Project: Piano GUI

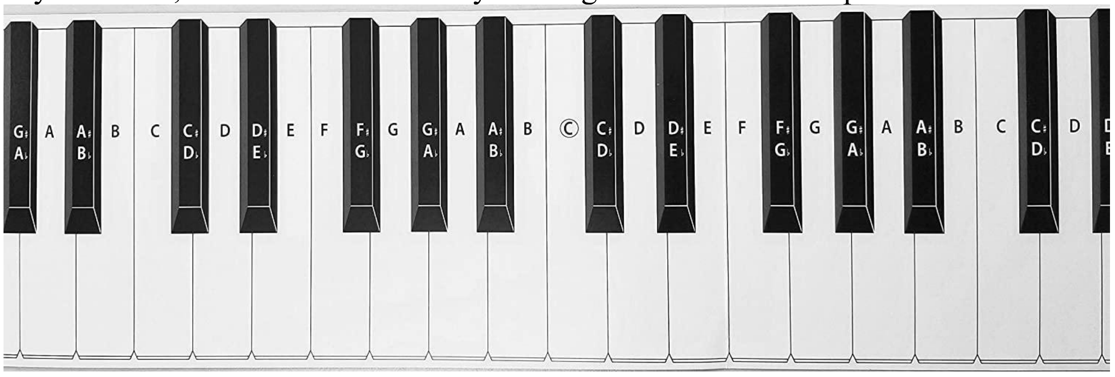
- *Class: 143577*
- *By: Group 22*:

Tran Binh Minh 20215094(TL)
Phan Do Hai Minh 20184154
Nguyen Huu Minh 20215091
Lieu Nhat Minh 20215089

I - Description

Overview: Piano is a popular musical instrument. In this project, you are asked to implement an
application that provides GUI for the user to virtually play an electronic piano.
Basic Knowledge: Data structure, Algorithms, Project Settings, and Music

Specifications:
- GUI: you can freely design your own GUI. However, since the basic aim of the project is to
develop an application based on OOP, focusing on the interface is not required
You can refer to these sources to have some idea:
https://www.youtube.com/watch?v=DFnUDyzF1Uk
https://www.youtube.com/watch?v=kvVdeARztt0
- To implement this project, you just need some basic music knowledge. *You don't have to implement a mechanism to play sound, you can directly import libraries from any source.*
You can refer to some libraries:
http://www.jfugue.org/
https://github.com/TitasNandi/Chord-JAVA
- Design:
+ 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. Remember to ask for confirmation
+ In the demonstration
  - Keyboard: C (Do), D (Re), E (Mi), F (Fa), G (Sol), A (La), B (Xi), … You can design
a keyboard, it doesn't need to be a standard one. However, the more details the keyboard has, the more assessments you can get. Here is an example:



- A button for increasing/decreasing volume (optional)
- A record button to record the play (optional)
- A button for changing music style (optional)

Note: You should use libraries and learn how to add libraries to your project. Again, you are
Not: asked to implement how the sound will be played.

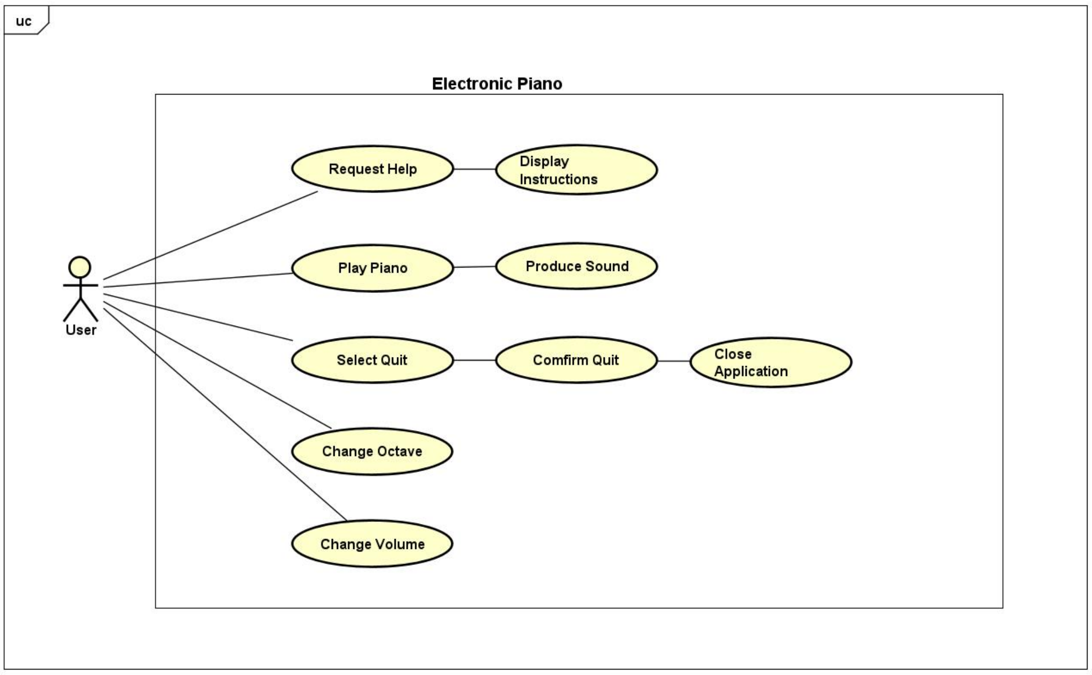
II - Member Roles

Tran Binh Minh:
- Idea generation and conceptual designs.
- Slide preparation.
- Framework decision.
- Code contribution.

Nguyen Huu Minh:
- Technical mentorship and guidance.
- Code integration and optimization.

Lieu Nhat Minh & Phan Do Hai Minh:
- Diagram creation.
- Code contribution.

III- Design



Usecase Diagram

- There are three main functionalities available to the user, each represented as a use case in our system.
- **Play Piano**: This is the central feature of our application. When the user engages with this function, they can play music using the electronic piano interface. Each keypress corresponds to a musical note, which is then processed by the system to "Produce Sound," the essential outcome of this interaction.

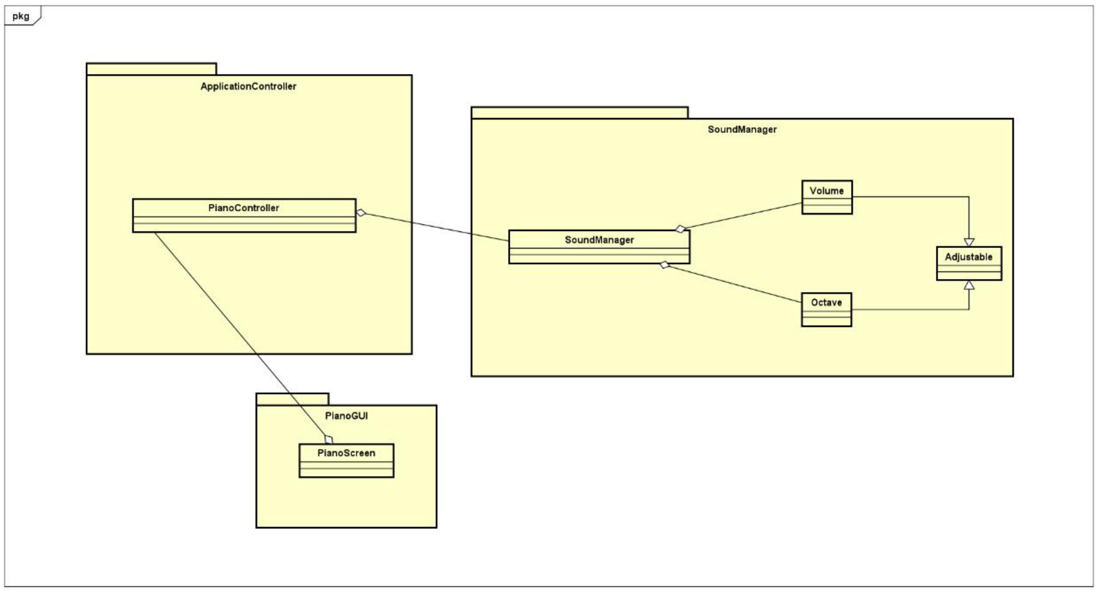
- Request Help: System responds by "Displaying Instructions." This feature is crucial for providing users with the guidance they may need to effectively utilize all the functions of the electronic piano.
- Select Quit: When the user decides to end their session, they can "Select Quit." This action triggers the system to "Confirm Quit," which may involve a prompt ensuring that the user indeed intends to close the application. This confirmation step is an important aspect of user experience design, as it prevents accidental closure of the application and potential data loss.
- The "Produce Sound" use case is directly linked to "Play Piano," indicating that sound production is an immediate and integral response to the user's interaction with the piano keys.
- Similarly, the "Display Instructions" use case is linked to "Request Help," which ensures users have access to support materials.
- The "Confirm Quit / Close Application" use case is connected to "Select Quit." This represents the system's response to the user's request to exit the program, ensuring that the user's intention is verified before the application shuts down.

Change Octave:

This use case represents the user's ability to change the octave in which they are playing. The system will respond to this action by adjusting the pitch range of the sounds produced. This feature allows the user to access different registers of the piano, enabling them to play a wider range of music.

Change Volume:

This use case allows the user to adjust the volume of the sound output. It's a critical functionality for user experience as it provides the flexibility to set the sound level to a comfortable listening volume or to match the ambiance of the user's environment.



General Class Diagram

Starting with the **ApplicationController** package, we have:

- **PianoController:** This is the core controller of our application. It serves as the intermediary between the user interface and the sound management system. The **PianoController** is responsible for handling user inputs, processing them, and determining the appropriate outputs.

Moving on to the **PianoGUI** package, there is:

- **PianoScreen:** It represents the graphical user interface of our Electronic Piano. The **PianoScreen** is where the user interacts with the application, triggering different events like playing notes, changing octaves, or adjusting the volume. It is directly connected to the **PianoController**, which suggests that any action on the GUI is communicated to the **PianoController** for processing.

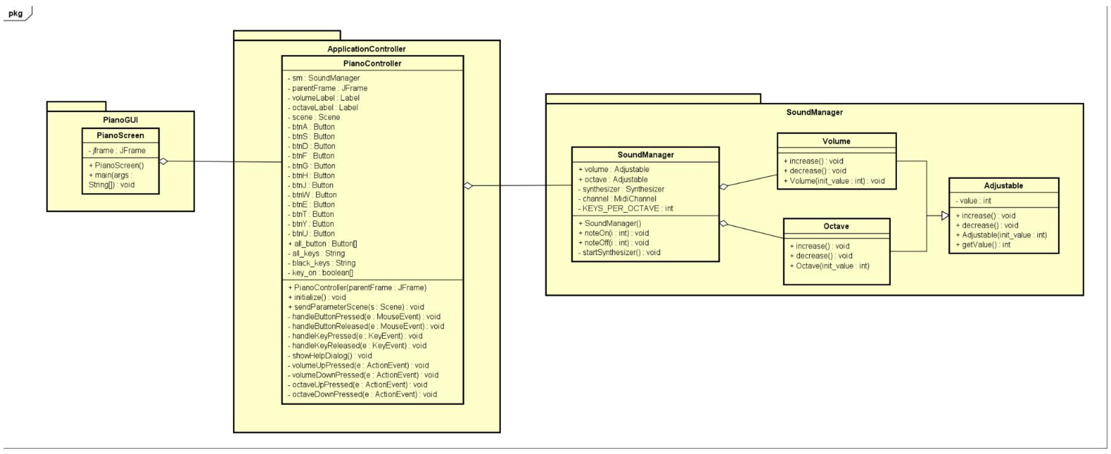
In the **SoundManager** package, we have:

- **SoundManager:** The **SoundManager** is a crucial component that handles the audio logic of our application. It is responsible for producing sounds when the user plays the piano and adjusting sound settings such as volume and octave.
-
    - **Volume:** A class that inherits from **Adjustable**. It specifically manages the volume level in the application. The **SoundManager** uses this to set the loudness of the sound produced.
    -
    - **Octave:** Another subclass of **Adjustable**, which handles the current octave setting. This allows the **SoundManager** to produce sounds in different pitches based on user interaction.
    -
- **Adjustable:** This abstract class defines a structure for objects that can have their value increased or decreased. Both **Volume** and **Octave** extend this class, which means they inherit its properties and implement its methods, **increase()** and **decrease()**, according to their specific context.
-

The arrows indicate the relationships between the classes:

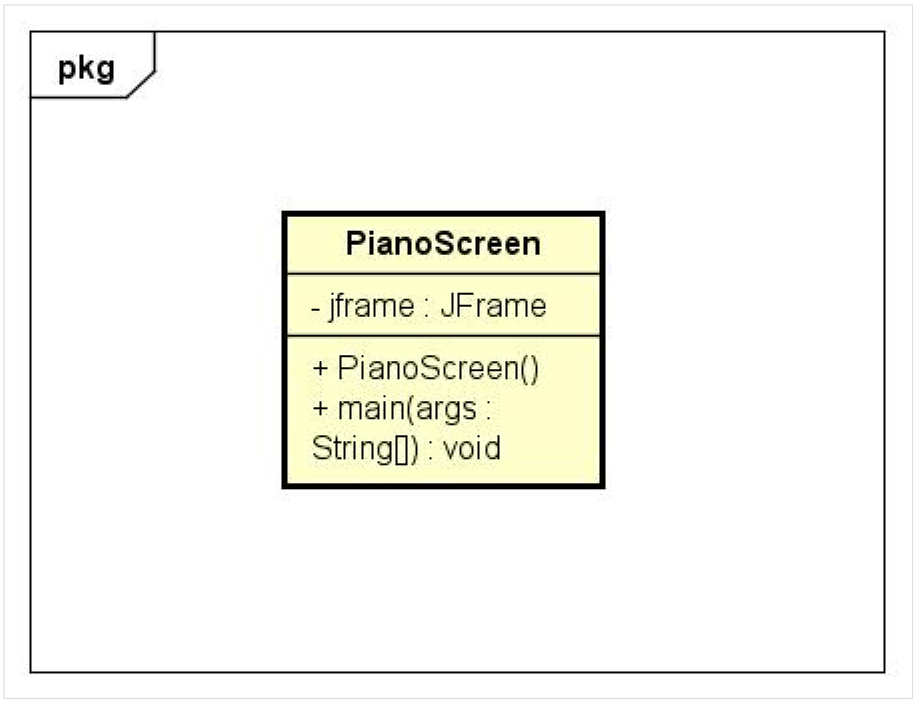
- The **PianoController** has a direct association with **SoundManager**, indicating that it utilizes **SoundManager** to produce the appropriate sounds.

- **SoundManager** has a composition relationship with **Volume** and **Octave**, meaning **SoundManager** is composed of these components and is responsible for their lifecycle.



Detailed Class Diagram

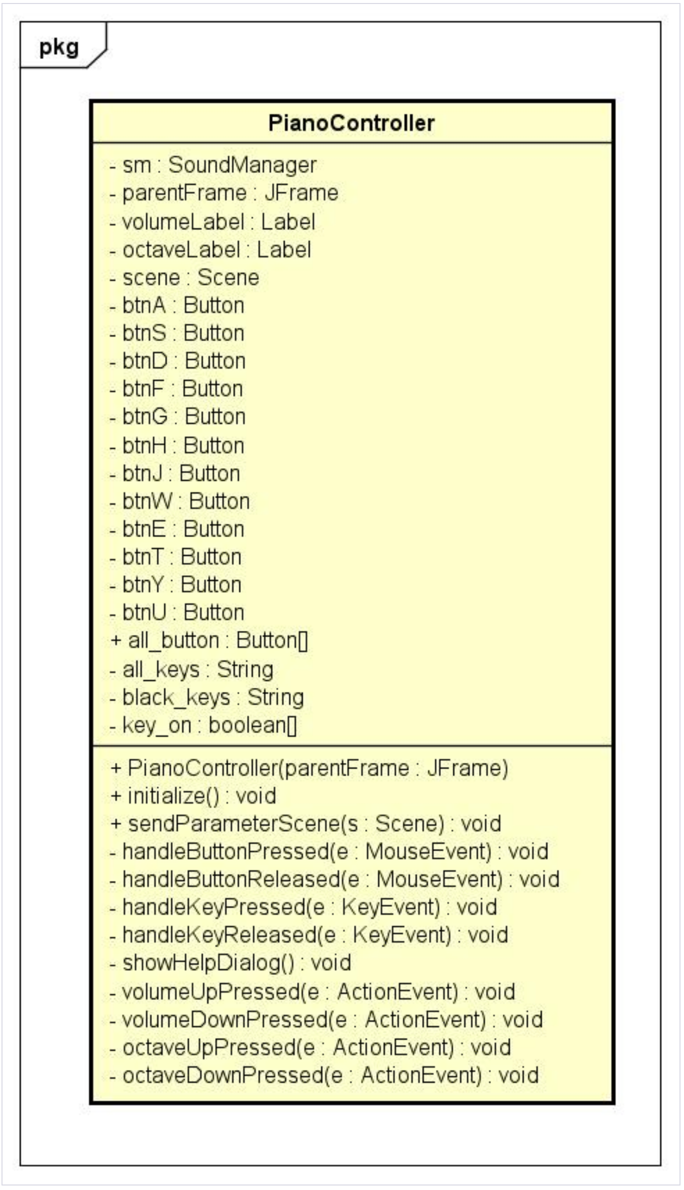
PianoGUI Package:



- **PianoScreen:**
  - Attribute: **- jFrame: JFrame** suggests that this class is responsible for the main application window.
  - Methods:

- **+ PianoScreen()** is the constructor for creating an instance of the PianoScreen.
- **+ main(args: String[])** is the entry point for the application, indicating it's likely a standalone Java application.

ApplicationController Package:

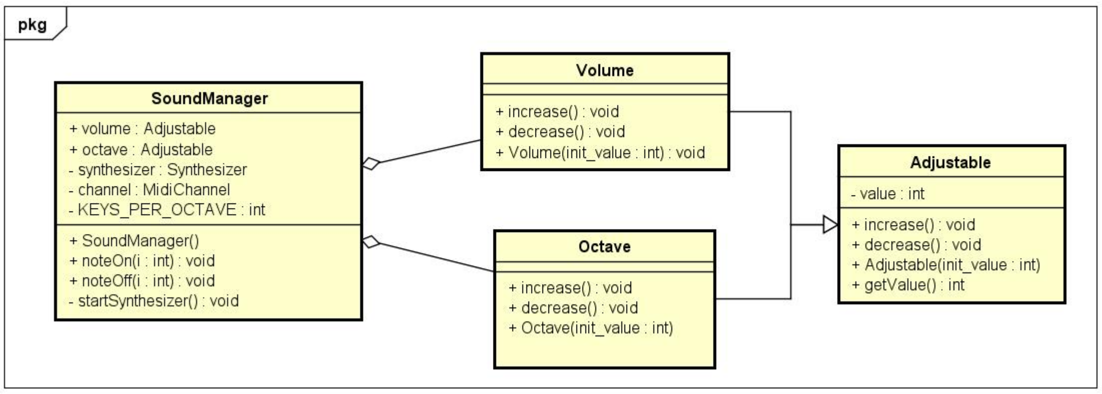


```
PianoController
- sm : SoundManager
- parentFrame : JFrame
- volumeLabel : Label
- octaveLabel : Label
- scene : Scene
- btnA : Button
- btnS : Button
- btnD : Button
- btnF : Button
- btnG : Button
- btnH : Button
- btnJ : Button
- btnW : Button
- btnE : Button
- btnT : Button
- btnY : Button
- btnU : Button
+ all_button : Button[]
- all_keys : String
- black_keys : String
- key_on : boolean[]

+ PianoController(parentFrame : JFrame)
+ initialize() : void
+ sendParameterScene(s : Scene) : void
- handleButtonPressed(e : MouseEvent) : void
- handleButtonReleased(e : MouseEvent) : void
- handleKeyPressed(e : KeyEvent) : void
- handleKeyReleased(e : KeyEvent) : void
- showHelpDialog() : void
- volumeUpPressed(e : ActionEvent) : void
- volumeDownPressed(e : ActionEvent) : void
- octaveUpPressed(e : ActionEvent) : void
- octaveDownPressed(e : ActionEvent) : void
```

- **PianoController:**
  - Attributes: Multiple fields like **- sm: SoundManager**, **- parentFrame: JFrame**, and various **- btn*: Button** suggest this class manages the user interface and interaction with the SoundManager.
  - Methods: Several methods for handling different user actions, like button presses and key events (**handleButtonPressed**, **handleKeyPressed**), and for managing volume and octave adjustments (**volumeUpPressed**, **octaveDownPressed**).

SoundManager Package:



- **SoundManager:**
  - Attributes: It contains adjustable volume and octave settings, a synthesizer, and a MIDI channel for playing sounds.
  - Methods: Include initializing the synthesizer (**startSynthesizer**) and playing or stopping notes (**noteOn**, **noteOff**).
- **Adjustable (abstract class):**
  - Attribute: **- value: int** represents a generic value to be adjusted.
  - Methods: Common methods to increase or decrease the value and a constructor.
- **Volume:**
  - Inherits from Adjustable and likely manages the volume setting specifically.
- **Octave:**
  - Also inherits from Adjustable, with methods tailored to adjusting the octave.

The diagram exhibits a clear separation of concerns among the classes, a fundamental principle in object-oriented design. The **PianoGUI** is concerned with the graphical user interface, the **ApplicationController** handles user input and application logic, and the **SoundManager** deals with audio output.

In terms of OOP principles, we see:

- **Encapsulation:** Each class has private attributes and public methods to manipulate these attributes safely.
- **Inheritance:** The **Volume** and **Octave** classes extend the **Adjustable** class, inheriting its properties and behaviors.
- **Composition:** The **PianoController** class contains an instance of **SoundManager**, showing a "has-a" relationship.
- **Polymorphism:** can be observed in the relationship between the Adjustable class and its subclasses Volume and Octave. Here's how polymorphism is applied:
  - Adjustable Class: This abstract class likely defines the increase() and decrease() methods, which are intended to adjust a value.

- Volume and Octave Classes: Both of these classes inherit from Adjustable. They will have their own implementations of the increase() and decrease() methods that are specific to their context (i.e., adjusting volume levels or octave levels).