierarchy structure starting from the specified module recursively:

```
/*

* Prints the module hierarchy structure starting from the specified module recursively

* @param module the starting module

*/

public static void printModule(Module module) {

System.out.println("Module: " + module.getName());

for (Course course : module.getCourses()) {

System.out.println("\tCourse: " + course.getName());

for (Session session : course.getSessions()) {

System.out.println("\t\tSession: " + session.getName());

}

}

}
```

Chapter 2: Abstract Classes and Interfaces

The abstracted classes and interfaces are used in various parts of this project to provide a structure for defining and implementing behaviors across different classes:

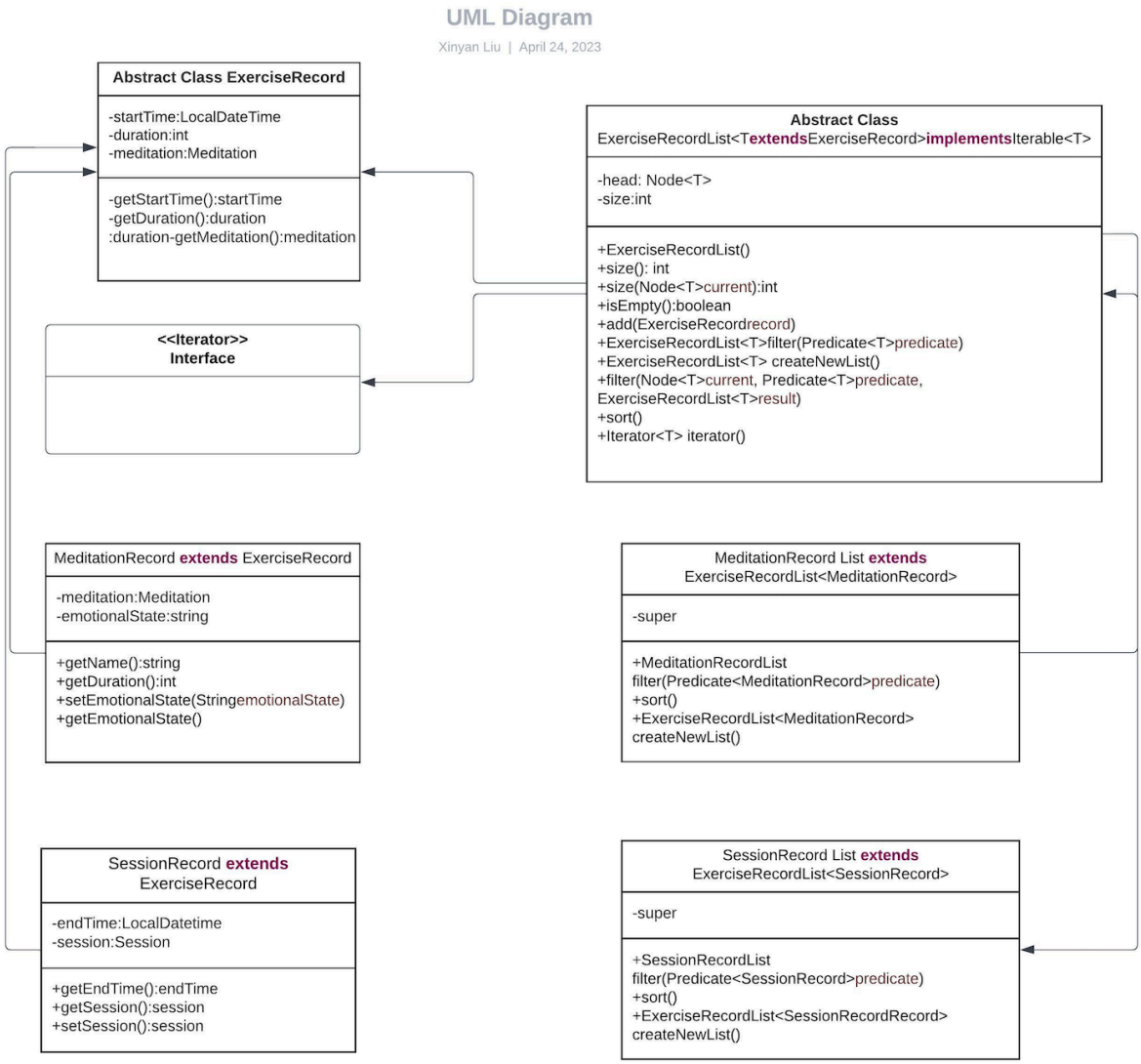


- The “Subject” interface contains methods to register, remove, and notify observers, and the “Exercise” class, “Session” class, and “Course” class implement the “Subject” interface to allow observers to subscribe and receive notifications.

- The “Exercise” class is an abstract class that implements the “Subject” class. It defines common properties and behaviors for all exercise types in the system. The Meditation class extend the Exercise class and implement their own unique properties and behaviors, while still inheriting the common properties and behaviors defined in the abstract class.

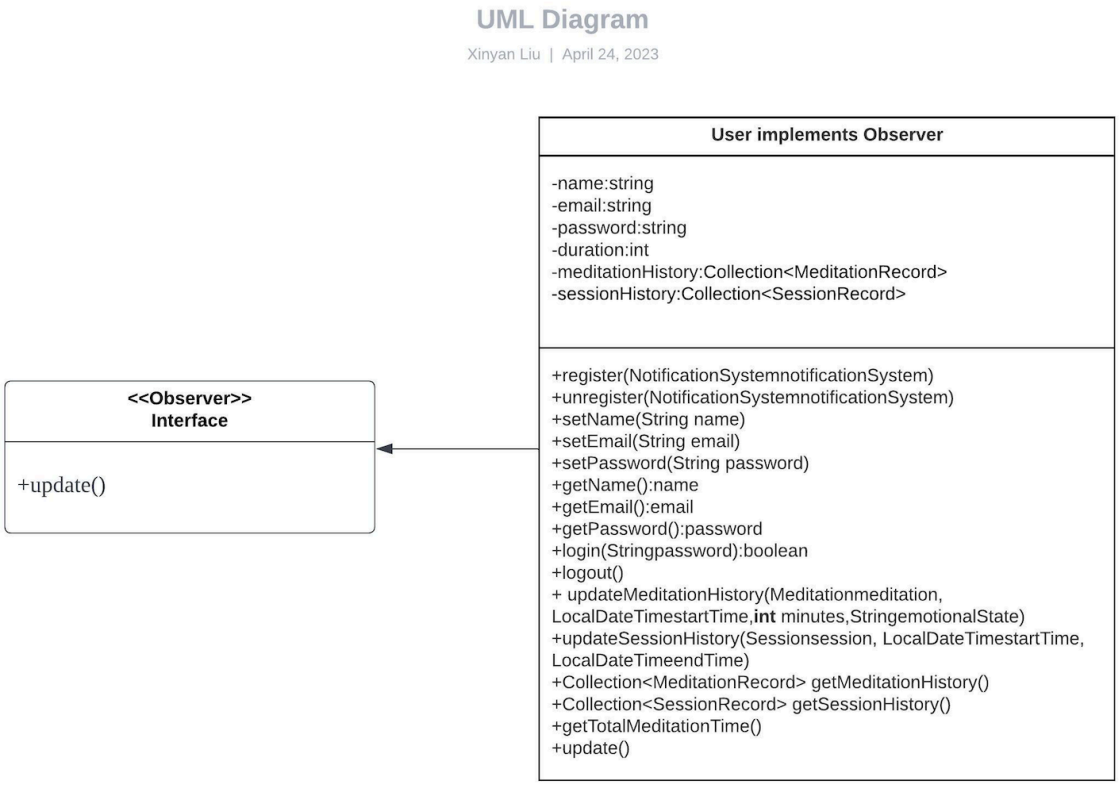
- The "ExerciseInterface" is the interface that provides method to control the execution of an exercise, and it includes methods for starting, pausing, resuming and stopping the exercise.

- "Meditation" class extends the "Exercise" class and implement the "ExerciseInterface". It keeps tracks of the meditation's instructor, session, course, start time, end time, current duration, pause or stop the meditation exercise, and also has a list of meditation records associated with it.



- The "ExerciseRecord" is an abstract class, and it can be extended by specific types of records. "MeditationRecord" and "SessionRecord" classes extend the "ExerciseRecord" class and inherit the basic properties to represent a single meditation session.

- The “ExerciseRecordList” is an abstract class represents a linked list data structure for storing each exercise record. The MeditationRecordList and SessionRecordList classes extend the ExerciseRecordList class and add additional methods specific to meditation and session records, respectively. These classes inherit the basic linked list functionality from the ExerciseRecordList class,but can also provide specialized behavior for their specific use cases.



- The “Observer” interface is part of the Observer design patter, it includes an update method, which is called by the subject to notify the observer. The “User” class implements the “Observer” interface by implementing the update() method. When a notification is received, the update() method prints a message to the console indicating that the user has been notified. The User class is able to decouple itself from the NotificationSystem and receive notifications.

Overall, abstracted classes and interfaces provide a flexible way to define and implement behaviors across different classes.

Chapter 3: Abstracted Linked Lists

The concept of abstracted linked lists has been applied in this project through the use of the ExerciseRecordList class and its subclasses. An abstracted linked list is a linked list that is defined using an abstract class or interface, which allows for the creation of a list with generic methods and properties that can be implemented in a subclass. This creates a more flexible and reusable structure for implementing specific linked lists.

In the ExerciseRecordList class, the linked list is defined with a generic type T, which is constrained to be a subtype of ExerciseRecord. This allows for the creation of a linked list that can store any type of ExerciseRecord. The class includes abstract methods, such as createNewList() and sort(), which are implemented in the MeditationRecordList and SessionRecordList subclasses, respectively.

Recursion is also used in the ExerciseRecordList class to implement the size() and filter() methods. The size() method uses a recursive helper method to traverse the linked list and count the number of nodes in the list. Similarly, the filter() method uses a recursive helper method to traverse the linked list and apply a predicate to each element to determine whether it should be included in the filtered list.

Overall, the use of abstracted linked lists and recursion in this project provides a flexible and efficient way to implement linked lists for different types of ExerciseRecords.

This is the complete code for the ExerciseRecordList class:

```
/**

* This abstract class represents a linked list data structure for storing each exercise record.

*

* @param <T> a type parameter indicating the type of exercise record that the list can store

*/



public abstract class ExerciseRecordList<T extends ExerciseRecord> implements Iterable<T>{

protected Node<T> head;

protected int size;

/*
```

```
* A reference to the head of the linked list.

*/

public ExerciseRecordList(){

head=null;

size=0;

}

/*

* Returns the number of nodes in the linked list.

* @return the size of the linked list

*/

public int size() {

return size(head);

}

/*

* Recursive method that computes the size of the linked list.

* @param current A reference to the current node in the list.

* @return the size of the linked list.

*/

private int size(Node<T> current) {

if (current == null) {

return 0;

}

return 1 + size(current.next);
```

```
}

/*

* Checks whether the linked list is empty

* @return true if the linked list is empty, false otherwise

*/

public boolean isEmpty() {

return size == 0;

}

/*

* Adds an exercise record to the end of the linked list

* @param record The exercise record to add

*/

public void add(ExerciseRecord record) {

Node<ExerciseRecord> newNode = new Node<>(record);

if (head == null) {

head = (ExerciseRecordList.Node<T>) newNode;

} else {

Node<ExerciseRecord> current = (ExerciseRecordList.Node<ExerciseRecord>) head;

while (current.next != null) {

current = current.next;

}

current.next = newNode;

}
```

```
size++;

}

/*

* Filters the linked list based on a specified predicate

* @param predicate A predicate that specifies the condition for filtering

* @return an ExerciseRecordList object containing the filtered exercise records

*/

public ExerciseRecordList<T> filter(Predicate<T> predicate) {

ExerciseRecordList<T> result = createNewList();

filter(head, predicate, result);

return result;

}

/*

* Abstract method that creates a new instance of the ExerciseRecordList subclass

* @return a new instance of the ExerciseRecordList subclass

*/

protected abstract ExerciseRecordList<T> createNewList();

/*

* recursive method that filters the linked list based on a specified predicate

*

* @param current reference to the current node in the list

* @param predicate that specifies the condition for filtering

* @param result: an ExerciseRecordList object to store the filtered exercise records
```

```
*/

private void filter(Node<T> current, Predicate<T> predicate, ExerciseRecordList<T> result) {

if (current == null) {

return;

}

if (predicate.test(current.data)) {

result.add(current.data);

}

filter(current.next, predicate, result);

}

/*

* Abstract method that sorts the linked list

*/

public abstract void sort();

/*

* Returns an iterator over the elements of the linked list

* @return an iterator over the elements of the linked list

*/

@Override

public Iterator<T> iterator(){

return new ExerciseRecordListIterator<>(head);

}

/*
```

```
* Node class for a linked list, contains the data and a reference to the next node

* @param <T> the type of data that the node holds

*/

protected static class Node<T> {

T data;

Node<T> next;

public ExerciseRecordList.Node<SessionRecord> prev;

public Node(T data) {

this.data = data;

next = null;

}

}

/*

* Iterator class for the linked list

* @param <T> the type of data that the node holds

*/

protected static class ExerciseRecordListIterator<T extends ExerciseRecord> implements Iterator<T>
{

private Node<T> current;

public ExerciseRecordListIterator(Node<T> head) {

current = head;

}

public boolean hasNext() {

return current != null;
```

```
}

public T next() {

T data = current.data;

current = current.next;

return data;

}

}

}
```

MeditaitonRecordList class:

```
/**

* This is an implementation of a linked list data structure abstract class for storing each exercise records

*/

import java.util.function.Predicate;

public class MeditationRecordList extends ExerciseRecordList<MeditationRecord> {

public MeditationRecordList() {

super();

}

/*

* Filters the meditation record list based on the given predicate

* @param predicate the predicate used to filter the meditation record list

* @return a new MeditationRecordList that contains only the meditation records that match the given predicate

*/

@Override
```

```
public MeditationRecordList filter(Predicate<MeditationRecord> predicate) {

return (MeditationRecordList) super.filter(predicate);

}

/*

* Bubble sort method sorts the list in ascending order of duration or date

* iterates over the list using two nested loops

* Comparing adjacent elements and swap them if they are out of order

* The outer loop continues until the list is sorted

*/

@Override

public void sort() {

//Bubble sort

boolean sorted = false;

Node<MeditationRecord> current;

Node<MeditationRecord> next;

while (!sorted) {

sorted = true;

current = head;

next = head.next;

while (next != null) {

if (current.data.getDuration() > next.data.getDuration()) {

// Swap the nodes

MeditationRecord temp = current.data;
```

```
current.data = next.data;

next.data = temp;

sorted = false;

}

current = current.next;

next = next.next;

}

}

}

/*

* Creates and returns a new MeditationRecordList instance

*

* @return a new MeditationRecordList instance

*/

@Override

public ExerciseRecordList<MeditationRecord> createNewList() {

return new MeditationRecordList();

}

}
```

SessionRecordList class:

```
/**

* This is an implementation of a linked list data structure abstract class for storing each exercise records

*/
```

```
import java.util.function.Predicate;

public class SessionRecordList extends ExerciseRecordList<SessionRecord>{

/*

* Constructor for creating a new empty SessionRecordList.

*/

public SessionRecordList(){

super();

}

/*

* Returns the total duration of all session records in the list.

* @return the total duration of all session records in the list

*/

public int getTotalDuration() {

int totalDuration = 0;

for (SessionRecord record : this) {

totalDuration += record.getDuration();

}

return totalDuration;

}

/*

* Filters the session record list based on the given predicate.

* @param predicate the predicate used to filter the session record list

* @return a new SessionRecordList that contains only the session records that match the given
predicate
```

```
*/

public SessionRecordList filter(Predicate<SessionRecord> predicate) {

return (SessionRecordList) super.filter(predicate);

}

/*

* Sorts the session record list using the insertion sort algorithm.

* Overrides the sort method from the ExerciseRecordList class.

*/

@Override

public void sort() {

//

Node<SessionRecord> current = head.next;

while (current != null){

SessionRecord currentData = current.data;

Node <SessionRecord> prev = current.prev;

while (prev != null && prev.data.getDuration() > currentData.getDuration()){

prev.next.data = prev.data;

prev = prev.prev;

}

if(prev == null){

head.data = currentData;

}

else {
```

```
prev.next.data = currentData;

}

current = current.next;

}

}

/*

* Creates a new SessionRecordList.

* Overrides the createNewList method from the ExerciseRecordList class.

* @return a new SessionRecordList

*/

@Override

public ExerciseRecordList<SessionRecord> createNewList() {

return new SessionRecordList();

}

}
```

Chapter 4: Higher order functions map, filter, and fold

The Higher order functions “map”, “filter”, “fold” is used in this project in several places.

The “ExerciseRecordList” class implements the filter method, which is a higher-order function that takes a Predicate object as an argument and returns a new “ExerciseRecordList” object containing only the exercise records that satisfy the predicate. This is a method that recursively filters the list based on given predicate. The “createNewList” method is declared as abstract in the ExerciseRecordList class. This method is responsible for creating a new instance of a concrete class that extends ExerciseRecordList. The filter method calls “createNewList” to create a new instance of the concrete class, and passes this instance to the filter method that performs the filtering logic.

```
/*

* Filters the linked list based on a specified predicate

* @param predicate A predicate that specifies the condition for filtering

* @return an ExerciseRecordList object containing the filtered exercise records

*/

public ExerciseRecordList<T> filter(Predicate<T> predicate) {

ExerciseRecordList<T> result = createNewList();

filter(head, predicate, result);

return result;

}

/*

* Abstract method that creates a new instance of the ExerciseRecordList subclass

* @return a new instance of the ExerciseRecordList subclass

*/

protected abstract ExerciseRecordList<T> createNewList();

/*
```

```
* recursive method that filters the linked list based on a specified predicate

* @param current reference to the current node in the list

* @param predicate that specifies the condition for filtering

* @param result: an ExerciseRecordList object to store the filtered exercise records

*/

private void filter(Node<T> current, Predicate<T> predicate, ExerciseRecordList<T> result) {

if (current == null) {

return;

}

if (predicate.test(current.data)) {

result.add(current.data);

}

filter(current.next, predicate, result);

}
```

To use this modified filter method, we need to implement the “createNewList” method in each concrete class that extends “ExerciseRecordList”.The filter method can be implemented in the “MeditationRecordList” and “SessionRecordList” classes using the filter method of the parent “ExerciseRecordList” class. Since the “MeditationRecordList” and “SessionRecordList” classes extend the “ExerciseRecordList” class, they will inherit the filter method.
Such like the filter method in the MeditationRecordList class overrides the “filter” method inherited from the “ExerciseRecordList” class and applies it specifically to “MeditationRecordList”.

```
/*

* Filters the meditation record list based on the given predicate

* @param predicate the predicate used to filter the meditation record list

* @return a new MeditationRecordList that contains only the meditation records that match the given predicate
```

```
*/

@Override

public MeditationRecordList filter(Predicate<MeditationRecord> predicate) {

return (MeditationRecordList) super.filter(predicate);

}
```

Filter method in SessionRecordList:

```
/*

* Filters the session record list based on the given predicate.

* @param predicate the predicate used to filter the session record list

* @return a new SessionRecordList that contains only the session records that match the given predicate

*/

public SessionRecordList filter(Predicate<SessionRecord> predicate) {

return (SessionRecordList) super.filter(predicate);

}
```

The concept of map, filter, fold, are used in the “Module” class in this project: “getTotalCredits()” method uses the “mapToInt()” and “sum()” functions to transform the list of courses into a list of their credit values and then calculate the sum of all credit values.

```
/*

* Returns the total number of credits for all courses in the module

* getTotalCredits method uses the "fold" function (reduce) to

* transform the list of courses into a list of their credit values

* and then calculate the sum of all credit values

*

* @return the total number of credits for all courses in the module

*/
```

```
public int getTotalCredits() {

return courses.stream()

.mapToInt(Course::getCredits)

.sum();

}
```

"getCourseNames()" method uses the "map()" function to transform the list of courses into a list of their names.

```
/*

* getCourseNames method uses the "map" function to transform the list of courses into a list of their names

*/

public List<String> getCourseNames() {

return courses.stream()

.map(Course::getName)

.collect(Collectors.toList());

}
```

"getRequiredCourses(Predicate<Course> predicate)" method uses the "filter()" function to filter the list of courses into a list of required courses based on a certain criterion.

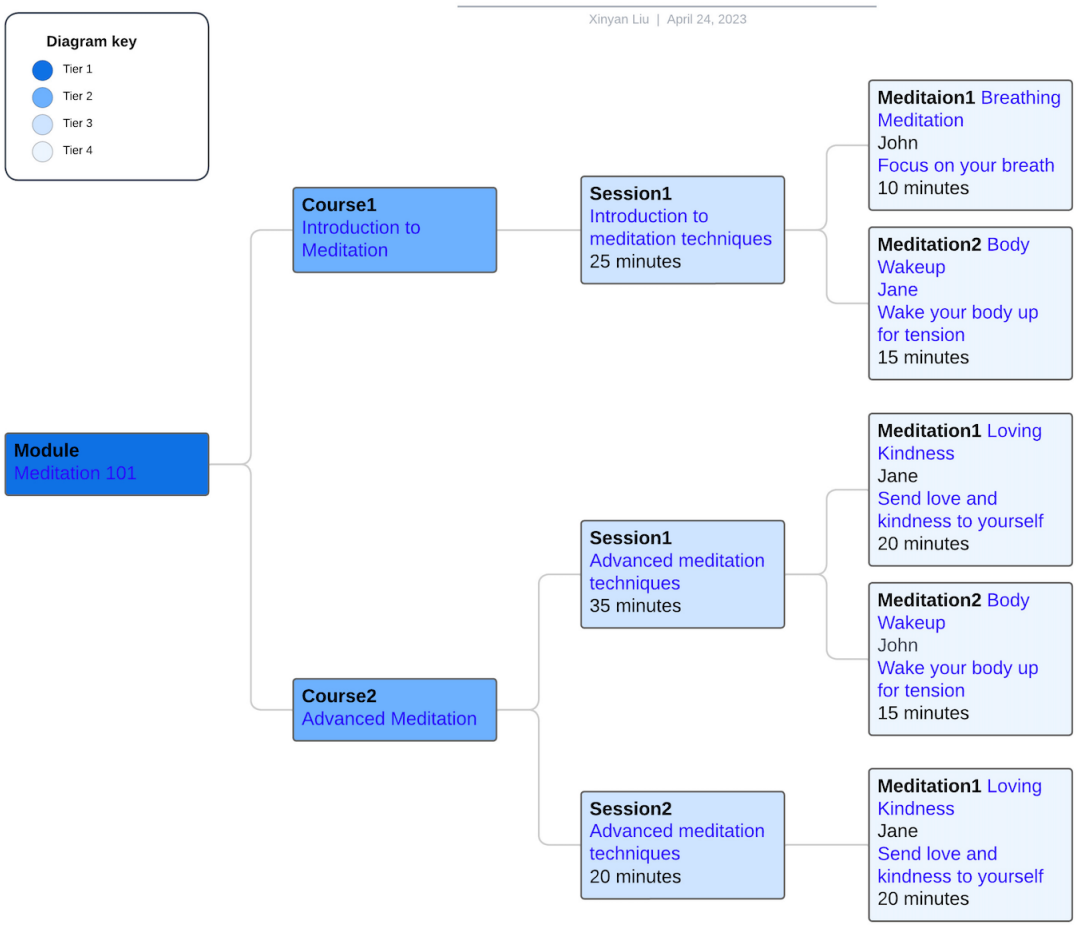
```
/*

* getRequiredCourses method uses the "filter" function to filter the list of courses into a list of required courses based on a certain criterion

*/

public List<Course> getRequiredCourses(Predicate<Course> predicate) {

return courses.stream()

.filter(predicate)

.collect(Collectors.toList());

}
```

Chapter 5: Hierarchical Data Representation

The concept of hierarchical data representation is applied in this project in several ways:

- The Module class contains a list of Course objects, which in turn contains a list of Session objects. This represents a hierarchical structure of a meditation program, where a program is composed of modules, each module is composed of courses, and each course is composed of sessions.
- The Meditation and Session classes contains a list of MeditationRecord and SessionRecord objects respectively, which represent the user's history of meditation and sessions. This hierarchical structure can represent the user's progress and history over time.
- The Observer design pattern is used to create a hierarchical structure of notification systems and users. The NotificationSystem object contains a list of Observer objects (User objects), which are notified when a new course is updated. This represents a hierarchical structure of users and their subscription to a notification system.

For example, this diagram shows the hierarchical structure representation for the modules, courses, sessions, and meditation exercises.



Chapter 6: MVC Design

MVC (Model-View-Controller) is a design pattern used to separate an application into three interconnected components: the model, the view, and the controller. For this project, here are some ways that can apply the concept of MVC design to this project:

Model:

The model represents the data and the logic of the application. In this project, the model could include classes like Course, Session, Exercise, ExerciseRecord, ExerciseRecordList, etc. These classes would contain the data and methods that manipulate the data.
For example:

- Exercise: represents an exercise, with a name, description, duration, and list of exercise records
- Meditation: a subclass of Exercise that represents a meditation exercise, with an additional instructor field
- Session: represents a session, with a name, description, duration, and list of exercises
- Course: represents a course, with a name, description, and list of sessions
- Module: represents a module, with a name, description, and list of courses
- Instructor: represents an instructor, with a name and bio
- User: represents a user, with a name, email, password, and lists of meditation and session records
- Payment: represents a payment, with an amount, date, and payment method
- Subscription: represents a subscription, with a start date, end date, and subscription level
- ExerciseRecord: represents a record of an exercise, with a start time, duration, emotions, and a reference to the exercise it corresponds to
- MeditationRecord: a subclass of ExerciseRecord that represents a record of a meditation exercise, with an additional reference to the meditation it corresponds to
- SessionRecord: a subclass of ExerciseRecord that represents a record of a session, with an additional end time and reference to the session it corresponds to

View:

The view is responsible for rendering the data to the user. In this project, the view could include classes like MeditationProgramUI, CourseUI, SessionUI, ExerciseUI, etc. These classes would display the data to the user in a graphical user interface.
For example:

- MainMenuView: displays the main menu and handles user input
- ModuleView: displays module information and handles user input
- CourseView: displays course information and handles user input
- SessionView: displays session information and handles user input
- ExerciseView: displays exercise information and handles user input
- PaymentView: displays payment information and handles user input

- SubscriptionView: displays subscription information and handles user input
- UserView: displays user information and handles user input

Controller:

The controller is responsible for handling user input and updating the model and view accordingly. In this project, the controller could include classes like ModuleController, CourseController, SessionController, ExerciseController, UserController,etc. These classes would receive input from the user via the view, update the model accordingly, and then update the view to display the updated data.
For example:

- MainMenuController: handles user input from the main menu view and navigates to the appropriate sub-views/controllers
- ModuleController: handles user input from the module view and navigates to the appropriate sub-views/controllers
- CourseController: handles user input from the course view and navigates to the appropriate sub-views/controllers
- SessionController: handles user input from the session view and navigates to the appropriate sub-views/controllers
- ExerciseController: handles user input from the exercise view and navigates to the appropriate sub-views/controllers
- PaymentController: handles user input from the payment view and performs payment-related actions
- SubscriptionController: handles user input from the subscription view and performs subscription-related actions
- UserController: handles user input from the user view and performs user-related actions

I have included the code for the User model, User controller, and user view as the code example:

User model:

```
/**

* User class stores the MeditationRecords and SessionRecord objects

*/

import java.time.LocalDateTime;

public class User implements Observer{

private String name;
```

```
private String email;

private String password;

private int duration;

private Collection<MeditationRecord> meditationHistory;

private Collection<SessionRecord> sessionHistory;

public User(String name, String email, String password) {

this.name = name;

this.email = email;

this.password = password;

this.meditationHistory = new ArrayList<>();

this.sessionHistory = new ArrayList<>();

}

/*

* Registers the user as an observer with the provided NotificationSystem object.

* @param notificationSystem the NotificationSystem object to register with

*/

public void register(NotificationSystem notificationSystem) {

notificationSystem.addObserver(this);

}

/*

* Unregisters the user as an observer with the provided NotificationSystem object.

* @param notificationSystem the NotificationSystem object to unregister with

*/
```

```
public void unregister(NotificationSystem notificationSystem) {

notificationSystem.removeObserver(this);

}

/*

* Sets the name of the user.

* @param name the name to set

*/

public void setName(String name) {

this.name = name;

}

/*

* Sets the email of the user.

* @param email the email to set

*/

public void setEmail(String email) {

this.email = email;

}

/*

* Sets the password of the user.

* @param password the password to set

*/

public void setPassword(String password) {

this.password = password;
```

```
}
/*
* Returns the name of the user.
* @return the name of the user
*/
public String getName() {
return name;
}
/*
* Returns the email of the user.
* @return the email of the user
*/
public String getEmail() {
return email;
}
/*
* Returns the password of the user.
* @return the password of the user
*/
public String getPassword() {
return password;
}
/*
```

```
* Returns true if the provided password matches the user's password, false otherwise.

* @param password the password to check

* @return true if the provided password matches the user's password, false otherwise

*/

public boolean login(String password) {

return this.password.equals(password);

}

/*

* Logs the user out of the system.

*/

public void logout() {

}

/*

* Updates the user's meditation history with a new record.

* @param meditation the Meditation object associated with the record

* @param startTime the start time of the meditation

* @param minutes the duration of the meditation in minutes

*/

public void updateMeditationHistory(Meditation meditation, LocalDateTime startTime, int
minutes,String emotionalState) {

MeditationRecord record = new MeditationRecord(startTime, duration, meditation);

meditation.addRecord(record);

record.setEmotionalState(emotionalState);

this.meditationHistory.add(record);
```

```
}

/*

* Updates the user's session history with a new record.

* @param session the Session object associated with the record

* @param startTime the start time of the session

* @param endTime the end time of the session

*/

public void updateSessionHistory(Session session, LocalDateTime startTime, LocalDateTime
endTime) {

SessionRecord record = new SessionRecord(startTime, endTime, session);

session.addRecord(record);

this.sessionHistory.add(record);

}

/*

* Returns the user's meditation history.

* @return the user's meditation history

*/

public Collection<MeditationRecord> getMeditationHistory() {

return meditationHistory;

}

/*

* Returns the user's session history.

* @return the user's session history

*/
```

```
public Collection<SessionRecord> getSessionHistory() {

return sessionHistory;

}

/*

* Returns the total number of minutes the user has meditated.

* @return the total number of minutes the user has meditated

*/

public int getTotalMeditationTime() {

int totalTime = 0;

for (MeditationRecord record : meditationHistory) {

totalTime += record.getDuration();

}

return totalTime;

}

/*

* Prints a notification message to the console indicating that a user has been notified

*/

@Override

public void update() {

System.out.println("User " + this.name + " has been notified");

}

}
```

User Controller:

```
/**
* The UserController class acts as the controller in the Model-View-Controller (MVC) design pattern.
* It handles user input and updates the user interface accordingly.
*/
public class UserController {
private User model;
private UserView view;
/*
* Constructor for the UserController class.
* @param model the User model that this controller interacts with.
* @param view the UserView that displays the User model to the user.
*/
public UserController(User model, UserView view) {
this.model = model;
this.view = view;
}
/*
* Sets the name of the User model.
* @param name the new name of the User model.
*/
public void setUserName(String name) {
model.setName(name);
```

```
}

/*

* Returns the name of the User model.

* @return the name of the User model.

*/

public String getUserName() {

return model.getName();

}

/*

* Sets the email of the User model.

* @param email the new email of the User model.

*/

public void setUserEmail(String email) {

model.setEmail(email);

}

/*

* Gets the email of the User model.

* @return the email of the User model.

*/

public String getUserEmail() {

return model.getEmail();

}

/*
```

```
* Sets the password of the User model.

* @param password the new password of the User model.

*/

public void setUserPassword(String password) {

model.setPassword(password);

}

/*

* Gets the password of the User model.

* @return the password of the User model.

*/

public String getUserPassword() {

return model.getPassword();

}

/*

* Updates the UserView to reflect changes made to the User model.

*/

public void updateView() {

view.display();

}

/*

* Sets the UserView that displays the User model to the user.

* @param view the new UserView that displays the User model to the user.

*/
```

```
public void setView(UserView view) {

this.view = view;

}

}
```

User view:

```
public class UserView extends JPanel{

private User user;

private JTextField nameField;

private JTextField emailField;

private JTextField passwordField;

private JLabel meditationHistoryLabel;

private JList<MeditationRecord> meditationHistoryList;

private JLabel sessionHistoryLabel;

private JList<SessionRecord> sessionHistoryList;

private UserController controller;

/*

*Display the user's information on the console.

*/

public void display() {

System.out.println("User Information");

System.out.println("----------------");

System.out.println("Name: " + user.getName());

System.out.println("Email: " + user.getEmail());
```

```
}

/*

* Constructor for the UserView class.

* @param model the User object to be displayed

*/

public UserView(User model) {

this.user = model;

setLayout(new BorderLayout());

// Create and add name field

JPanel namePanel = new JPanel(new GridLayout(1, 2));

JLabel nameLabel = new JLabel("Name:");

nameField = new JTextField(model.getName());

namePanel.add(nameLabel);

namePanel.add(nameField);

add(namePanel, BorderLayout.NORTH);

// Create and add email field

JPanel emailPanel = new JPanel(new GridLayout(1, 2));

JLabel emailLabel = new JLabel("Email:");

emailField = new JTextField(model.getEmail());

emailPanel.add(emailLabel);

emailPanel.add(emailField);

add(emailPanel, BorderLayout.CENTER);

// Create and add password field
```

```
JPanel passwordPanel = new JPanel(new GridLayout(1, 2));

JLabel passwordLabel = new JLabel("Password:");

passwordField = new JTextField(model.getPassword());

passwordPanel.add(passwordLabel);

passwordPanel.add(passwordField);

add(passwordPanel, BorderLayout.SOUTH);

// Create and add meditation history list

JPanel meditationHistoryPanel = new JPanel(new BorderLayout());

meditationHistoryLabel = new JLabel("Meditation History:");

meditationHistoryList = new JList<>(model.getMeditationHistory().toArray(new
MeditationRecord[0]));

meditationHistoryList.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);

JScrollPane meditationHistoryScrollPane = new JScrollPane(meditationHistoryList);

meditationHistoryPanel.add(meditationHistoryLabel, BorderLayout.NORTH);

meditationHistoryPanel.add(meditationHistoryScrollPane, BorderLayout.CENTER);

add(meditationHistoryPanel, BorderLayout.WEST);

// Create and add session history list

JPanel sessionHistoryPanel = new JPanel(new BorderLayout());

sessionHistoryLabel = new JLabel("Session History:");

sessionHistoryList = new JList<>(model.getSessionHistory().toArray(new SessionRecord[0]));

sessionHistoryList.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);

JScrollPane sessionHistoryScrollPane = new JScrollPane(sessionHistoryList);

sessionHistoryPanel.add(sessionHistoryLabel, BorderLayout.NORTH);

sessionHistoryPanel.add(sessionHistoryScrollPane, BorderLayout.CENTER);
```

```
add(sessionHistoryPanel, BorderLayout.EAST);

}

/*

* Returns the name entered by the user in the name field.

* @return the name entered by the user

*/

public String getName() {

return nameField.getText();

}

/*

* Returns the email entered by the user in the email field.

* @return the email entered by the user

*/

public String getEmail() {

return emailField.getText();

}

/*

* Returns the password entered by the user in the password field.

* @return the password entered by the user

*/

public String getPassword() {

return passwordField.getText();

}
```

```
/*
* Sets the User object to be displayed.
* @param model the User object to be displayed
*/
public void setModel(User model) {
this.user = model;
nameField.setText(model.getName());
emailField.setText(model.getEmail());
passwordField.setText(model.getPassword());
meditationHistoryList.setListData(model.getMeditationHistory().toArray(new MeditationRecord[0]));
sessionHistoryList.setListData(model.getSessionHistory().toArray(new SessionRecord[0]));
}
/*
* Sets the controller for the UserView object.
* @param controller the UserController object that controls this view
*/
public void setController(UserController controller) {
this.controller = controller;
}
}
```

We can create a GUI view using JFrame. We create a new “JFrame” and add the “UserView” object to its content panel, and make the frame visible. This will display the GUI view for the “User” object.

Overall, the MVC design pattern can help to separate the concerns of the application, making it easier to develop, test, and maintain.

Chapter 7: SOLID Principles

SOLID principle is an acronym for five design principles that can help software designs more understandable, flexible, and maintainable. In this project, we have applied the concept of SOLID principles throughout the design and the structure of this project.

- *Single Responsibility Principle (SRP):*
  Each class has a single responsibility and does not have more than one reason to change. For example, the User class is responsible for managing user data and the UserController class is responsible for controlling user-related actions. The Course class is responsible for managing information about courses, while the NotificationSystem class is responsible for sending notifications.

- *Open-Closed Principle (OCP):*
  The project is designed in a way that new features can be added without modifying the existing code. For example, new types of exercises or instructors can be added without changing the existing classes. New types of courses can be added to the system without changing the existing Course class by subclassing it and adding new methods.

- *Liskov Substitution Principle (LSP):*
  The project follows the LSP by ensuring that subclasses can be used in place of their parent classes without affecting the correctness of the program. For example, the Meditation class is a subclass of Exercise, and it can be used in place of Exercise in the Session class.

- *Interface Segregation Principle (ISP):*
  The project follows the ISP by ensuring that interfaces are tailored to specific client needs. For example, the UserView interface only contains methods related to displaying user data, while the UserController interface only contains methods related to controlling user actions. The Observer interface has only one method, update(), which is the only method needed by clients who want to observe changes in the system.

- *Dependency Inversion Principle (DIP):*
  The project follows the DIP by depending on abstractions instead of concrete implementations. For example, the UserController depends on the UserView interface, not on a specific implementation of the view. The Course class depends on the Session interface rather than a concrete implementation of Session, which allows different types of sessions to be used interchangeably with Course.

Chapter 8: Other Design Pattern: Observer Pattern

Observer Pattern can be used to notify users about new courses, sessions, or exercises that are added to the system. Users can register as observers and receive notifications when new content is available.

First we create the "Observer" interface:

```
/**

* Observer interface is part of the Observer design pattern

* include an update method, which is called by the subject to notify the observer

*/

public interface Observer {

public void update();

}
```

This interface defines a single method update that is called by the Observable when a notification is sent.

Then we can create the NotificationSystem and one User objects. This class implements the Observable interface and has a list of registered observers and a list of courses. It provides a method addCourse that adds a new course to the list of courses and sends a notification to all registered observers by calling the notifyObservers method. The addObserver, removeObserver, and notifyObservers methods are implemented to manage the list of observers.

```
public class NotificationSystem{

private List<Course> courses;

private List<Observer> observers;

/*

* Constructs a NotificationSystem object with an empty list of courses and an empty list of observers.

*/

public NotificationSystem() {

this.courses = new ArrayList<>();
```

```
this.observers = new ArrayList<>();

}

/*

* Adds an observer to the list of observers

* @param observer an Observer object to be added to the list of observers

*/

public void addObserver(Observer observer) {

this.observers.add(observer);

}

/*

* Removes an observer from the list of observers

* @param observer an Observer object to be removed from the list of observers

*/

public void removeObserver(Observer observer) {

this.observers.remove(observer);

}

/*

* Adds a new course to the list of courses and notifies all observers

* @param course a Course object to be added to the list of courses

*/

public void addCourse(Course course) {

this.courses.add(course);

notifyObservers(course);
```

```
}

/*

* Notifies all observers that a new course has been added to the list of courses

* @param course a Course object that has been added to the list of courses

*/

void notifyObservers(Course course) {

System.out.println("New course available! New course added: " + course.getName());

for (Observer observer : observers) {

observer.update();

}

}

/*

* Returns the list of courses

* @return the list of courses

*/

public List<Course> getCourses() {

return this.courses;

}

}
```

We can register the user as the observer of the NotificationSystem using the addObserver() method. Then, we can add a new course to the system using the addCourse() method, which will notify all registered observers of the new course.

Project Summary

In summary, this project provides a basic structure of a meditation application that allows users to access courses, sessions, exercises, instructor information related to the meditation. The application allows users to select modules that contains multiple courses, and each course contains multiple sessions that focus on specific exercise. This project implements SOLID principles and ensure loose coupling between objects. The Driver class is the main class that runs the application and manages objects such as instructors, meditation exercises, sessions, courses, modules, users, and payment and subscription information. The Driver class interacts with users by displaying information and prompting for input via the console. The application also implements the Observer pattern and Notification system, where users can register as observers to receive updates about new courses being added. Overall, this application provides a structured approach to learning and practicing creating a program while implementing SOLID principles for maintainability.

This is the complete code for the main function:
It first creates and manages various object such as instructors, meditation exercises, sessions, courses, modules, users, and payment. Then it interacts with users by displaying information and prompting for input via the console to let user select a module. It displays the selected module information including course information, session, meditation exercise information, instructor, duration, etc. in the console. It asks the user to record their emotional status after the module, and the states will be updated in user's meditation history. It then showcases the notification system by registering and unregistering users as observers and notifying them of updates to courses. Finally, it tests the Model-View-Controller, and display the information about sessions and exercises.

```
/**

* This Driver class is the main class that runs the meditation application

* It creates and manages various objects such as instructors, meditation exercises,

* sessions, courses, modules, users, and payment and subscription information.

* It also interacts with users by displaying information and prompting for input via the console.

*/

public class Driver {

public static void main(String[] args) {

// Create instructors

Instructor instructor1 = new Instructor("John", "John is an experienced meditation instructor.");
```

```
Instructor instructor2 = new Instructor("Jane", "Jane is a theropist.");

// Create meditation exercises

Meditation meditation1 = new Meditation ("Breathing Meditation", "Focus on your breath.", 10,
instructor1);

Meditation meditation2 = new Meditation ("Body Wakeup", "Wake your body up for tension.", 15,
instructor1);

Meditation meditation3 = new Meditation ("Loving Kindness", "Send love and kindness to yourself.",
20, instructor2);

// Create sessions

List<Exercise> sessionExercises1 = new ArrayList<>();

sessionExercises1.add(meditation1);

sessionExercises1.add(meditation2);

Session session1 = new Session("Session 1", "Introduction to meditation techniques", 25,
sessionExercises1);

List<Exercise> sessionExercises2 = new ArrayList<>();

sessionExercises2.add(meditation3);

Session session2 = new Session("Session 2", "Advanced meditation techniques", 20,
sessionExercises2);

// Create courses

List<Session> courseSessions1 = new ArrayList<>();

courseSessions1.add(session1);

Course course1 = new Course("Introduction to Meditation", "Learn the basics of meditation.",
courseSessions1, instructor1,3, true);

List<Session> courseSessions2 = new ArrayList<>();

courseSessions2.add(session2);

Course course2 = new Course("Advanced Meditation", "Deepen your meditation practice.",
courseSessions2, instructor2,5,false);
```

```
// Create modules

List<Course> moduleCourses1 = new ArrayList<>();

moduleCourses1.add(course1);

Module module1 = new Module("Meditation 101", "Beginner's Guide to Meditation",
moduleCourses1);

List<Course> moduleCourses2 = new ArrayList<>();

moduleCourses2.add(course2);

Module module2 = new Module("Meditation 201", "Advanced Meditation Techniques",
moduleCourses2);

// Display module information

module1.printModule(module1);

module2.printModule(module2);

// User selects a module

List<Module> modules = new ArrayList<>();

modules.add(module1);

modules.add(module2);

Scanner scanner = new Scanner(System.in);

System.out.print("Enter the number of the module you want to select: ");

int selectedModuleIndex = scanner.nextInt() - 1;

Module selectedModule = modules.get(selectedModuleIndex);

System.out.println("You selected: " + selectedModule.getName());

System.out.println("Description: " + selectedModule.getDescription());

// Display course information

for (Course course : selectedModule.getCourses()) {
```

```
System.out.println("\nCourse: " + course.getName());

System.out.println("Description: " + course.getDescription());

System.out.println("Instructor: " + course.getInstructor().getName());

System.out.println("Bio: " + course.getInstructor().getBio());

System.out.println("Credits: " + course.getCredits());

// Get required courses for the module based on a certain criterion

List<Course> requiredCourses = module1.getRequiredCourses(c -> c.isRequired());

// Display required courses

System.out.println("\nRequired courses:");

for (Course reqCourse : requiredCourses) {

System.out.println(reqCourse.getName());

}

// Display session information

for (Session session : course.getSessions()) {

System.out.println("\nSession: " + session.getName());

System.out.println("Description: " + session.getDescription());

System.out.println("Total Duration: " + session.getDuration() + " minutes");

// Display exercise information

System.out.println("\nExercises:");

for (Exercise exercise : session.getExercises()) {

System.out.println("Name: " + exercise.getName());

System.out.println("Description: " + exercise.getDescription());

System.out.println("Duration: " + exercise.getDuration() + " minutes");
```

```
System.out.println("Instructor: " + ((Meditation)exercise).getInstructor().getName());

}

}

}

// Create users

User user1 = new User("Alice", "alice@example.com", "password");

User user2 = new User("Bob", "bob@example.com", "password");



// Prompt the user to enter their emotional state after the session

Scanner scanner2 = new Scanner(System.in);

System.out.print("Please enter your emotional state (positive/negative/neutral): ");

String emotionalState = scanner2.nextLine();

// Validate the user input

while (!emotionalState.equalsIgnoreCase("positive")
&& !emotionalState.equalsIgnoreCase("negative") && !emotionalState.equalsIgnoreCase("neutral")) {

System.out.print("Invalid emotional state. Please enter a valid emotional state
(positive/negative/neutral): ");

emotionalState = scanner2.nextLine();

}

// Update user's meditation history

LocalDateTime startTime = LocalDateTime.now();

int duration = 10;

user1.updateMeditationHistory(meditation1, startTime, duration,emotionalState);

// Update user's session history
```

```
LocalDateTime endTime = LocalDateTime.now().plusMinutes(30);

user1.updateSessionHistory(session1, startTime, endTime);

// Display user's meditation and session history

System.out.println("User's meditation history:");

for (MeditationRecord record : user1.getMeditationHistory()) {

System.out.println(record.getMeditation().getName() + " " + record.getStartTime() + " " +
record.getDuration()+ " Emotion status:" + record.getEmotionalState());

}

System.out.println("\nUser's session history:");

for (SessionRecord record : user1.getSessionHistory()) {

System.out.println(record.getSession().getName() + " " + record.getStartTime() + " " +
record.getEndTime());

}

// Create a payment

Payment payment = new Payment(99.99, new Date(), "Credit card");

// Create a subscription

Subscription subscription = new Subscription(new Date(), new Date(System.currentTimeMillis() +
30L * 24L * 60L * 60L * 1000L), "Premium");

// Print payment and subscription information

System.out.println("Payment amount: " + payment.getAmount());

System.out.println("Payment date: " + payment.getDate());

System.out.println("Payment method: " + payment.getPaymentMethod());

System.out.println("Subscription start date: " + subscription.getStartDate());

System.out.println("Subscription end date: " + subscription.getEndDate());

System.out.println("Subscription level: " + subscription.getSubscriptionLevel());
```

```
// Test MeditationRecord, SessionRecord, and ExerciseRecordList classes:

// Create a new meditation record

MeditationRecord meditationRecord1 = new MeditationRecord(startTime, duration, meditation1);

MeditationRecord meditationRecord2 = new MeditationRecord(startTime, duration, meditation2);

MeditationRecord meditationRecord3 = new MeditationRecord(startTime, duration, meditation3);

// Create a new meditation record list and add the records

MeditationRecordList meditationRecordList = new MeditationRecordList();

meditationRecordList.add(meditationRecord1);

meditationRecordList.add(meditationRecord2);

meditationRecordList.add(meditationRecord3);

// Print the details of the meditation record

System.out.println("Meditation Record list:");

for (MeditationRecord record : meditationRecordList) {

System.out.println("Meditation" + record.getName() + record.getStartTime() + " - " +
record.getDuration() + " minutes");

}

// Sort the list by ascending order of duration

System.out.println("\nMeditation Record list by ascending order of duration:");

meditationRecordList.sort();

for (MeditationRecord record : meditationRecordList) {

System.out.println(record.getStartTime() + " - " + record.getDuration() + " minutes");

}

// Create a new session record

LocalDateTime sessionStartTime = LocalDateTime.now();
```

```
LocalDateTime sessionEndTime = sessionStartTime.plusMinutes(30);

SessionRecord sessionRecord = new SessionRecord(sessionStartTime, sessionEndTime, session1);

// Print the details of the session record

System.out.println("Session Record:");

System.out.println("Start Time: " + sessionRecord.getStartTime());

System.out.println("End Time: " + sessionRecord.getEndTime());

System.out.println("Duration: " + sessionRecord.getDuration() + " seconds");

System.out.println("Session: " + sessionRecord.getSession().getName());

// Create a notification system

System.out.println("\nSend notification");

System.out.println("-----------------------------------\n");

NotificationSystem notificationSystem = new NotificationSystem();

// Register users as observers

notificationSystem.addObserver(user1);

notificationSystem.addObserver(user2);

// Create a notification

Course course3 = new Course("Meditation for Stress Reduction", "Learn meditation techniques to
reduce stress.", new ArrayList<Session>(), new Instructor("Jane Smith", "Jane is a renowned
meditation expert."), 5, false);

// Add another course to the notification system

notificationSystem.addCourse(course3);

// Unregister one of the users as an observer of the notification system

notificationSystem.removeObserver(user1);

// Update one of the courses
```

```
course1.setName("Programming Fundamentals");

// Notify all the registered observers of the notification system

notificationSystem.notifyObservers(course1);



// Create a user view and controller

UserView userView = new UserView(user1);

// Set controller for the user and view

UserController userController = new UserController(user1, userView);

// Set controller for the view

userView.setController(userController);

//Create a new JFrame and add the UserView object to its content panel

JFrame frame = new JFrame("User Information");

frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

frame.getContentPane().add(userView);

frame.pack();

frame.setVisible(true);

//display the user information

userController.updateView();

// Update the user data

userController.setUserEmail("newemail@example.com");

userController.setUserName("new name");

// Display the updated user information

userController.updateView();
```

```
}

}
```

By running this program, the console should display:

```
Module: Meditation 101

Course: Introduction to Meditation

Session: Session 1

Module: Meditation 201

Course: Advanced Meditation

Session: Session 2

Enter the number of the module you want to select: 1

You selected: Meditation 101

Description: Beginner's Guide to Meditation

Course: Introduction to Meditation

Description: Learn the basics of meditation.

Instructor: John

Bio: John is an experienced meditation instructor.

Credits: 3

Required courses:

Introduction to Meditation

Session: Session 1

Description: Introduction to meditation techniques

Total Duration: 25 minutes

Exercises:

Name: Breathing Meditation
```

Description: Focus on your breath.

Duration: 10 minutes

Instructor: John

Name: Body Wakeup

Description: Wake your body up for tension.

Duration: 15 minutes

Instructor: John

Please enter your emotional state (positive/negative/neutral): positive

User's meditation history:

Breathing Meditation 2023-04-25T16:33:19.591554 10 Emotion status:positive

User's session history:

Session 1 2023-04-25T16:33:19.591554 2023-04-25T17:03:19.593232

Payment amount: 99.99

Payment date: Tue Apr 25 16:33:19 EDT 2023

Payment method: Credit card

Subscription start date: Tue Apr 25 16:33:19 EDT 2023

Subscription end date: Thu May 25 16:33:19 EDT 2023

Subscription level: Premium

Meditation Record list:

MeditationBreathing Meditation2023-04-25T16:33:19.591554 - 10 minutes

MeditationBody Wakeup2023-04-25T16:33:19.591554 - 15 minutes

MeditationLoving Kindness2023-04-25T16:33:19.591554 - 20 minutes

Meditation Record list by ascending order of duration:

2023-04-25T16:33:19.591554 - 10 minutes

2023-04-25T16:33:19.591554 - 15 minutes

2023-04-25T16:33:19.591554 - 20 minutes

Session Record:

Start Time: 2023-04-25T16:33:19.604130

End Time: 2023-04-25T17:03:19.604130

Duration: 1800 seconds

Session: Session 1

Send notification

New course available! New course added: Meditation for Stress Reduction

User Alice has been notified

User Bob has been notified

New course available! New course added: Programming Fundamentals

User Bob has been notified

User Information

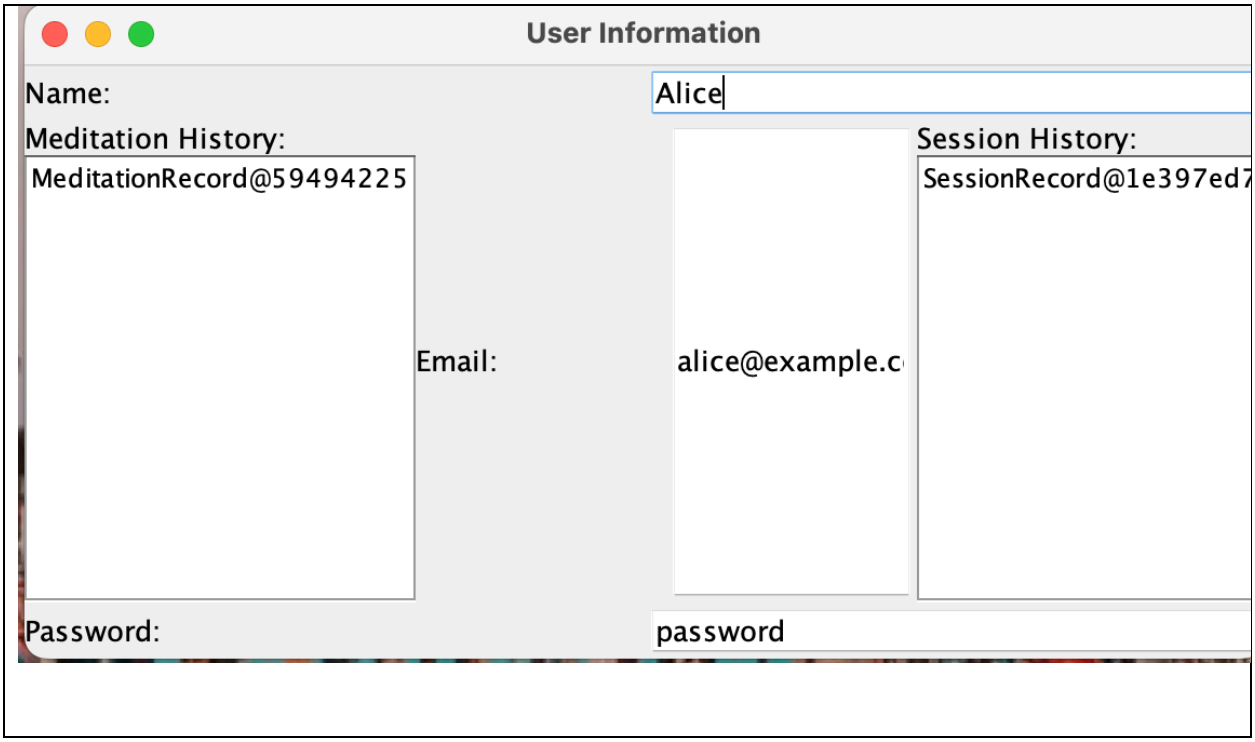
Name: Alice

Email: alice@example.com

User Information

Name: new name

Email: newemail@example.com


User Information
Name:
Alice
Meditation History:
MeditationRecord@59494225
Email:
alice@example.c
Session History:
SessionRecord@1e397ed7
Password:
password

Rubric

Concept 1 : Recursion in Practice	8 Points
Concept 2 : Abstract Classes and Interfaces	10 Points
Concept 3 : Abstracted Linked Lists	10 Points
Concept 4 : Higher order functions map, filter, and fold	8 Points
Concept 5 : Hierarchical Data Representation	10 Points
Concept 6 : MVC Design	10 Points
Concept 7 : SOLID Design Principles	10 Points
Concept 8 : Any design pattern	8 Points
Overall Quality	10 Points
Total Possible Points Out of 100	**84 Points**
Possible adjustments	
Extension	+10 Points
Bonus Points for creativity and going above and beyond	+10 Points
Work originality	-100 Points
Code quality	-50 Points
Video code walkthrough (if applicable)	-100 Points