Chapter 1: Report

Code Report: Music Notation Editor

Introduction

The Music Notation Editor is a Java Swing application designed for creating and editing musical scores. The application allows users to compose music by placing notes on a grand staff, play the composed music, and save or import projects. This report provides an overview of the project's focus, contents, and purpose, discussing the methods used, implementation details, and testing strategies.

Project Focus and Purpose

The primary focus of the Music Notation Editor is to provide a user-friendly interface for creating and editing musical compositions. The application is designed to offer essential functionalities such as note placement, playback, and project management. It targets musicians, composers, and enthusiasts who want a digital tool to facilitate the process of musical composition.

Contents

The project consists of the following major components:

1. **App Class:** The main entry point of the application. It initializes the main frame and triggers the creation of the UI.
2. **UI Class:** Responsible for initializing the user interface, including the toolbar, and the grand staff. It utilizes other classes like `ToolBar` and `GrandStaff` to create a cohesive user experience.
3. **ToolBar Class:** Implements the tool bar containing buttons representing whole, half, and quarter notes. It uses icons for the note buttons and provides an intuitive way for users to select note durations.
4. **GrandStaff Class:** Represents the central part of the application where users can place notes on a grand staff. It includes measures, clefs, and vertical bars to distinguish between musical sections.
5. **Measure Class:** Represents an individual measure within the grand staff. It draws clefs, lines, and vertical bars, creating a structured layout for musical composition.
6. **Note class:** represents a musical note within the Music Note Placement Application. Each instance of the Note class encapsulates information about a specific note, including its position on the staff, pitch, duration, and visual representation.
7. **MIDIPlayer Class:** Responsible for handling MIDI playback functionality within the Music Note Placement Application. It provides methods for adding musical notes to a MIDI sequence, playing the sequence, stopping playback, and managing the MIDI sequencer's state. The MIDIPlayer class interfaces with the Java Sound API to generate MIDI events for note playback.

Methods Used

Layout Management:

- **BorderLayout:** Utilized in the `UI` class to organize the `ToolBar`, and `GrandStaff` within the frame.
- **FlowLayout:** Applied to the `ToolBar` class for button arrangement within the tool bar.
- **GridBagLayout:** Employed in the `GrandStaff`class to provide fine-grained control over the layout of musical measures and components within the staff. Allows for precise positioning and sizing of musical elements, ensuring proper alignment and spacing in the grand staff display.

Event Handling:

- **ActionListener:** Implemented in the `NoteButtonListener` class to handle button clicks on the `ToolBar`. It appends the selected note symbol to the music text area.

Implementation Details

The application employs the Model-View-Controller (MVC) architectural pattern to separate concerns and enhance maintainability. The UI is structured with different classes responsible for specific aspects of the user interface. Icons are used for visual representation, enhancing user understanding.

Implementation History

We first started writing code manually as we had little to no experience using ChatGPT to write any code. This was very slow and the first few days had little progress. We then learned from the professor's example project the usefulness of ChatGPT.

We decided to scrap everything and start over building with ChatGPT. It gave impressively detailed skeleton code but had many errors. We would copy and paste the implementations it recommended and built up a few functionalities.

1. Ability to show measures with lines segmenting their rows and columns
2. Placement of time signatures and clefs (these had hard coded absolute coordinates)
3. Placement of notes, we struggled to center them because of their non consistent png origins
4. Ability to make multiple grand staffs (we had problems with interfering mouse click events between the different staffs)
5. Ability to delete notes
6. Snapping placement of x and y positions. These could now act similar to the placement on a real note sheet
7. Note placement rules according to music theory
8. Ability to drag notes

9. Play the song using a MIDI player

As we continued to develop, we found that a lot of the decisions we made in the past (also copying from ChatGPT) caught up with us. We tightly coupled classes and relied on hard coded values which made it difficult to modify existing code.

We decided to refactor the grandstaff, measure, and note classes. This removed a lot of the functionality we already had but made it easier to create new features. We pushed the tracking and adding of notes into the measure class itself (grandstaff doesn't need to know about that). We also changed the click events so the measures themselves are separate components and no longer work with absolute coordinates.

With this new code we reimplemented all of the previous features (except note placement rules) and added some new ones.

1. A note shows when you are hovering your mouse so you see where it will be placed ahead of time
2. Notes can now play their sound (rudimentary)

We have a very quick implementation of the sound features. It has already shown that it will be a little confusing to implement.

We started by having the measure play a sound for each note in a measure. The outcome was a mess and lagged the entire program. Every time a note was placed, the sound system would have to start up and configure the sound needed for the note.

We also had synchronization issues with the notes being played at random and the same times. This left us to try compiling the song when it needs to be played instead of on note placement. We then delegated the toolbar play button to get the song from the grandstaff, which would get the notes from its measures.

The best way to play the song was passing an entire list of frequencies and note durations to the sound system so it can control when to play the notes. We foresee changing the sound of notes (since they are just frequencies) but this method is a good starting point.

We have now finished the project and we are happy with the progress that has been made. We ended up changing the way we play music by using Java MIDI. This is built for playing music based off of pitches and durations. This worked perfectly with our design, the only problem was figuring out what pitch to play based on where we placed a note.

We found the best way to store notes and their locations was with a hashmap. The key was a normalized X location from 0 to 1, indicating location within a measure. This differs from the location stored within a note class. The value was an array of notes at that X location. This

hashmap could be passed bottom up from the measure class, to the grandstaff class, and finally to the UI class. At each step, we would update the positions to account for different measures and rows. It made it incredibly easy to work with and could be easily interpreted by our MIDI player.

The MIDI website has a very nice conversion of MIDI units to notes on a piano. We originally plotted the points on a graph to try and figure out the conversion but this failed due to the black keys being non consistent. All we do is add each note and their index to MIDI's sequence and it plays the song perfectly, even chords!

The song that's being passed to the UI to play even works across multiple grand staffs

Testing Strategies

The testing strategy involves both unit testing and functional testing:

- **Unit Testing:** Verify the functionality of individual components, such as the `NoteButtonListener` and the methods of storing notes with their position.
- **Functional Testing:** Evaluate the overall functionality of the application, including note placement on the grand staff and playback.
- **User Acceptance Testing (UAT):** Involve end-users to ensure the application meets their requirements and is intuitive to use.

Conclusion

The Music Notation Editor is a comprehensive application catering to the needs of musicians and composers. By offering essential features in an organized and user-friendly manner, it provides a valuable tool for digital music composition. The utilization of layout managers, event handling, and proper MVC architecture ensures a robust and maintainable codebase. Testing strategies encompass unit, functional, and user acceptance testing to ensure the application's reliability and user satisfaction.

Chapter 2: User Manual

Music Notation Editor User Manual

Placing Notes

To place notes on the grand staff:

1. Click on a note in the menu to select it.
2. Hover your mouse over the desired location on the grand staff, a hovering note will appear showing you where the note will be placed.
3. Click to place the note.

Notes can be stacked but not placed in the exact same location. Ensure your notes adhere to the time signature of the song.

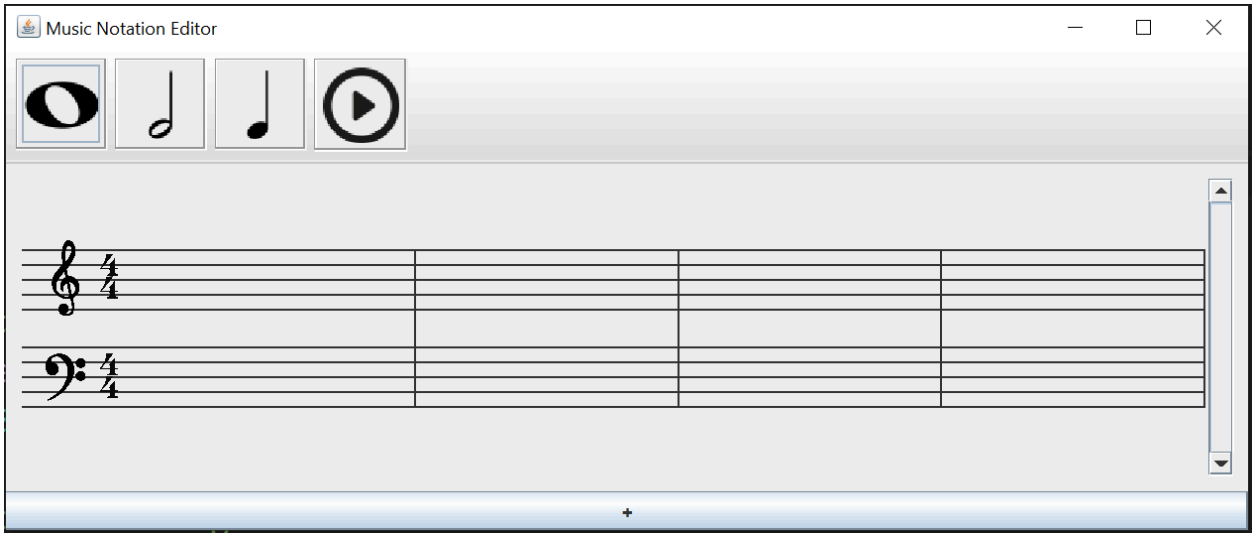
Editing Notes

- **Deleting Notes:** Right-click on a note to delete it.
- **Adjusting Note Positions:** Click on a note, drag it to a new location, and release to adjust its position.

Toolbar Buttons

- **Whole note** - click to select the whole note
- **Half note** - click to select the half note
- **Quarter note** - click to select the quarter note
- **Play:** Start playing the song from the playback head.

Chapter 3: Design



Design Manual: Music Notation Editor

1. Architecture Overview

The Music Notation Editor follows a layered architecture, with clear separation of concerns between presentation, application logic, and data access layers. The architecture can be summarized as follows:

- **Presentation Layer:** Responsible for rendering the user interface and handling user interactions. Components include Swing GUI components such as JFrame, JPanel, and JToolBar.
- **Application Layer:** Implements the core functionality of the music notation editor, including parsing, rendering, and editing music scores. Components include Measure, Note, GrandStaff, and UI classes.
- **Data Access Layer:** Not applicable in this project as there is no external data source. All data is generated and manipulated within the application.

2. Design Patterns

The Music Notation Editor utilizes several design patterns to enhance maintainability, extensibility, and reusability:

- **Model-View-Controller (MVC):** The Swing-based user interface follows the MVC pattern, separating the presentation (view), application logic (controller), and data (model) layers.

- **Factory Method:** Used in the creation of Note objects based on user input or other application events.
- **Observer:** Utilized in the event handling mechanism, where components such as GrandStaff observe changes in Note objects and trigger updates accordingly.

3. Component Descriptions

- **Measure:** Represents a single measure of music notation, containing a collection of Note objects. Responsible for rendering the measure, handling user input, and managing notes within the measure.
- **Note:** Abstract base class for different types of musical notes (e.g., WholeNote, HalfNote). Defines common properties and behaviors for all note types, such as position, duration, and graphical representation.
- **GrandStaff:** Composite component representing a grand staff, consisting of two staves (treble and bass) arranged vertically. Manages the layout and rendering of multiple measures and handles user interactions at the staff level.
- **UI:** Facade class responsible for initializing and configuring the user interface components, including menu bars, toolbars, and main application window.
- **MIDIPlayer:** Provides functionality for playing MIDI notes and sequences within the application. Responsible for managing MIDI playback, including the creation and manipulation of MIDI sequences, addition of notes to sequences, and control over playback.

4. Standards and Conventions

- **Coding Standards:** Follows industry-standard coding conventions, including consistent indentation, meaningful variable names, and adherence to Java coding conventions.
- **Naming Conventions:** Classes, methods, and variables are named descriptively and follow camelCase naming conventions. Package names follow a logical hierarchy based on functionality.
- **Documentation:** Comments are used to provide clear descriptions of purpose when the code is unclear.

The Music Notation Editor's design emphasizes modularity, flexibility, and usability, making it well-suited for further development, maintenance, and extension. By adhering to established design principles and patterns, the application achieves a high degree of cohesion and low coupling, facilitating ease of understanding, modification, and enhancement by developers.

Class Descriptions:

1. **App:**
    - Description: The main entry point of the Music Notation Editor application. Extends JFrame to create the main application window.
    - Responsibilities:
        - Initializes the application window with a title, size, and close operation.

        - Initializes the user interface components by instantiating the UI class.
    - Collaborators: UI
2. **UI:**
    - Description: Facade class responsible for initializing and configuring the user interface components, including menu bars, toolbars, and main application window.
    - Responsibilities:
        - Initializes the menu bar, toolbars, and main application window.
        - Configures event listeners for user interactions.
        - Manages the addition and layout of GrandStaff components within the main application window.
        - Handles playback of music scores.
    - Collaborators: App, MenuBar, ToolBar, GrandStaff
3. **ToolBar:**
    - Description: A toolbar containing buttons for selecting note types and initiating playback of music scores.
    - Responsibilities:
        - Creates and configures buttons for selecting note types (whole note, half note, quarter note) and initiating playback.
        - Registers event listeners to handle user actions on toolbar buttons.
        - Scales and displays icons for toolbar buttons.
    - Collaborators: JButton, ImageIcon
4. **GrandStaff:**
    - Description: Represents a grand staff consisting of two staves (treble and bass) arranged vertically. Contains multiple measures of music notation.
    - Responsibilities:
        - Manages the addition and layout of Measure components within the grand staff.
        - Handles user interactions at the staff level, such as adding new measures.
        - Provides methods for retrieving the musical score represented by the grand staff.
    - Collaborators: Measure
5. **Measure:**
    - Description: Represents a single measure of music notation within a grand staff. Contains a collection of Note objects.
    - Responsibilities:
        - Renders the measure, including clefs, notes, and other symbols.
        - Handles user input for adding and deleting notes within the measure.
        - Manages the collection of Note objects and provides methods for retrieving them.
    - Collaborators: Note
6. **Note (and its subclasses - WholeNote, HalfNote, QuarterNote, etc.):**

- Description: Represents a musical note within a measure. Defines properties such as position, duration, and graphical representation.
- Responsibilities:
  - Provides a graphical representation of the note on the staff.
  - Stores metadata such as pitch, duration, and sound information.
- Collaborators: None

7. **MIDIPlayer:**
   - Description: Central component responsible for managing MIDI playback within the Music Notation Editor application. Facilitates the creation, manipulation, and playback of MIDI sequences, enabling the rendition of musical compositions.
   - Responsibilities:
     - Sequence Management: Handles the creation, modification, and playback of MIDI sequences, ensuring seamless playback of musical scores.
     - Note Addition: Provides functionality to add individual musical notes to MIDI sequences, specifying their pitch, duration, and velocity.
     - Playback Control: Enables control over MIDI sequence playback, including initiation, pausing, stopping, and resetting playback.
     - Resource Management: Ensures proper handling of MIDI resources, such as sequencers and tracks, to optimize memory usage and prevent resource leaks.
   - Collaborators: None (MIDIPlayer functions independently within the application, interfacing with other components as necessary for MIDI playback functionality).

Each class in the Music Notation Editor plays a specific role in the application's architecture and contributes to its overall functionality. Collaboration between these classes enables the creation, manipulation, and playback of musical scores within the user interface.