HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

****

PROJECT
Object-Oriented Programming

Electronic Piano

Instructor: ***Prof.Nguyen Thi Thu Trang***

Group 12: ***Ngo Thi Thu Huyen 20200289***
Vu Tan Khai 20200311
Nguyen Duy Khanh 20204914
Nguyen Ngoc Khanh 20204915

I.Assignment of members:

a. Ngo Thi Thu Huyen 20200289: Design GUI, handle button and test the features of app
    - MainScreens.fxml
    - Item.fxml
    - HelpScreen.fxml
    - RecordList.fxml
    - HelpScreenController.java
    - MainScreenController.java
    - RecordListController.java
    - Record demo video
b. Vu Tan Khai 20200311 : Sound of piano , handle button, test GUI
    - MainScreenController.java
    - Player.java
    - OrganPlayer.java
    - PianoPlayer.java
    - GuitarPlayer.java
    - MainScreens.fxml
    - Report editor
c. Nguyen Duy Khanh 20204914: music recording, handle button, test GUI
    - Recorder.java
    - MainScreenController.java
    - RecordListController.java
    - Item.fxml
    - RecordList.fxml
    - ItemController.java
d. Nguyen Ngoc Khanh 20204915: Design GUI, test the features of app
    - MainScreenController.java
    - MainScreens.fxml
    - Helptext.txt
    - Review code
    - Main writer for the report
    - Design presentation Slides

II.Mini-project description:

1.Mini-project requirement:

-Implement an application that provides GUI for the user to virtually play an electronic piano.

-Design requirement:

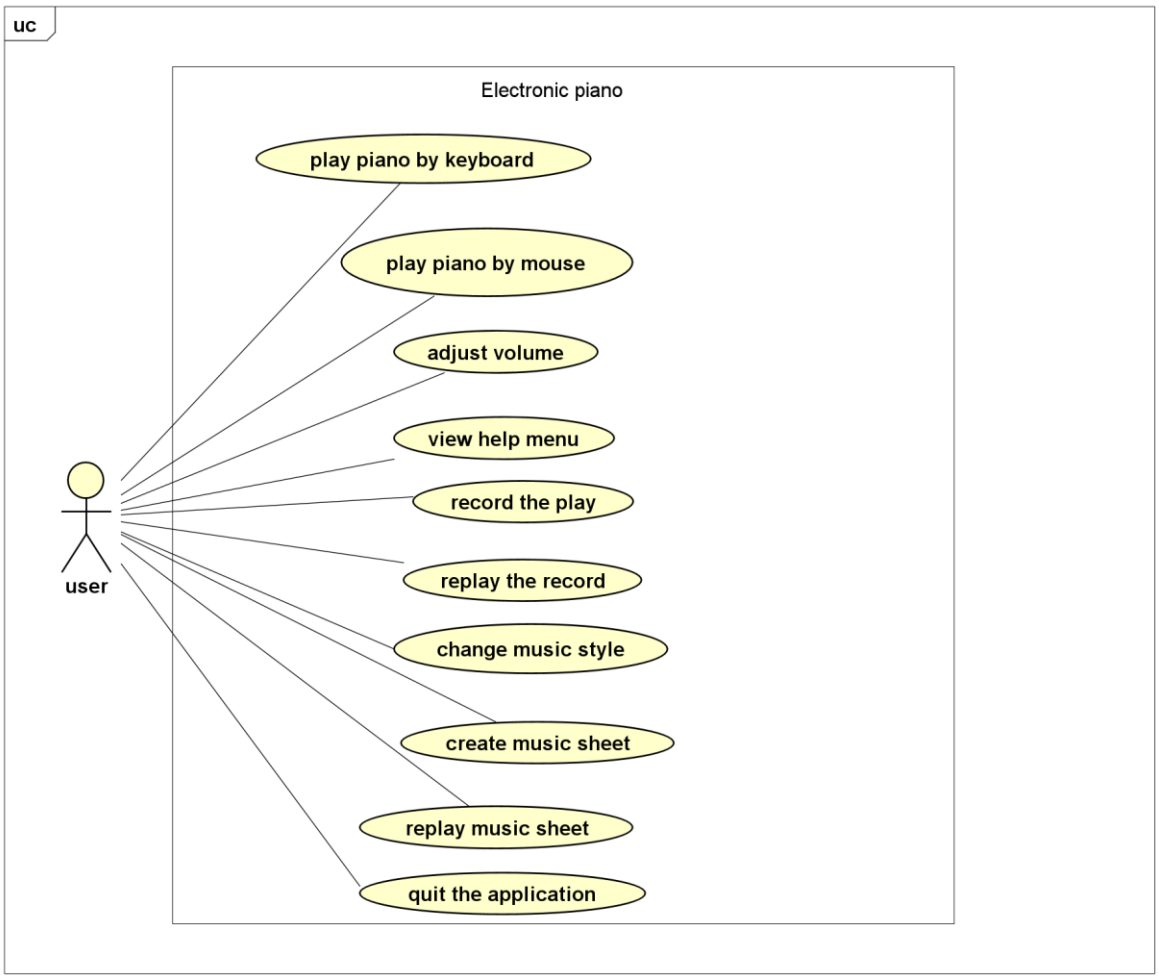
+ On the main menu: title of the application, piano GUI, help menu, quit
• User can play the piano by interacting with GUI
• Help menu shows the basic usage and aim of the program
• Quit exits the program. Ask for confirmation to quit

+Demonstration:

• Keyboard: C (Do), D (Re), E (Mi), F (Fa), G (Sol), A (La), B (Xi). The keyboard is

designed to be as much details as possible.
• A button for increasing/decreasing volume (optional)
• A record button to record the play (optional)
• A button for changing music style (optional)

2.Use case diagram:



Explanation:

-Play piano by keyboard: Users are able to play the electronic piano by pressing keyboard button. Which keyboard button corresponds to which piano key is shown on the piano.

-Play piano by mouse: User can also play the electronic piano by left clicking on the piano key.

-Adjust volume: there is a volume slider for user to increase or decrease the volume of the sound.

-View help menu: user can click on the help button to get the instructions of using the app.

-Record the play: allows user to record their play and then output a .wav file.

-Replay the record: user can replay their record by clicking on play button after clicking on record button.

-Change music style: there are 3 styles of sound for user to choose (piano, guitar, organ).

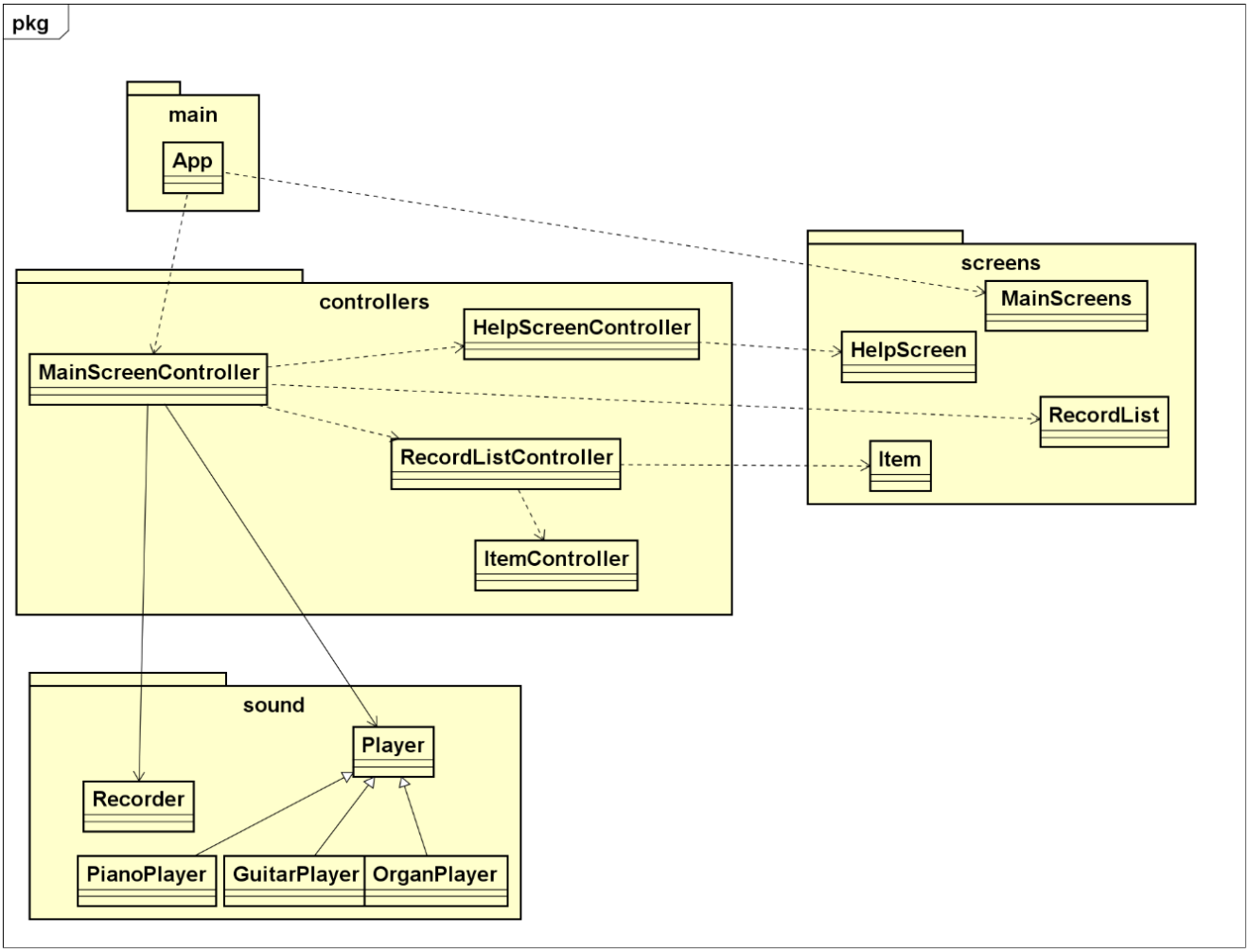
-Create music sheet: Each music key user play is logged in the music sheet (the white area right above the piano keyboard). Click on Clear button to restart a new sheet.

-Replay music sheet: click on Replay button to automatically play the music sheet.

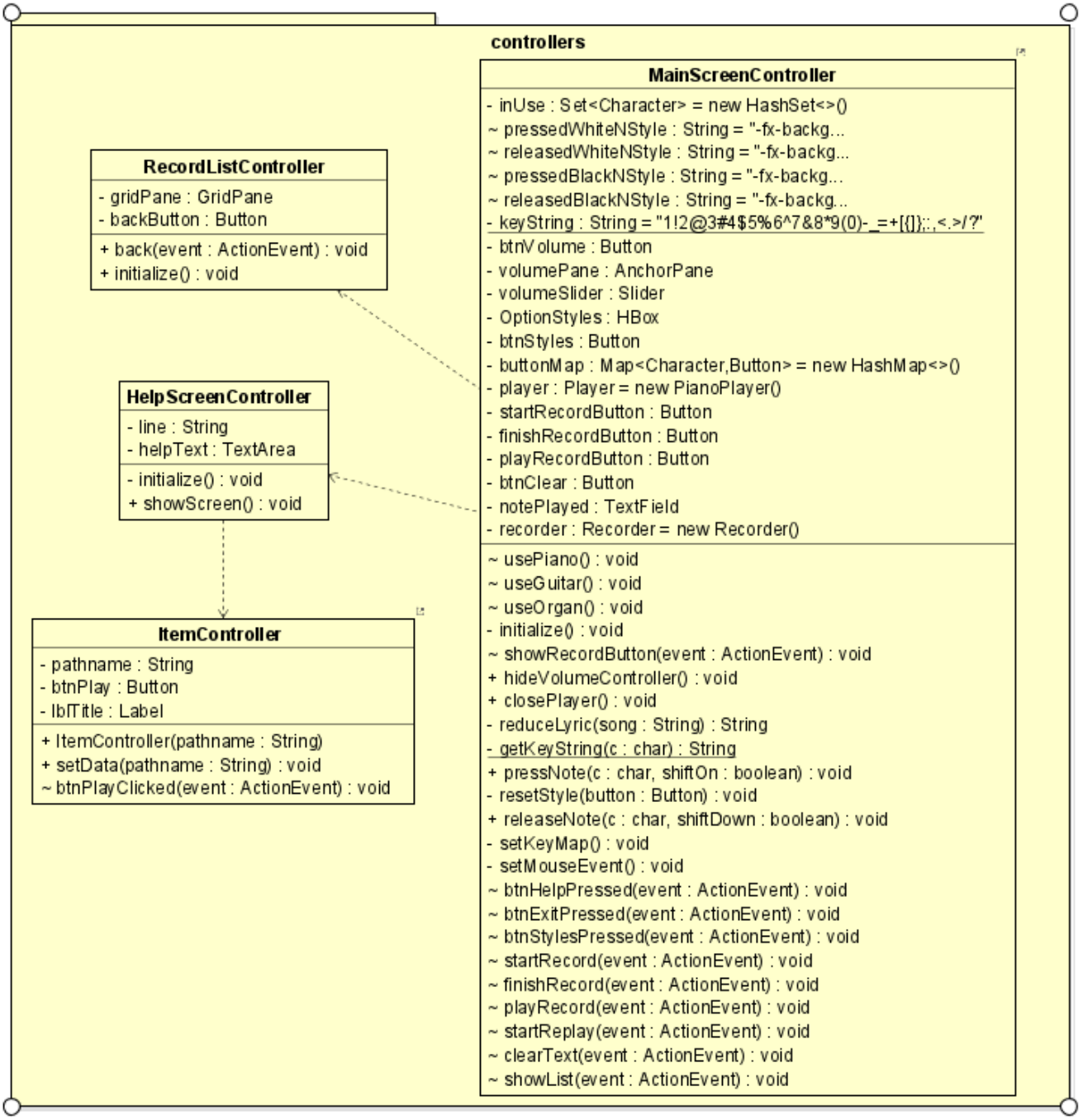
-Quit the application: click on exit button to terminate the program. User is asked for comfimation.

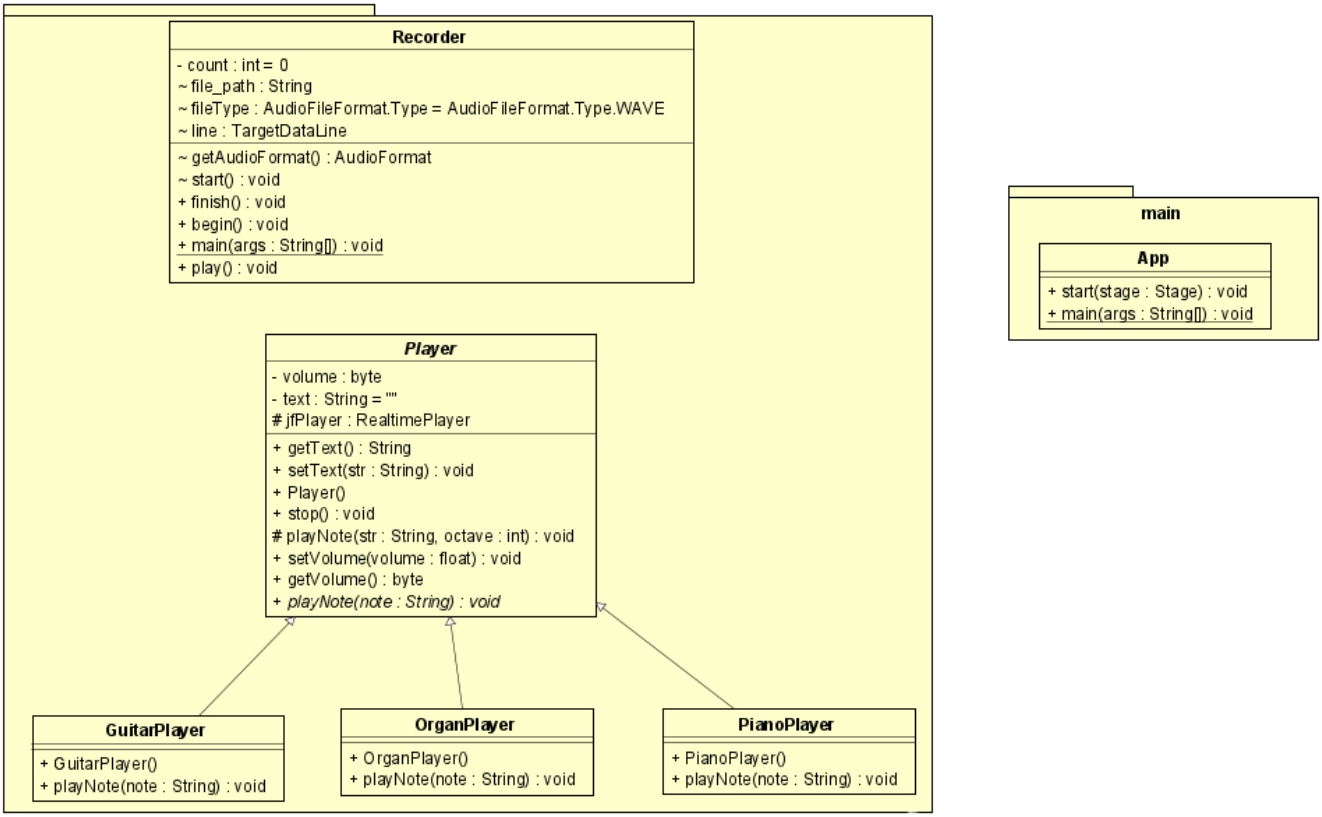
III.Design:

1.General class diagram:



2.Detail class diagram:





3.Explanation of the design:

-Inheritance: Our application allows user choosing how piano sounds from three types: guitar, organ and piano. Then, we create three classes corresponding to each type of the sound respectively: GuitarPlayer, OrganPlayer and PianoPlayer. These three classes all share same methods of playing note(playNote(note:String), playNote(str:String, octave:int)), adjusting the volume of the sound (setVolume(volume:float), getVolume(volume: float)), stop playing(stop()) and create the music sheet(getText(), setText(str:String)). To promote the reusability of the code, from that also enhance its reliability, we create an abstract class Player containing all shared methods and let GuitarPlayer, OrganPlayer, PianoPlayer inherit. To force each subclass to override playNote(note:String): void method, we make this method to be abstract method , that's why we let Player be an abstract class.

-Encapsulation: Within our project, in many cases, we want our classes' fields to be hidden from classes in other packages. Therefore, to change the internal state of a class instance, the developer will need to use available functions provided by that class. This allows us to control the usage of classes, make them less error-prone and easier to upgrade or refactor the project in the future. To achieve this, in this project, we use class modifier (private, protected, public) for each class field and getter/setter to retrieve/change class fields and other public functions to interact with them. For example, in package sound, Player class has volume field of type byte, jfPlayer field of type org.jfugue.realtime.RealtimePlayer. Developers can get current volume value by calling getter function. And to set the volume, they will use the setter function, which will then call a function of jfPlayer to change the volume. Or to play a certain musical note, developers must call function playNote instead of interacting directly with jfPlayer field. Suppose in the future, we want to use another external library for playing sound then we only need to change the fields of this class and keep the signatures of methods as before. This helps the development process easily adapt to changes.

-Polymorphism:

```
@Override
public void playNote(String note) {
    playNote(note, 5);
}

protected void playNote(String str, int octave) {
    //implementation
}
```

+Method polymorphism(method overloading): each instance of the class Player has two playNote methods with different signature: playNote(str: String, octave: int), playNote(note: String). The main difference between two methods is that while playNote(str: String, octave: int) let the programmer decide the octave parameter which is pitch[1] of the sound, the other fixes the octave parameter. Instead of having two methods with different name by changing playNote(str: String, octave: int) method to playNoteOctaveFive or playNoteOctaveFour or something similar to it, we take the advantage of method overloading technique to have two methods with the same name. So that , we successfully enhance the readability of the code.

+Object polymorphism:

```
public class PianoPlayer extends Player {
    @Override
    public void playNote(String note) {
        playNote(note, 5);
    }

    //methods
}
```

```
public class OrganPlayer extends Player {

    @Override
    public void playNote(String note) {
        playNote(note, 4);
    }

    //methods

}
```

[1] In music theory, pitch defines the highness of the sound

```java
public class GuitarPlayer extends Player {

    @Override
    public void playNote(String note) {
        playNote(note, 6);
    }

    //methods
}
```

```java
    private void setMouseEvent() {
        C1.setOnMousePressed(e -> {
            player.playNote("C1");
            notePlayed.setText(reduceLyric(player.getText()));
        });
        Db1.setOnMousePressed(e -> {
            player.playNote("Db1");
            notePlayed.setText(reduceLyric(player.getText()));
        });
        D1.setOnMousePressed(e -> {
            player.playNote("D1");
            notePlayed.setText(reduceLyric(player.getText()));
        });
        Eb1.setOnMousePressed(e -> {
            player.playNote("Eb1");
            notePlayed.setText(reduceLyric(player.getText()));
        });
        E1.setOnMousePressed(e -> {
            player.playNote("E1");
            notePlayed.setText(reduceLyric(player.getText()));
        });
        F1.setOnMousePressed(e -> {
            player.playNote("F1");
            notePlayed.setText(reduceLyric(player.getText()));
        });
        Gb1.setOnMousePressed(e -> {
            player.playNote("Gb1");
            notePlayed.setText(reduceLyric(player.getText()));
        });

        //Similar event handle for other piano keys

}
```

In the source code above, we can easily see the polymorphism in the player.playNote() statement. Thanks to the help of dynamic binding at runtime, instead of using three variables to store different types of player : PianoPlayer, OrganPlayer, GuitarPlayer, we can use a single player variable of Player type. So that, polymorphism reduces the size of the code, make it less complicated. As the result, it is easier to debug the code in the future.

-Association: Notice that in our implementation, MainScreenController class has two fields (player: Player and recoder: Recorder) so that it models the association between MainScreenController and Player, MainScreenController and Recorder. MainScreenController use Player object to control of how the application sound and log the music sheet, it also use Recorder object to manage all record operations.

4.Implementation of some important methods:

- playNote

  Input: str: String indicates a note

  octave: int indicates the pitch of the sound

```
protected void playNote(String str, int octave) {
    text=text+ str+ " ";
    int note;
    switch (str.charAt(0)) {
        case 'C': note = 0; break;
        case 'D': note = 2; break;
        case 'E': note = 4; break;
        case 'F': note = 5; break;
        case 'G': note = 7; break;
        case 'A': note = 9; break;
        case 'B': note = 11; break;
        default:
            System.out.println("Invalid note " + str.charAt(0));
            return;
    }
    if (str.contains("#")) note += 1;
    if (str.contains("b")) note -= 1;
    if (str.length() > 1) {
        char c = str.charAt(str.length() - 1);
        octave += c - '1';
    }
    note += octave * 12;
    jfPlayer.startNote(new Note(note));
}
```

This method is based on JFugue package to play sound. First it append a new note string to text variable (text contains the content of the music sheet). Then the method convert the note string to the corresponding integer number defining the frequency of the sound. If the note string contains “#” character, increment the note variable, if the note string contains “b” character, decrement the note variable. In music theory, “#” indicates a sharp note, “b” indicates a flat note. When a note's pitch is sharpened, it is raised by a semitone (or a half-step). Similarly, when a note's pitch is flattened, it is lowered by a semitone. The method also increase the note variable by octave * 12 given that each octave is different by 12 units. Finally, it executes jfPlayer.startNote(new Note(note)) statement to play sound.

-Recorder class:

```
void start() {
    try {
        AudioFormat format = getAudioFormat();
        DataLine.Info info = new DataLine.Info(TargetDataLine.class, format);

        if (!AudioSystem.isLineSupported(info)) {
            System.out.println("Line not supported");
            System.exit(0);
        }
        line = (TargetDataLine) AudioSystem.getLine(info);
        line.open(format);
        line.start();

        AudioInputStream ais = new AudioInputStream(line);

        System.out.println("Start recording...");
        while (true) {
            File file = new File("/OOP.DSAI.20212.12/resources/audio/RecordAudio"+ String.valueOf(count+1)+".wav");
            if (file.exists()== true) {
                count+=1;

        }
            else {
                break;
            }
        }
        count+= 1;
        file_path = "/OOP.DSAI.20212.12/resources/audio/RecordAudio"+ String.valueOf(count)+".wav";
        File wavFile = new File(file_path);
        AudioSystem.write(ais, fileType, wavFile);

    } catch (LineUnavailableException ex) {
        ex.printStackTrace();
    } catch (IOException ioe) {
        ioe.printStackTrace();
    }
}
```

```
public void begin() {
    Thread stopper = new Thread(new Runnable() {
            public void run() {
                start();
            }
        });
    stopper.start();
}
```

```
public void finish() {
    line.stop();
    line.close();
    System.out.println("Finished");
}
```

```
public void play() {
    File sound = new File(file_path);

    try {
        AudioInputStream ais = AudioSystem.getAudioInputStream(sound);
        Clip c = AudioSystem.getClip();
        c.open(ais);
        System.out.println("Playing");
        c.start();

        Thread.sleep((int)(c.getMicrosecondLength() * 0.001));
    } catch (Exception e) {
        System.out.println(e.getMessage());
    }

}
```

The Recorder class is dedicated to storing the input sound from the environment into a wav file. The wav is stored in the location stored in the file_path variable in start method. The whole record procedure is executed in a different thread from the main thread which will increase the performance of the application. The play() method is implemented to help the user play the record they want.