

FLASHCARD APPLICATION

Project 9

By

Group 9 & 29

NAME	BITS ID	EMAIL-ID
Abhinav Mishra	2021A7PS0706P	f20210706@pilani.bits-pilani.ac.in
Avyakt Garg	2020B1A71902P	f20201902@pilani.bits-pilani.ac.in
Raghav Luthra	2019B4A80639P	f20190639@pilani.bits-pilani.ac.in
Rohit Raj	2020B3A70906P	f20200906@pilani.bits-pilani.ac.in
Siddharth S. Shah	2021A7PS2428P	f20212428@pilani.bits-pilani.ac.in

For the course

Object Oriented Programming (CS F213)

December 2022

ANTI-PLAGIARISM STATEMENT

We hereby declare that the submission made is completely made by us and not plagiarized from online or other sources in part or completely. We acknowledge that lecture notes, java docs, and other references were used during the completion of the project. We also certify that this project was not submitted earlier as part of any other course or other submissions and is our own original creation.



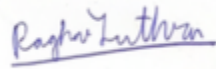
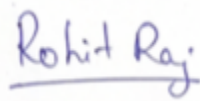

Signatures :-



The image displays five handwritten signatures in blue ink, arranged in two rows. The top row contains four signatures: 'Abhi.', 'Aarg', 'Raghu Luthran', and 'Rohit Raj'. The bottom row contains a single, larger signature that appears to be 'Siddharth'.

Contribution Table

The project was a collaborative effort and all members of the group except one contributed equally which made the project successful. Each and every one helped the other while doing the process, All were involved in the initial ideation and final implementation

Name	Contribution	Signature
Abhinav Mishra	Fibcard,mcq,user, user class, sequential patterns, solid principles, revisedeck, sequential diagram, researched on file handling for future versions and made a prototype,	
Avyakt Garg	Card,cardtype,user user class, researched on design patterns, UML, revise card, create def cardController, create fibcardController	
Raghav Luthra	AdminServer, Card, CardType, Deck, User, CardGenerator. Logic for buttons, displaying relevant information and UI for LoginScreen(login, register, showing appropriate error messages), , UserHomeScreen and CategoryScreen.	
Rohit Raj	Category, researched on ui ,defcard, researched on design patterns, UML, user homescreen, ui research,edit card, worked on ispublic implementation of decks	
Siddharth S. Shah	Deck,user, card generator, solid principles, Custom exception handling, app.java, categoryscrenecontroller, deckScreenController,loginscr eencontroller, scenehandler, MCQcardcontroller,main.java	
Manas	NO CONTRIBUTION	

Design Pattern Used

Factory

This pattern ensures that the correct type of object is instantiated at runtime based on an input from the user. This is seen when we instantiate a new card- it can have types like True-False, Fill in the Blanks, Definition or MCQ which is decided at runtime by the class cardGenerator. In our case, use of this pattern ensures that the code is scalable, meaning we can add other types of cards very easily since this outsources the responsibility of creating cards to the Factory class(CardGenerator). The code snippet is provided below.

```
public class CardGenerator {
    public Card newCard(String question, String answer, Category category, CardType cardType)
    {
        switch(cardType)
        {
            case DEFINITION:
                return new DefCard(question,answer,category);
            case FIB:
                return new FIBCard(question,answer,category);
            case MCQ:
                return new MCQCard(question, answer, category);
            case TF:
                return new TFCard(question, answer, category);
        }
        return null;
    }
}
```

Singleton

Sometimes we have a case such that a class must have one instantiation only across the whole application. In our case, we have an *AdminServer* class which is responsible for registering new users, logging in existing users and getting the top contributors. We require only one instance of this which multiple users can use at same the time, hence we decided to go with singleton pattern here. We also have a singleton pattern in *SceneHandler* class which is responsible for switching between different scenes. Both of these classes also support multithreading as the `getInstance()` method is synchronized in both classes.

```

public class AdminService {
    private static AdminService instance;

    synchronized public static AdminService getInstance() {
        if(instance == null)
            instance = new AdminService();

        return instance;
    }
}

```

Snippet for AdminService

```

public class SceneHandler {

    private static SceneHandler instance;

    synchronized public static SceneHandler getInstance() {
        if(instance == null)
            instance = new SceneHandler();

        return instance;
    }
}

```

Snippet for SceneHandler

Mediator

We have 6 different scenes in our application and they frequently need to communicate and propagate data between each other. The communication logic between scenes is pretty complex so there is a risk of coupling between these classes. Therefore we decided to implement a mediator design pattern here. The *SceneHandler* class acts as a central point of communication between different scenes in our project and we can use its singleton instance to switch between scenes and pass user information, etc between scenes.

```

public void switchToCreateMCQCardScene(Stage stage, Scene previousScene, Deck deck)
{
    FXMLLoader loader = new FXMLLoader(getClass().getResource("CreateMCQCard.fxml"));
    Parent root = null;
    try{
        root = loader.load();
    }catch (IOException e)
    {
        throw new RuntimeException(e);
    }//TODO: custom exception
    CreateMCQCardController createMCQCardController = loader.getController();
    createMCQCardController.setDeck(deck);

    String username = stage.getTitle();
    createMCQCardController.setUser(username);

    createMCQCardController.setPreviousScene(previousScene);
    Scene nextScene = new Scene(root);
    stage.setScene(nextScene);
}

public void switchToEditCardScene(Stage stage, Scene previousScene, Card card)
{
    FXMLLoader loader = new FXMLLoader(getClass().getResource("EditCardScene.fxml"));
    Parent root = null;
    try{
        root = loader.load();
    }catch (IOException e)
    {
        throw new RuntimeException(e);
    }//TODO: custom exception
    EditCardSceneController editCardSceneController = loader.getController();
    editCardSceneController.setCard(card);
    editCardSceneController.setPreviousScene(previousScene);
    Scene nextScene = new Scene(root);
    stage.setScene(nextScene);
}

public void switchToScene(Stage stage, Scene scene)
{
    stage.setScene(scene);
}

```

Analysis of Code using SOLID Principles

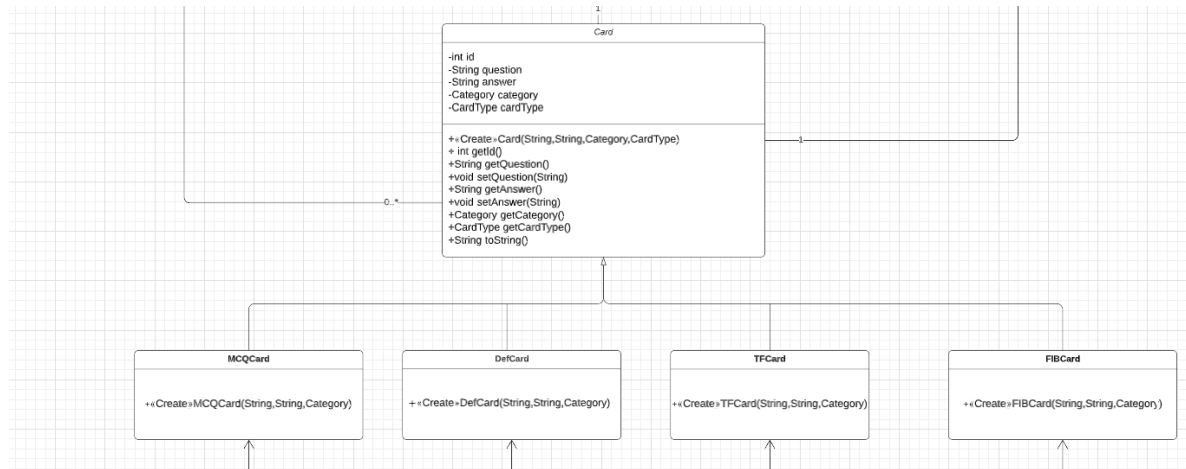
1. Single Responsibility Principle

Single responsibility principle states that a class should only be responsible for one thing and therefore, should have a single reason to change. If a single class has a lot of responsibilities, it decreases the scalability of the code.

In our project, we used multiple classes for different functionalities of the application. There were separate classes for card related, deck related, category and user related functions. To generate a specific type of card we used the cardGenerator class. Also we made separate classes for handling custom exceptions. This increased the reusability of the code throughout the project.

2. Open Closed principle

- a. As we know in open closed principle, the existing functionalities must be closed for modification but the code should easily be ready for expansion i.e. to add a new feature the old classes should not be severely affected.
- b. We have implemented an abstract class named *Card* which is used to create different types of Cards - FIB, Definition, T/F, MCQ. If we want to add a new type of card to our application, we can do so without touching the existing *Card* class. All we need to do is create a class for the new card type and make sure it inherits from *Card*. UML for the code implementation is attached below.



3. Liskov substitution principle

- Liskov substitution principle mentions that the superclass references must be able to hold objects of the child class.
- In the example attached below, we are using a reference of superclass *Card* to store an object of Type *DefCard* which is its child class. Since there are 4 types of cards in our application, we have to create them based on users' input at runtime and we can store them in a *Card* reference, since it is the superclass.

```
Card card = (new CardGenerator()).newCard(questionText.getText(), answerText.getText(), deck.getCategory(), CardType.DEFINITION);
deck.addCard(card);
```

- Continuing on the same example, the *CardGenerator* can return any of the 4 types of cards, and we can use the return type *Card* for them since it is the superclass.

```
8 usages  ▲ Raghav Luthra
public class CardGenerator {
    4 usages  ▲ Raghav Luthra
    public Card newCard(String question, String answer, Category category, CardType cardType)
    {
        switch(cardType)
        {
            case DEFINITION:
                return new DefCard(question,answer,category);
            case FIB:
                return new FIBCard(question,answer,category);
            case MCQ:
                return new MCQCard(question, answer, category);
            case TF:
                return new TFCard(question, answer, category);
        }
        return null;
    }
}
```


4. Interface segregation principle

- a. Having multiple specific interfaces is preferable to having one general purpose interface since that ensures that users don't need to define any unnecessary methods or pass any parameters they do not have use for.
- b. Currently we have used only one abstract class and no interface but in the future we plan to include said principle in our project implementation. One way we can do this is to segregate different card type classes based on their answer type. For example, the Definition card and Fill-in-the-blanks card both have String type answers, so they can both implement an interface like <StringAnswerCard>. Similarly True-False card can implement an interface like <BooleanAnswerCard> and MCQ card can implement an interface like <IntegerAnswerCard>.

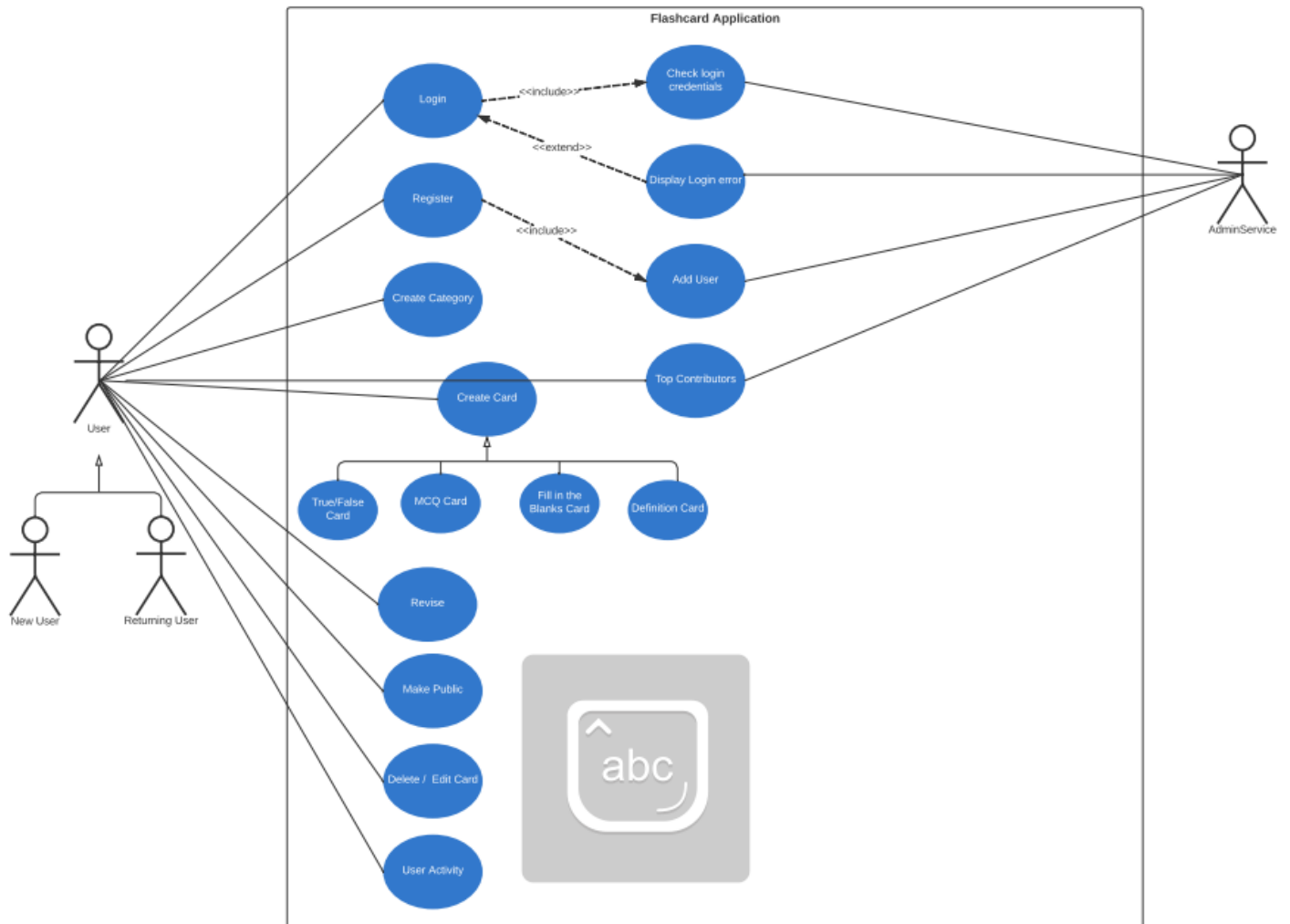
5. Dependency Inversion principle

- a. The Dependency Inversion principle states that our classes should depend upon interfaces or abstract classes instead of concrete classes and functions.
- b. The main feature of our application - the flashcards - are dependent on the abstract class *Card*. What we can do in the future is, make the Card class into an interface instead of an abstract class, so the different cards can implement other specific interfaces and this also leaves open a space for inheriting from some other class in the future since Multiple inheritance is not allowed in Java.

Use Case:

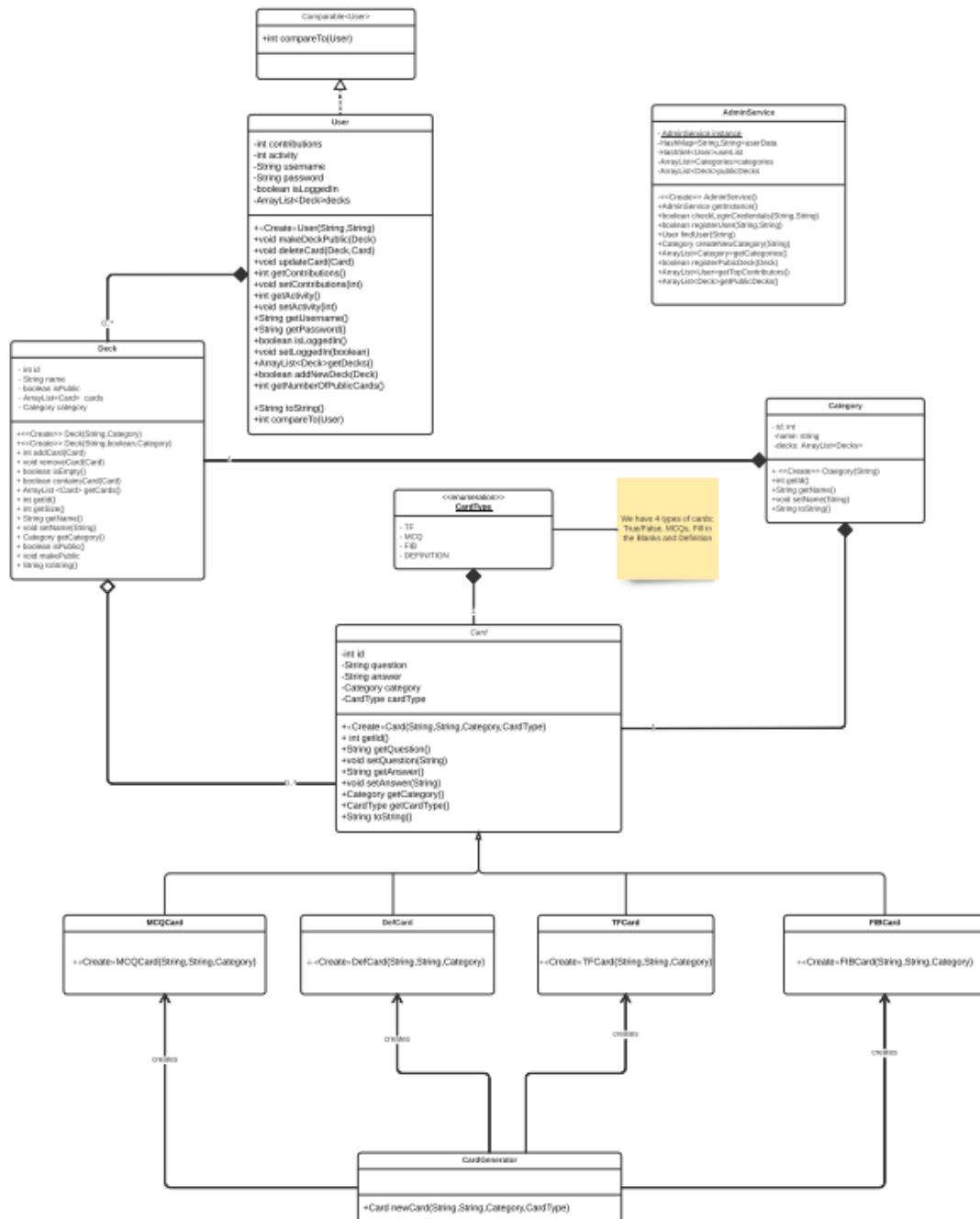
Flashcard Application Use Case Diagram

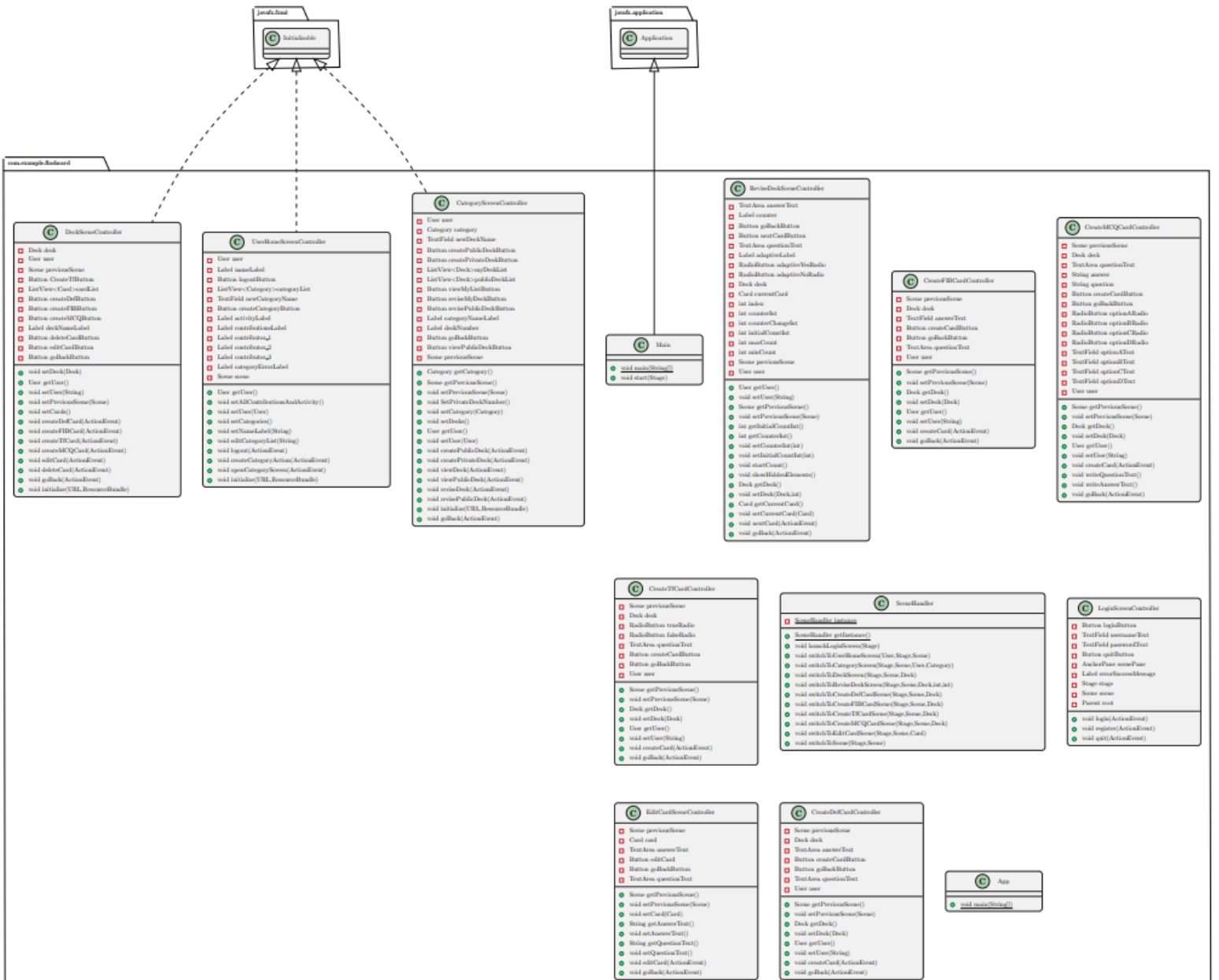
GROUP 29



UML:

Flashcard Application UML Class Diagram

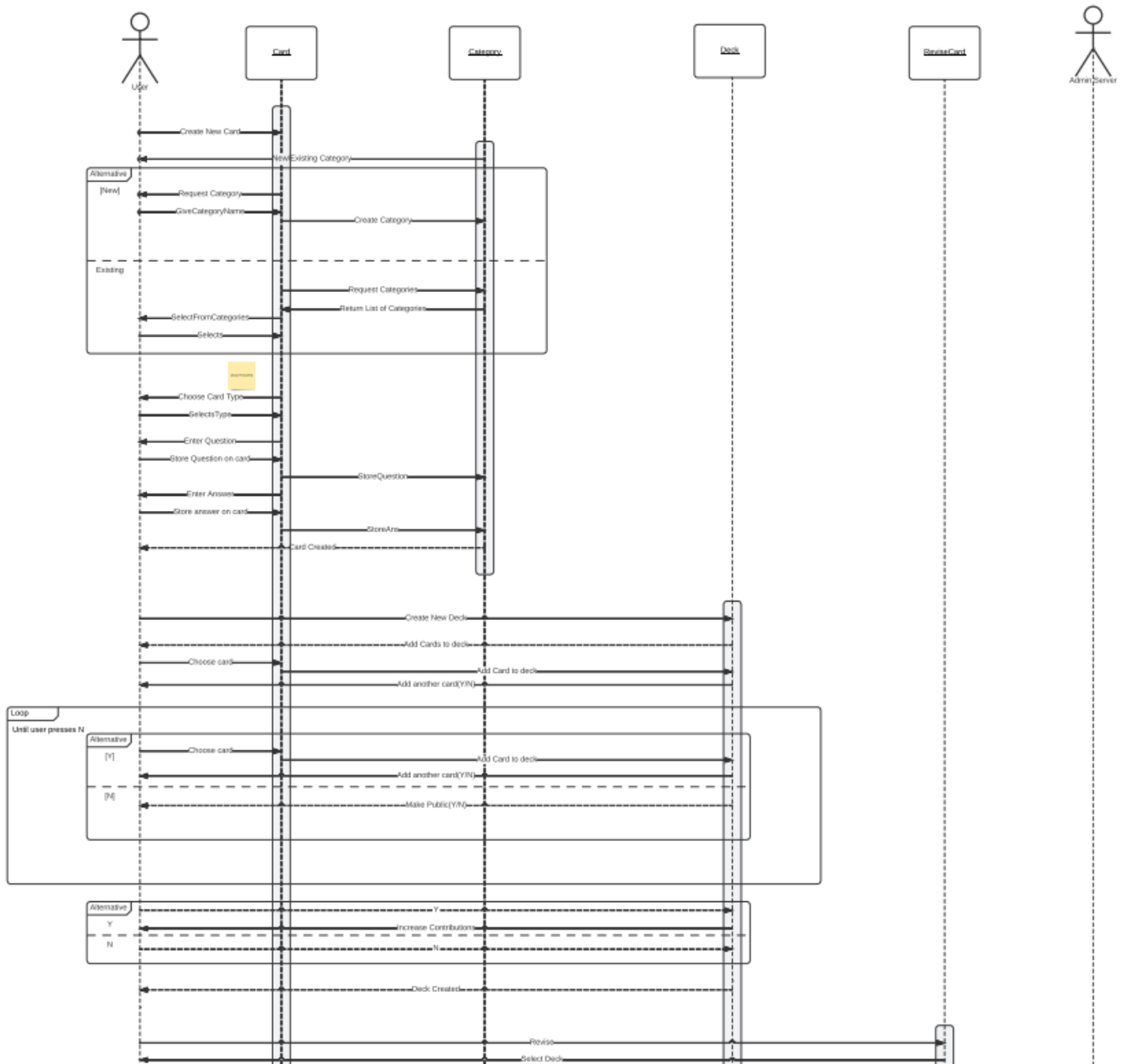


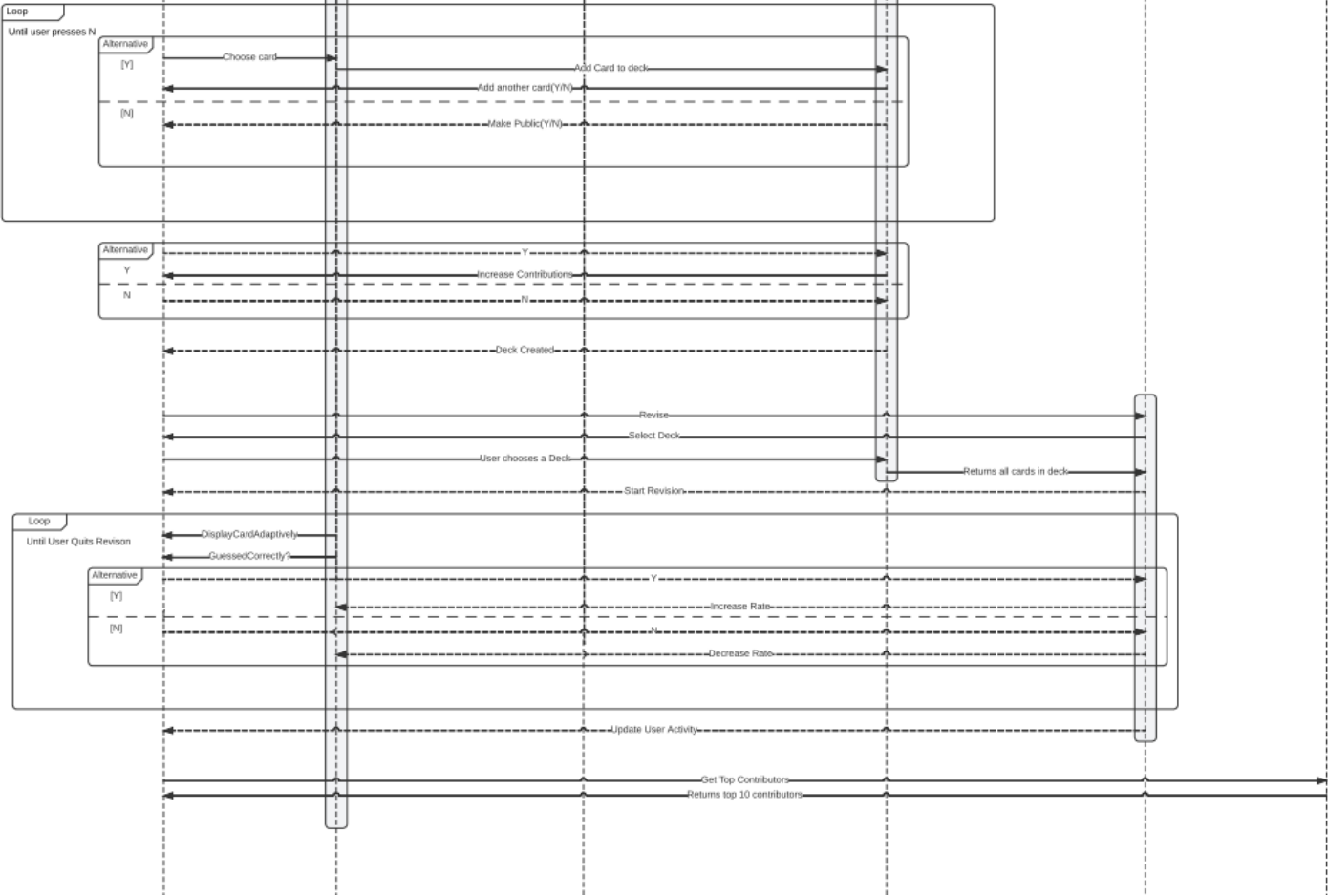


Sequence Diagram:

Flashcard Application UML Sequence Diagram

GROUP 23





Bonus

GUI

The members of the project used JavaFX along with SceneBuilder to build an interactive user-friendly GUI for the FlashCard Application. The members involved in developing the GUI put in considerable amounts of effort to make sure that the program did not have any bugs, along with implementing Exception Handling to make sure that the program runs smoothly.

GIT:

The members of the project put in effort to understand version control and learn how git works to collaborate on this project.

Multithreading

- Multiple users can login and use the application at the same time because the admin server supports multithreading
- Many components of JavaFX being used during GUI implicitly use multithreading.
- A timer thread is created every time we revise a card in an adaptive learning fashion.

Assumptions

Types of Cards

- MCQ
- Fill in the Blanks
- Definition
- True and False

Algorithm for Adaptive Learning

The algorithm works on the user's input- whether the user was able to correctly guess the answer or not. If the user is correct, then the next flashcard is displayed on the screen for 2 seconds less than the previous card. If the user is wrong, then the card is displayed at the screen for 2 seconds more than the previous card. There is also a ceiling of 20 seconds and a floor of 4 seconds i.e. the range of time would always lie in the range of 4 to 20 seconds.

Brief Description of Application

The application is designed for the user to use flashcards to help in their study for exams. The application gives the user the ability to create 4 different types of cards- true or false, MCQ, Definition, Fill in the Blanks which they can then view, edit and remove. There is also a functionality to revise them at an adaptive learning rate, view other users' public decks.