Nathan Pattison, Cameron Rachman

object oriented analysis and design Project

Table of Contents

**Part A2**

Use Case Diagram2

Domain Model3

Class Diagram3

Sequence Diagrams4

**Part B5**

Q15

Observer design pattern5

Strategy design pattern6

Q28

**Bibliography11**

**Logs12**

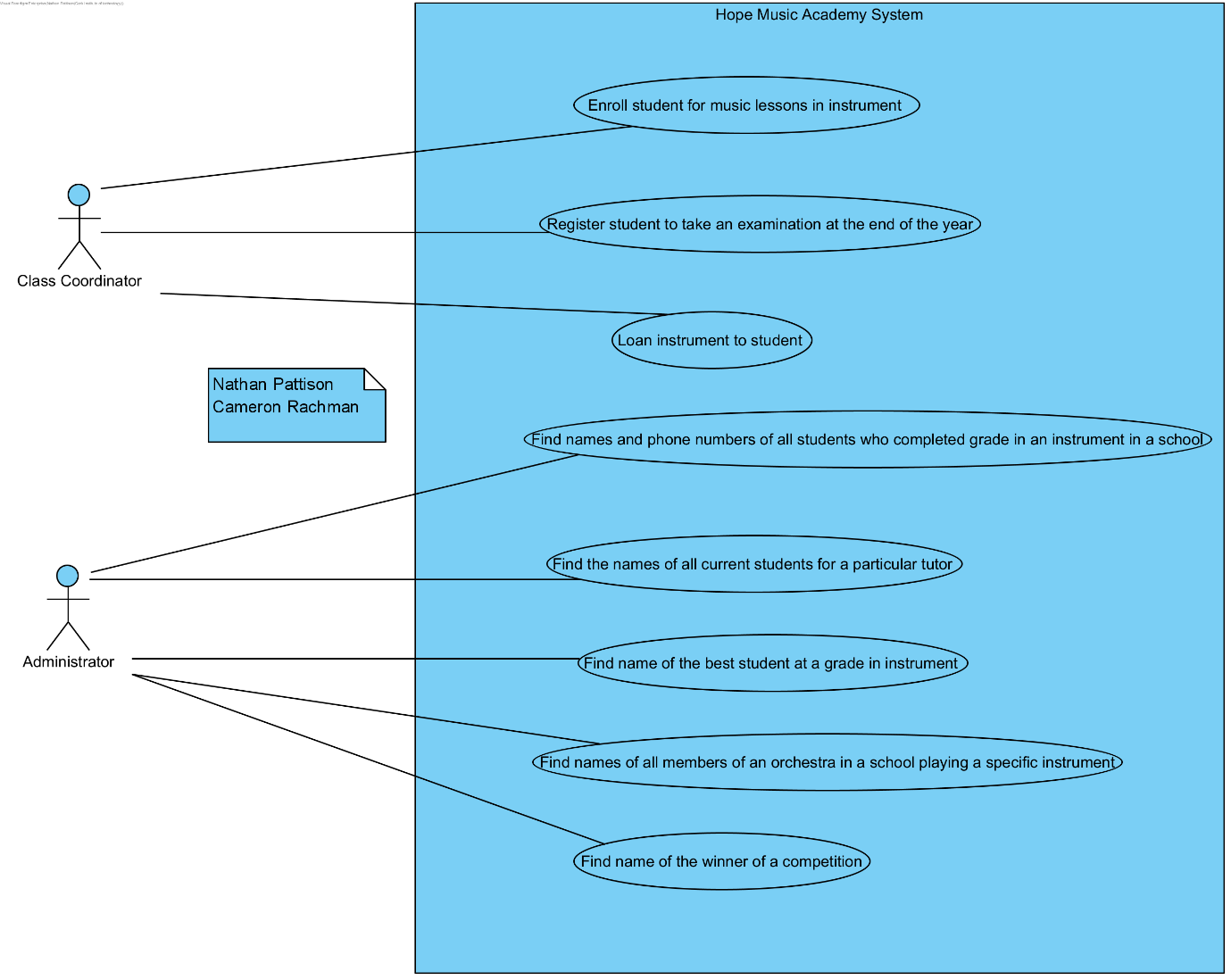
Nathan Pattison12

Cameron Rachman13

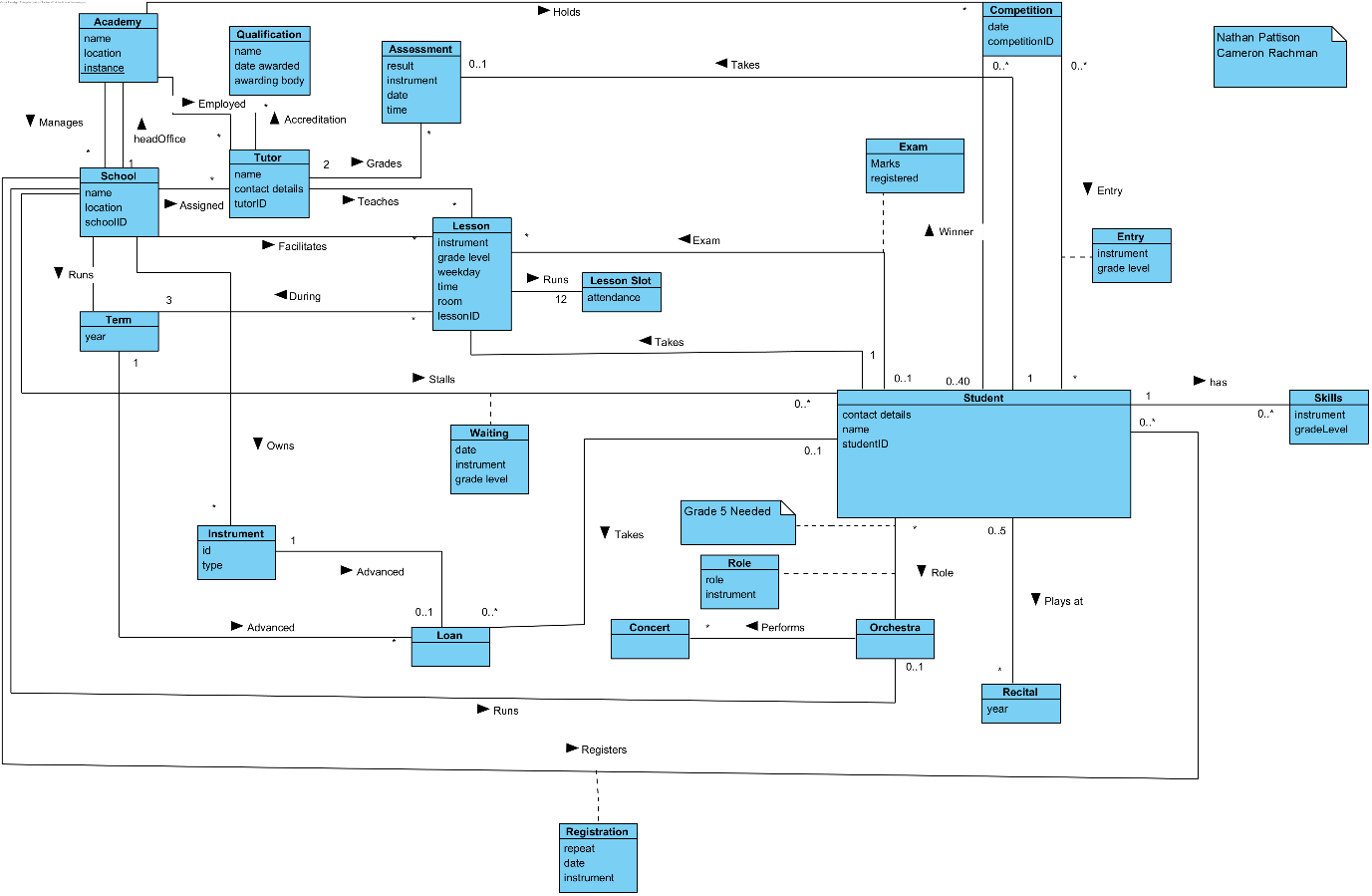
**OOA Final Project**

Part A

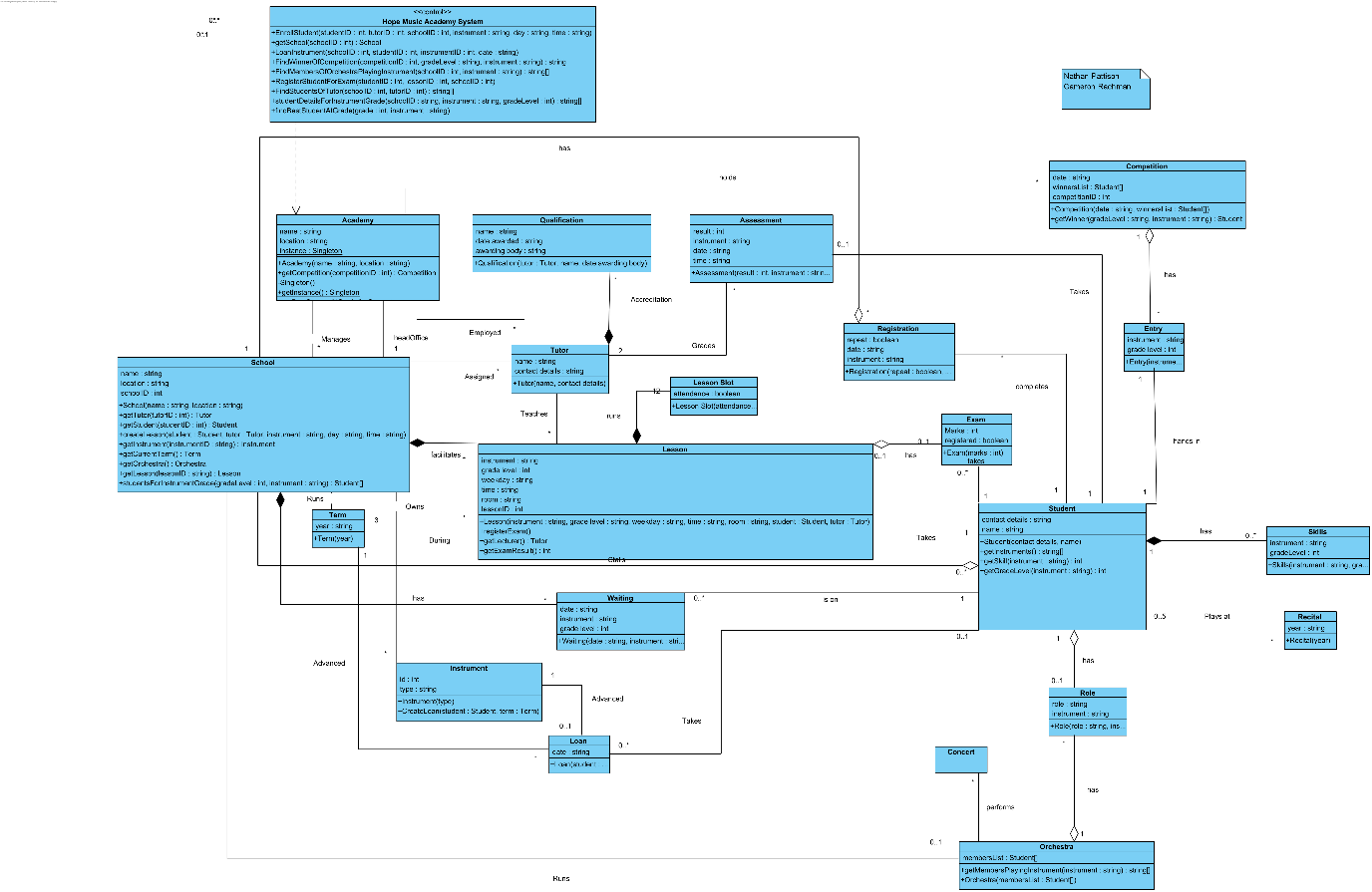
Use Case Diagram



Domain Model

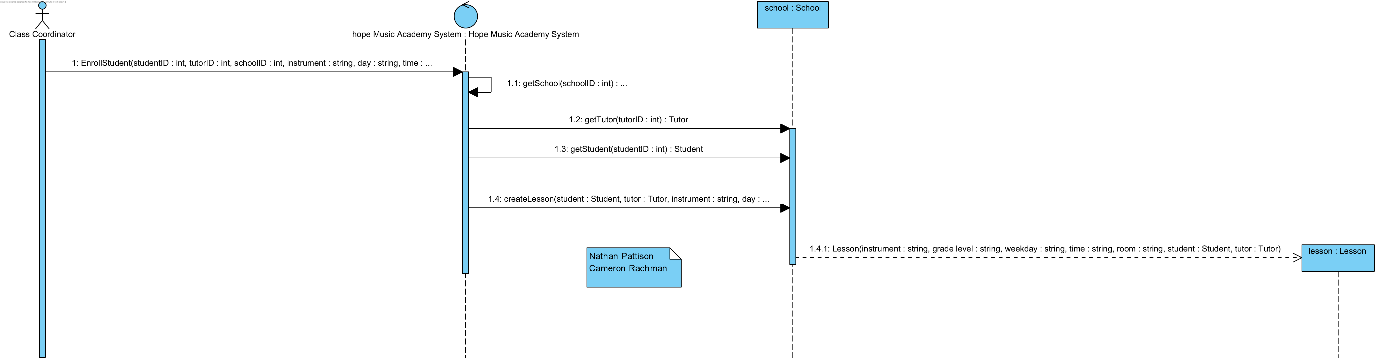


Class Diagram

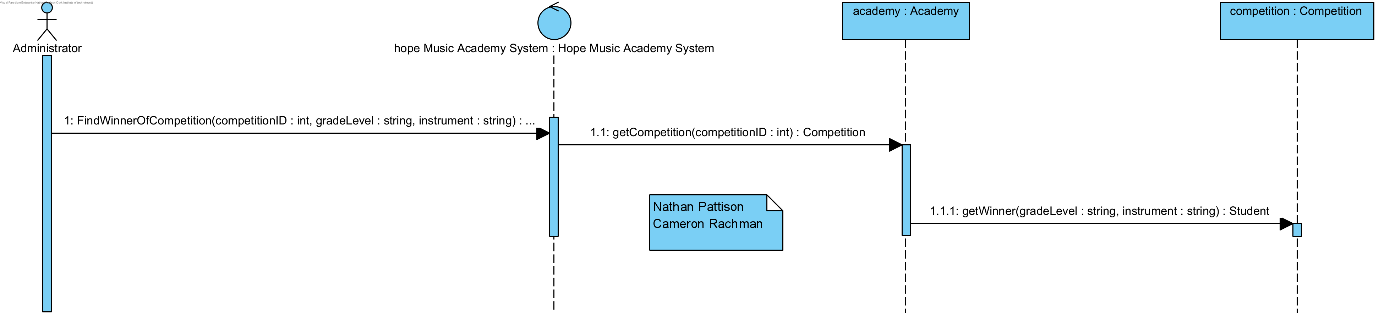


Sequence Diagrams

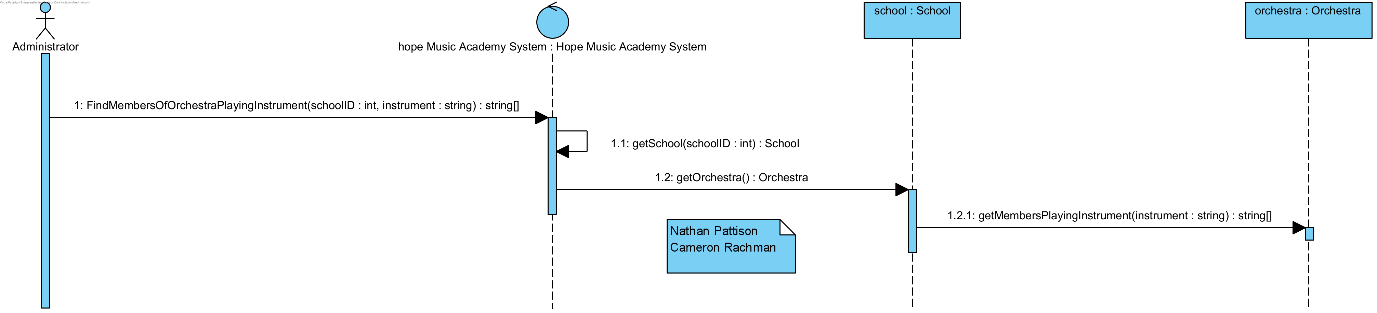
Enrol student for music lessons in instrument



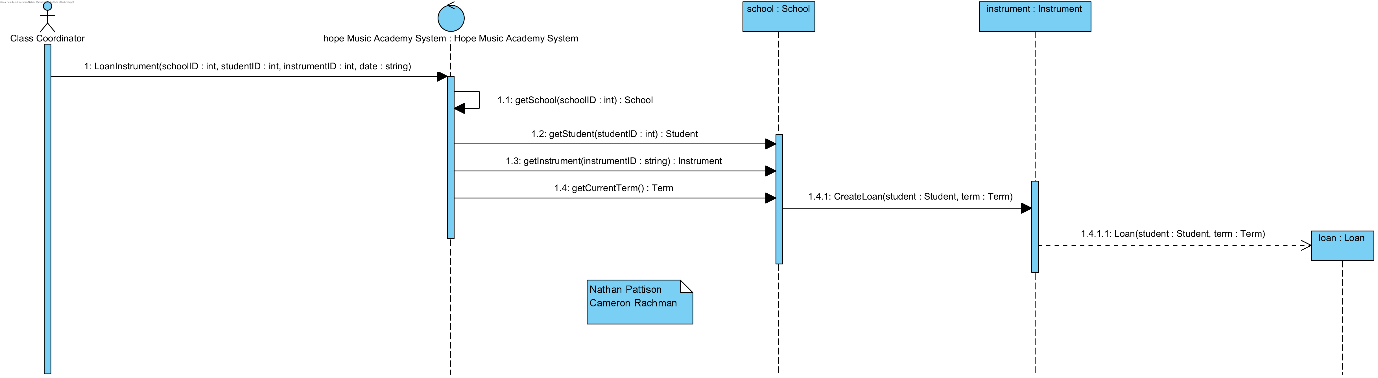
Find name of the winner of a competition



Find names of all members of an orchestra in a school playing a specific instrument



Loan instrument to student



Part B

Q1

**Observer design pattern**

i)

The purpose of the Observer design pattern is to define a one-to-many dependency between objects so that when one object (the subject) changes its state, all dependent objects (observers) are automatically notified and updated. This allows for a loose coupling between the subject and its observers, promoting modularity and flexibility in the system design. (w3sDesign, 2018)(pg237)

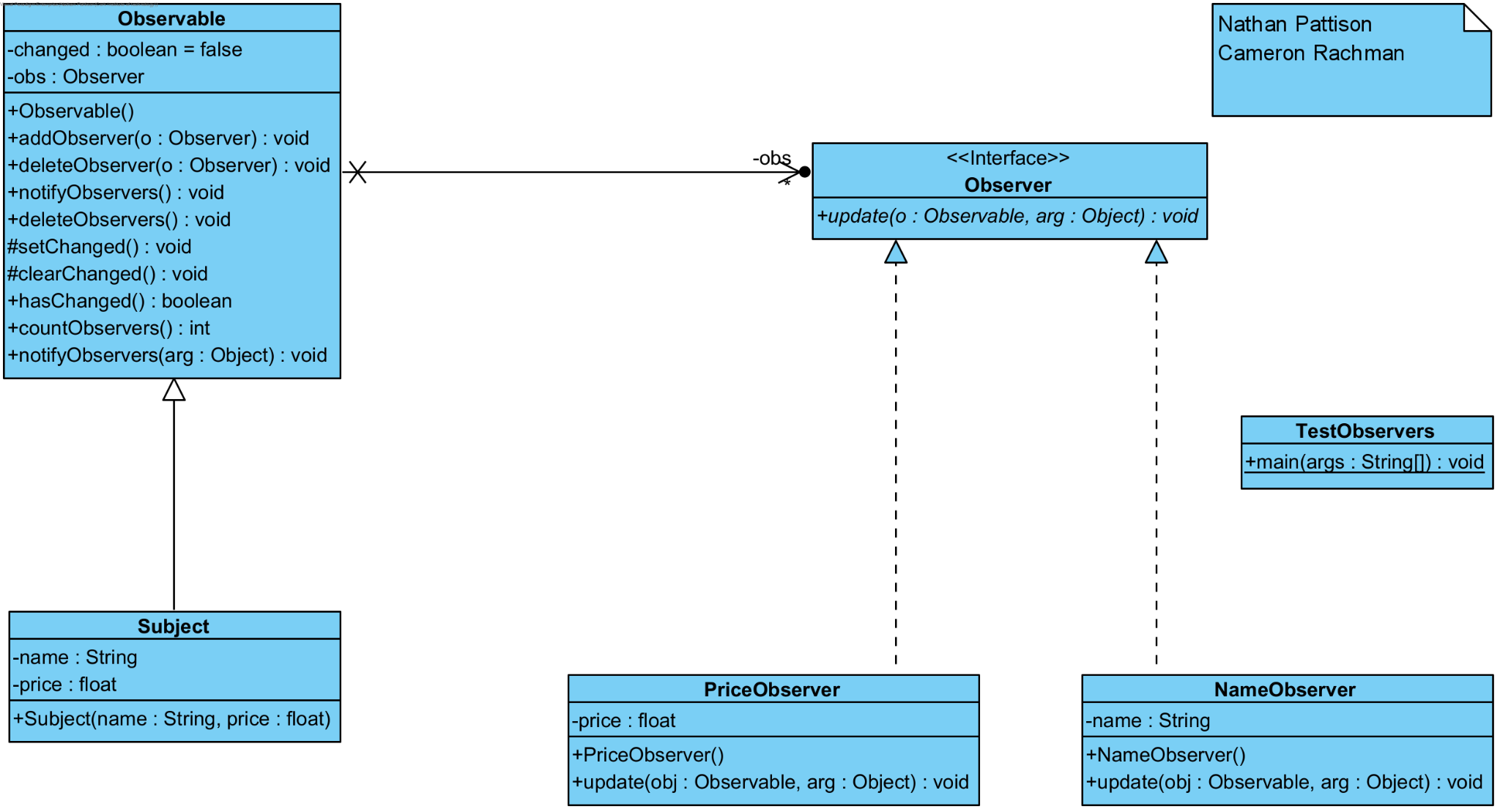
ii)

The Observer pattern solves problems associated with tight coupling between objects, such as increased complexity, difficulty in maintaining and updating code, and reduced reusability. By decoupling the subject and its observers, this pattern allows for easier modification and extension of the codebase. (w3sDesign, 2018)(pg237)

iii)

1. “Subject” class extends the “java.util.Observable” class, which provides the core functionality of the Observer pattern. The “Subject” class has two properties, “name” and “price”, and corresponding setter methods that update these properties.
2. When the “name” or “price” is updated through the setter methods, the “setChanged()” method from the “Observable” class is called to mark the state as changed. Then, the “notifyObservers()” method is called to notify all registered observers of the change.
3. Two observer classes, “NameObserver” and “PriceObserver”, implement the “java.util.Observer” interface, which requires them to implement an “update()” method. This method is called automatically by the subject when the state changes.
4. In the “TestObservers” class, a “Subject” object is created and instances of both “NameObserver” and “PriceObserver” are registered as observers. When the subject's state changes, both observers are notified and their “update()” methods are called.

iv)



**Strategy Design Pattern**

i.

The purpose of the Strategy Design Pattern is to provide a way to define a family of algorithms, encapsulate each one of them, and make them interchangeable. This pattern allows the behaviour of an object to be selected at runtime, depending on the strategy chosen, which enables the client code to vary the algorithm used without modifying the object. (w3sDesign, 2018)(pg268)

ii.

The Strategy Design Pattern addresses the following problems:

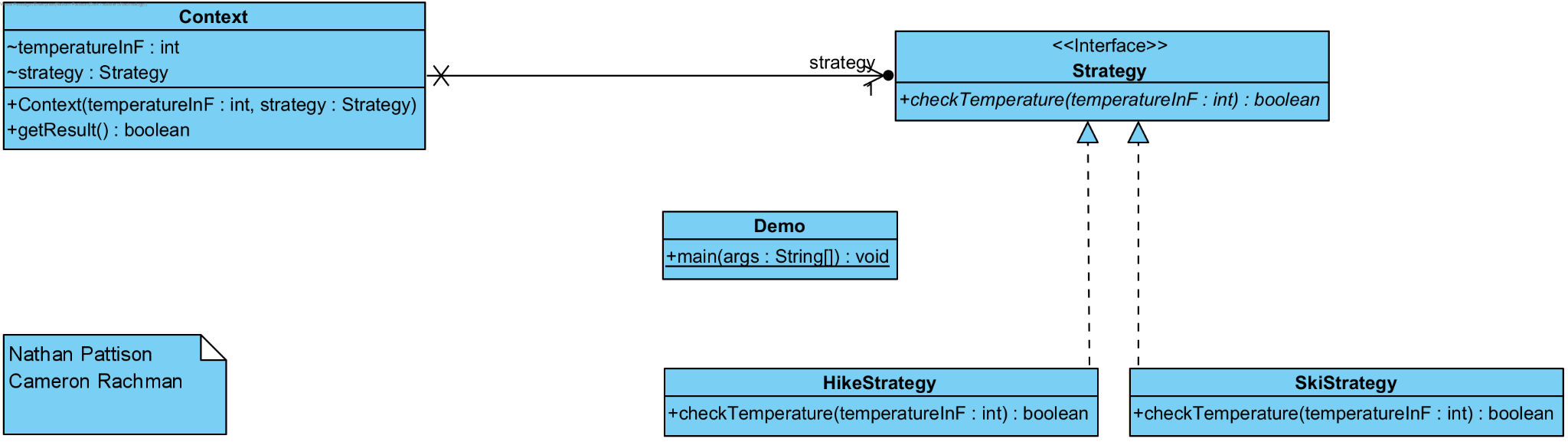
1. The need for multiple conditional statements to select appropriate behaviour at runtime.
2. The need to support the Open/Closed Principle, which states that a class should be open for extension but closed for modification.
3. The need for code reusability and maintainability when dealing with various algorithms that have similar functionalities but different implementations.

iii.

In the provided code, the Strategy Design Pattern is applied as follows:

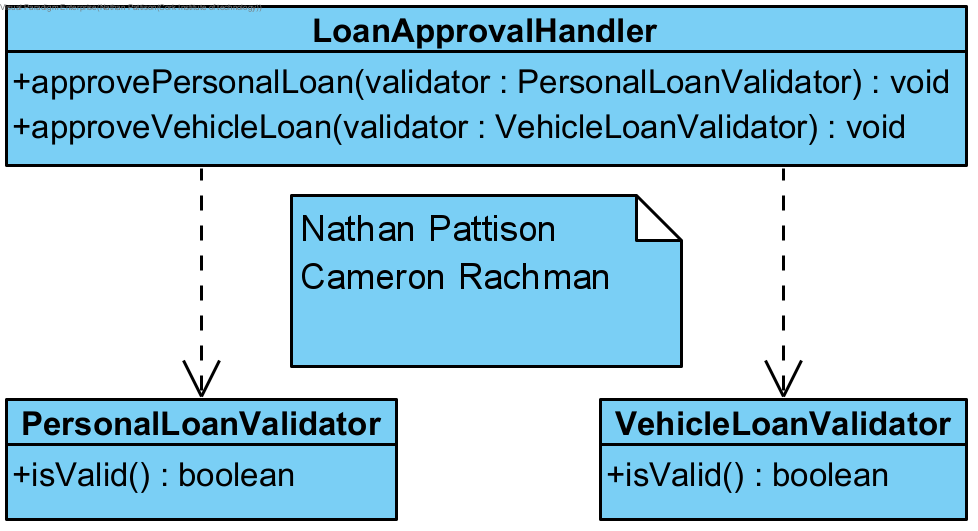
1. The “Strategy” interface is defined with a single method “checkTemperature(int temperatureInF)”, which is the common method signature for all supported strategies.
2. Two concrete strategy classes are implemented: “HikeStrategy” and “SkiStrategy”. Both implement the “Strategy” interface and provide their own implementation for the “checkTemperature” method.
3. The “Context” class is used to configure and maintain a reference to a specific strategy object. It has a method “getResult()” that delegates the temperature check to the chosen strategy's “checkTemperature” method.
4. The “Demo” class demonstrates how to use the pattern. It creates a “Context” object with an initial strategy (in this case, “SkiStrategy”). Then it changes the strategy to “HikeStrategy” using the “setStrategy()” method of the “Context” object. In both cases, the “getResult()” method is called to check if the temperature is appropriate for the selected activity.

iv)



Question 2

i.



ii.

The open-closed principle states that software entities (classes, modules, functions, etc.) should be open for extension but closed for modification. The current “LoanApprovalHandler” class violates this principle because every time a new type of loan needs to be added (like the holiday loan), you have to modify the class by adding a new method to handle the approval process for that loan type. This makes the class less maintainable and less flexible in the long run. (Meyer, 1988)

iii.

To fix the violation of the open-closed principle, we can introduce an interface “LoanValidator” that has the “isValid()” method. Then, we can modify the “LoanApprovalHandler” class to have a single method “approveLoan()” that takes a “LoanValidator” object as a parameter. This way, we can add new types of loans without modifying the “LoanApprovalHandler” class.

java

public interface LoanValidator {

boolean isValid();

}

public class PersonalLoanValidator implements LoanValidator {

@Override

public boolean isValid() {

// Validation logic

}

}

public class VehicleLoanValidator implements LoanValidator {

@Override

public boolean isValid() {

// Validation logic

}

}

public class LoanApprovalHandler {

public void approveLoan(LoanValidator validator) {

if (validator.isValid()) {

// Process the loan.

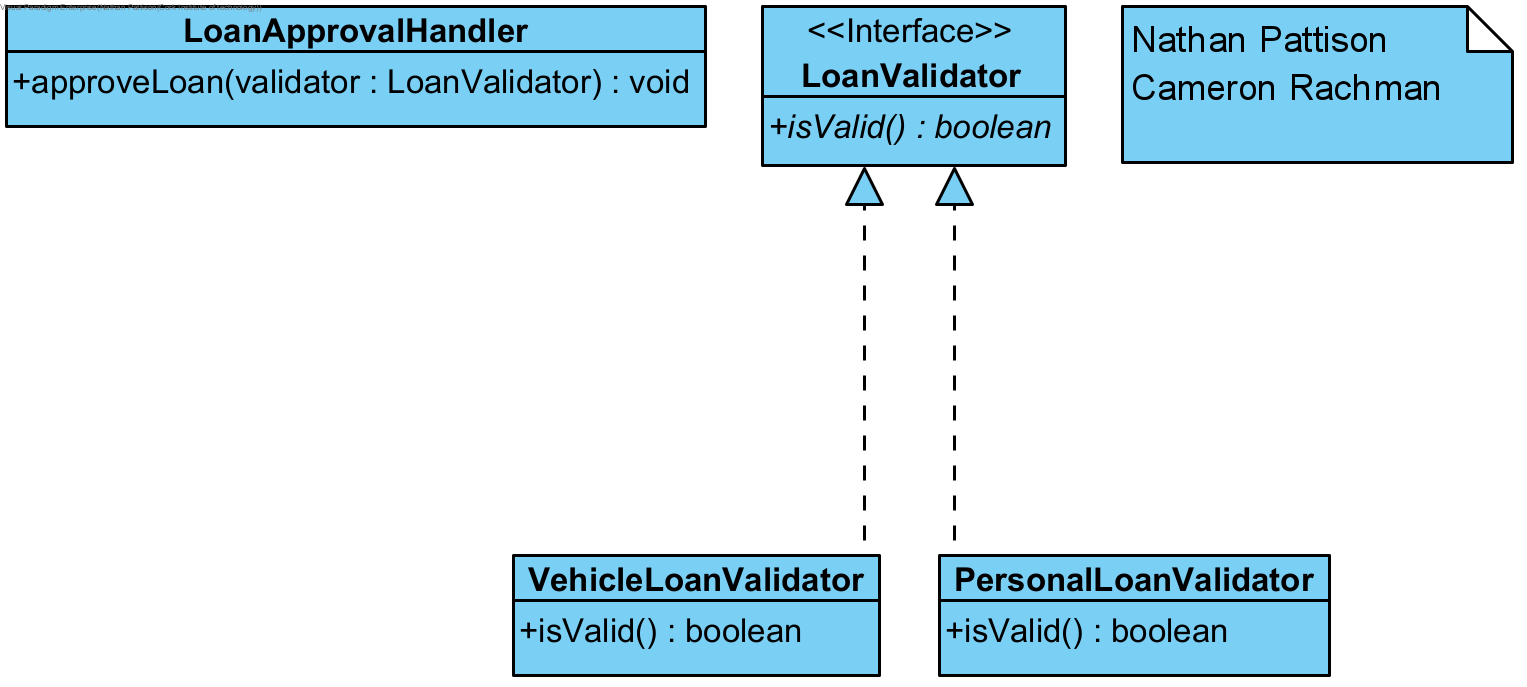
}

}

}

Now, when a new type of loan needs to be added, you can simply create a new class implementing the “LoanValidator” interface and pass it to the “approveLoan()” method without modifying the “LoanApprovalHandler” class. This adheres to the open-closed principle, making the code more maintainable and flexible.

iv.



# Bibliography

Meyer, B. (1988). *Object-Oriented Software Construction.* Prentice Hall.

w3sDesign. (2018, December 1). *GoF\_Design\_Patterns\_Reference.* Retrieved from http://w3sdesign.com: http://w3sdesign.com/GoF\_Design\_Patterns\_Reference0100.pdf

**Logs**

Nathan Pattison

|  |  |  |  |
| --- | --- | --- | --- |
| Date | Activity Description | Start time | End Time |
| 21 Mar 23 | Setting up workspace | 4:40pm | 4:47pm |
| 21 Mar 23 | Reviewing project brief | 4:47pm | 4:54pm |
| 21 Mar 23 | Breaking down project brief for domain model | 4:54pm | 5:55pm |
| 22 Mar 23 | Breaking down project brief for domain model | 12:35pm | 12:58pm |
| 22 Mar 23 | Domain Model Sketching | 2:15pm | 2:40pm |
| 22 Mar 23 | Breaking down project brief for domain model | 2:40pm | 3:05pm |
| 22 Mar 23 | Creating draft domain model in VP from sketch | 3:05pm | 3:43pm |
| 22 Mar 23 | Working on Observer Pattern Answers | 3:43pm | 3:55pm |
| 23 Mar 23 | Working on Observer Pattern Answers | 5:40pm | 5:51pm |
| 27 Mar 23 | Making VP Domain Model v2 from sketch | 1:32pm | 1:50pm |
| 27 Mar 23 | Making VP Domain Model v2 from sketch | 5:39pm | 6:07pm |
| 28 Mar 23 | Refining VP Domain Model | 11:47am | 12:59pm |
| 28 Mar 23 | Refining VP Domain Model | 1:15pm | 1:55pm |
| 28 Mar 23 | Reverse Engineering Observer Design Pattern code | 2:26pm | 2:43pm |
| 28 Mar 23 | Refining VP Domain Model | 4:10pm | 5:59pm |
| 6 Apr 23 | Voice call to divide work | 5:25pm | 5:47pm |
| 8 Apr 23 | Adjusting Domain Model | 4:42pm | 6pm |
| 13 Apr 23 | Adjusting Domain Model | 5:11pm | 6:04pm |
| 17 Apr 23 | Working on Class Operations/Diagram | 9:01am | 11:50am |
| 17 Apr 23 | Working on Class Operations/Diagram/Sequence | 1:30pm | 3:30pm |
| 17 Apr 23 | Working on reverse engineering for part b | 3:40pm | 5:50pm |
| 17 Apr 23 | Working on reverse engineering for part b | 7:25pm | 9:40pm |
| 18 Apr 23 | Working on VP Domain Model | 8:40am | 9:09am |
| 18 Apr 23 | Working on reverse engineering for part b | 10:15am | 10:22am |
| 18 Apr 23 | Working on reverse engineering for part b | 12:14pm | 12:30pm |
| 18 Apr 23 | Making Class Diagram | 1:15pm | 2pm |
| 18 Apr 23 | Making notes from Mary and Plans for future work | 4:40pm | 5:50pm |
| 21 Apr 23 | Working on reverse engineering for part b | 11:30am | 12:55pm |
| 21 Apr 23 | Working on domain model | 1:35pm | 2:16pm |
| 24 Apr 23 | Sequence Diagrams | 9:38am | 10:48 |
| 24 Apr 23 | Class diagram | 11:30am | 12:20pm |
| 24 Apr 23 | Sequence Diagrams | 12:20pm | 2pm |
| 25 Apr 23 | Part B | 9:33am | 10am |
| 25 Apr 23 | Working on report document/making corrections in vp | 10am | 12:47pm |
| 25 Apr 23 | Working on report document/making corrections in vp | 1:08pm | 3:22pm |

Cameron Rachman

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Activity** | **Date** | **Start - End Time** |
| **1** | Setting up workspace | 21/March/23 | 4:40pm - 4:45pm |
| **2** | Reviewing project brief | 21/March/23 | 4:45pm - 5:00pm |
| **3** | Break down domain model & Sketching | 21/March/23 | 4:50pm - 6:00pm |
| **4** | Reviewing part B | 26/March/23 | 9:30pm - 10:00pm |
| **5** | Refining domain model | 26/March/23 | 10:00pm - 11:05pm |
| **6** | Sketch class diagrams for part B iv | 27/March/23 | 7:30pm - 8:30pm |
| **7** | Refining Domain model | 28/March/23 | 12:00am - 2:00pm |
| **8** | Reviewing Work and Brief | 06/April/23 | 5:30pm - 6:00pm |
| **9** | Part B Q3 Draft | 07/April/23 | 1:30pm - 2:30pm |
| **10** | Reviewing Class Diagram | 14/April/23 | 3:30pm - 4:00pm |
| **11** | Moving Class Diagram to VP | 16/April/23 | 2:30pm - 2:50pm |

|  |  |  |  |
| --- | --- | --- | --- |
| **12** | Refining Class Diagram | 16/April/23 | 2:50pm - 3:50pm |
| **13** | Reviewing class diagram | 17/April/23 | 9:00am - 12:00pm |
| **14** | Reviewing operations | 17/April/23 | 1:00pm - 3:30pm |
| **15** | Reviewing finished work | 24/April/23 | 9:30am - 11:00am |
| **16** | Sketching Sequence Diagrams | 24/April/23 | 11:00am - 1:00pm |
| **17** | Modifying domain model | 24/April/23 | 1:00pm - 1:35pm |
| **18** | Modifying class diagram | 24/April/23 | 1:35pm - 2:30pm |
| **19** | Creating Sequence Diagrams | 25/April/23 | 10:30am - 12:00pm |
| **20**  **21** | Polishing and refining class diagram  Finalising Project | 25/April/23  25/April/23 | 12:00pm - 2:30pm  2:30pm - 3:45pm |