Impromptune

Design Document

Team 5

Ben Ahlbrand

Chris Doak

Sean Phillips

Jacob Richwine

1. Purpose

Impromptune is a Java based application that allows users to algorithmically generate music. Users can manually input their own motifs and generate music based off of the input and chosen parameters. The music is displayed as an editable page of sheet music, and users also have the capability to playback their creations.

The system will be divided into three distinct key components:

1. Graphical User Interface
    a. Static GUI: Display input controls for creation parameters
    b. Dynamic GUI: Display, as sheet music, the current task and update when changes are made
    c. Display drop-down menus
2. Analyzer and Generative Algorithms
    a. Analyze current selection
    b. Process generative input parameters
    c. Create accompaniment algorithmically
3. Manual Input and Other Program Logic
    a. Handle creation input from GUIs
    b. Handle playback controls
    c. Handle Open, Saving Tasks

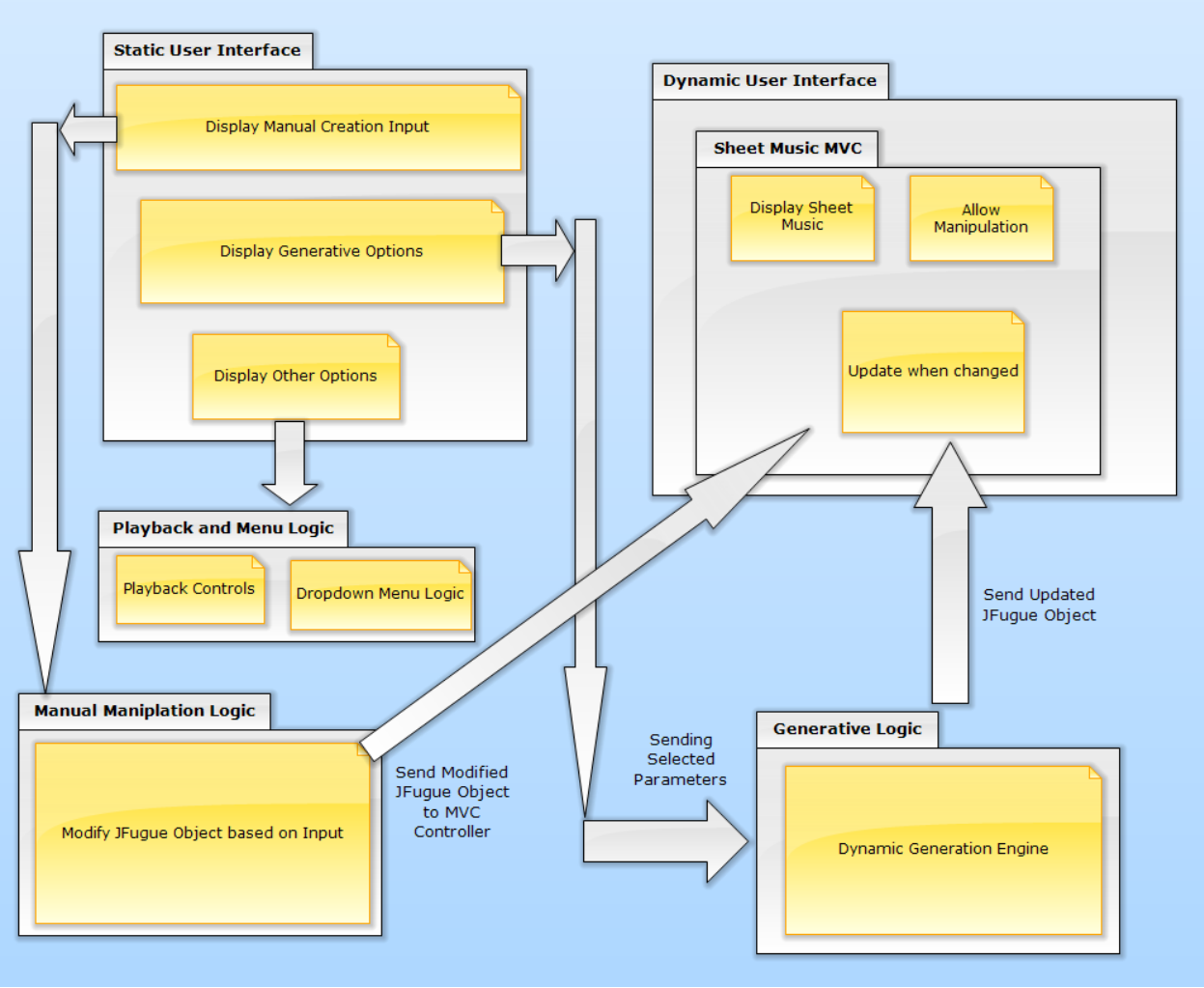
From a user perspective, the purposes will be:

1. Allow user-friendly input for music creation through use of on-screen piano and easily visible selectable options
2. Allow playback of entire composition or selected bars
3. Allow user to create a few bars of music and generate different voices based on chosen parameters
4. Allow user to save composition for later use
5. Allow user to view results immediately as user input is made

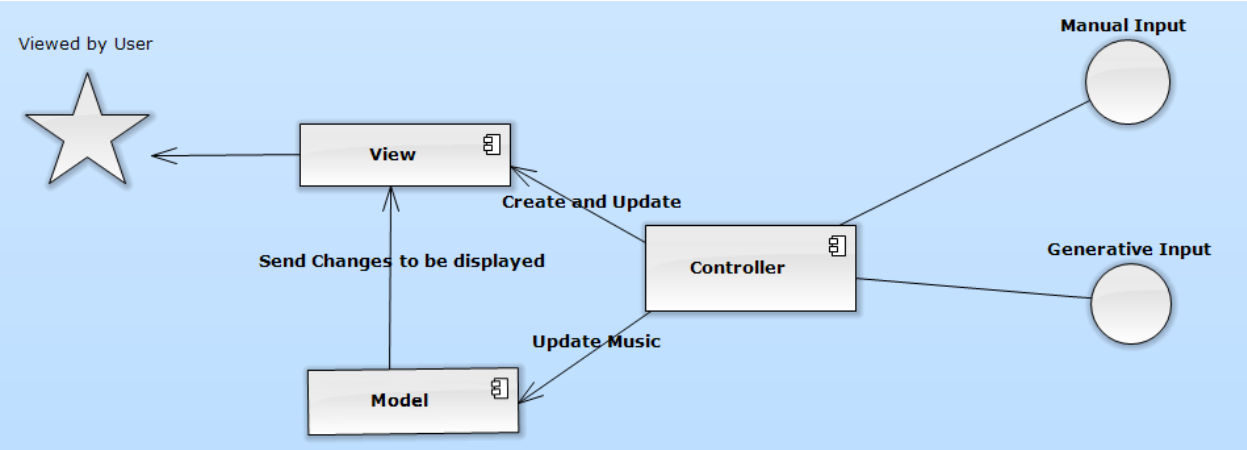
2. Design Outline

Impromptune will incorporate multiple architectures spread over different components. Overall, we will be using a layered system, dividing the program into the GUI layer and Logic Layer. Each layer will also be split up into different sections. For the GUI layer, there will be a static GUI that displays controls and manual input options and a dynamic GUI that displays the current sheet music and updates it when necessary. For the dynamic GUI, we will also incorporate a Model-View-Control architecture to handle input, update the user's view, and change the underlying model. For the logic layer, we will have the manual input logic and dynamic generation logic.

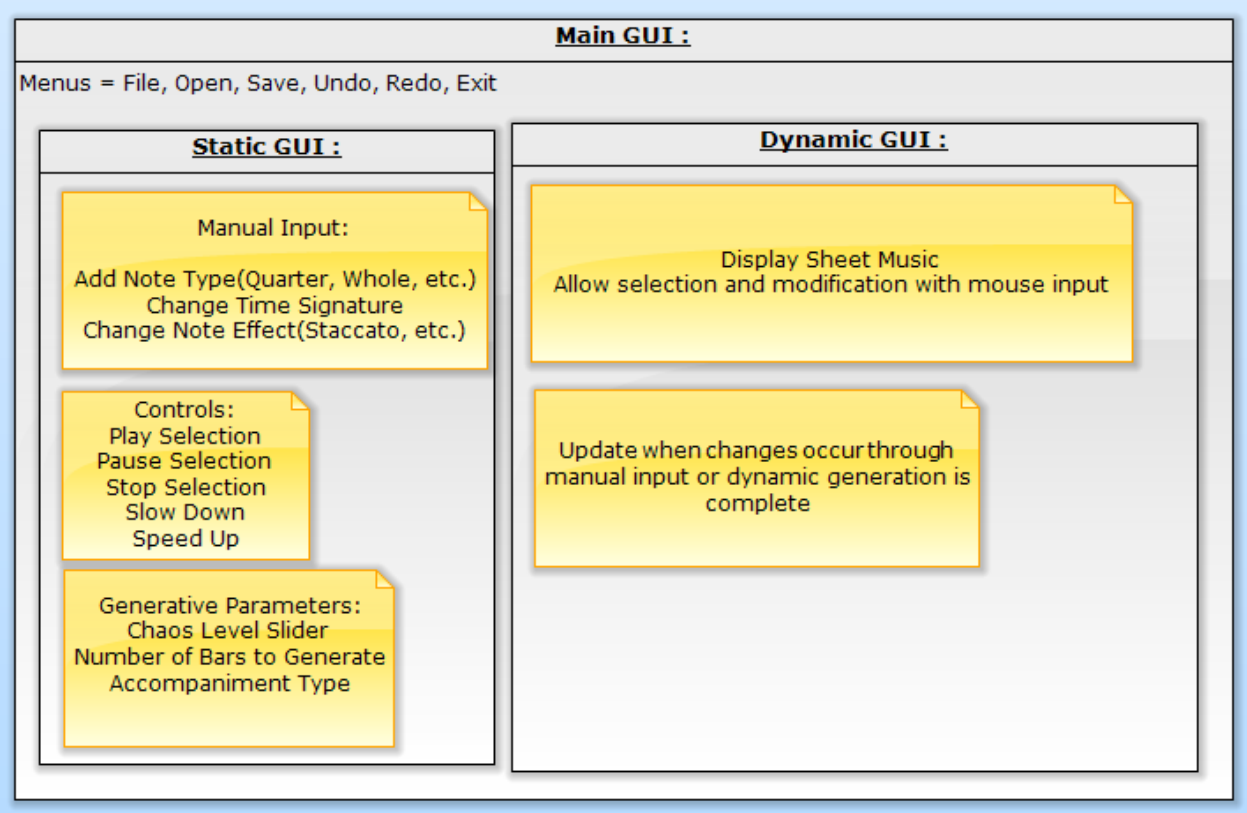
Overall Layered Architecture



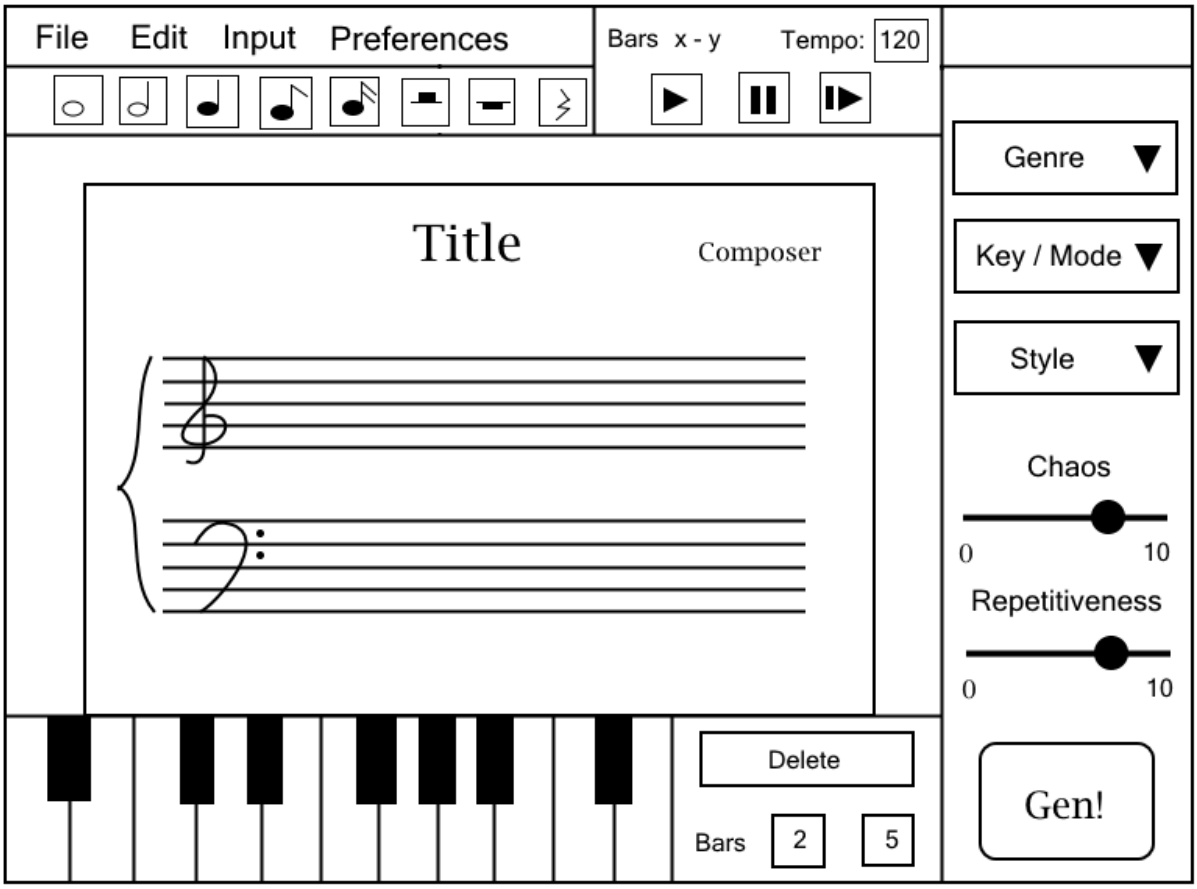
Model View Control Layout



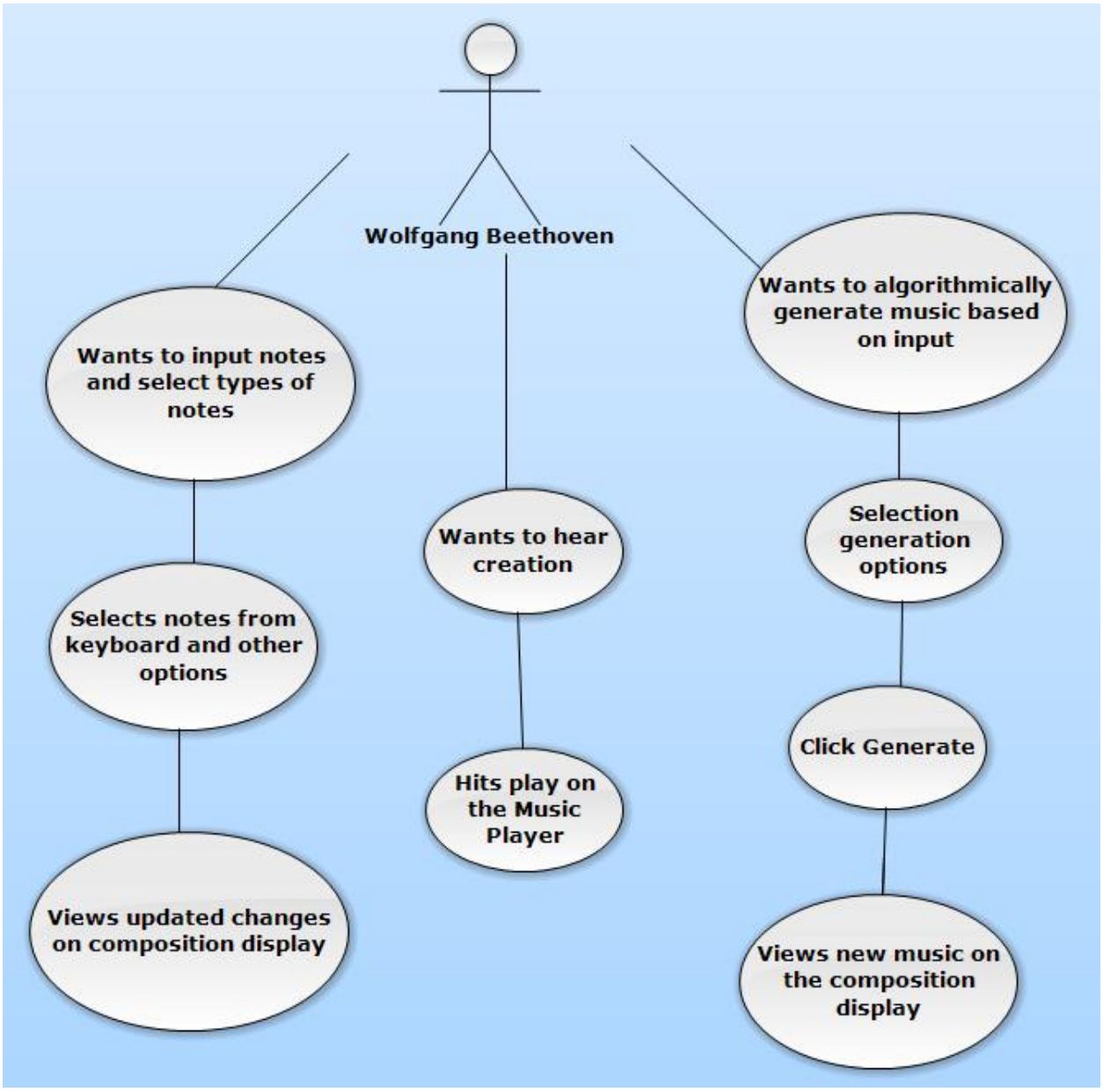
GUI Component Overview and Summary



GUI Mockup

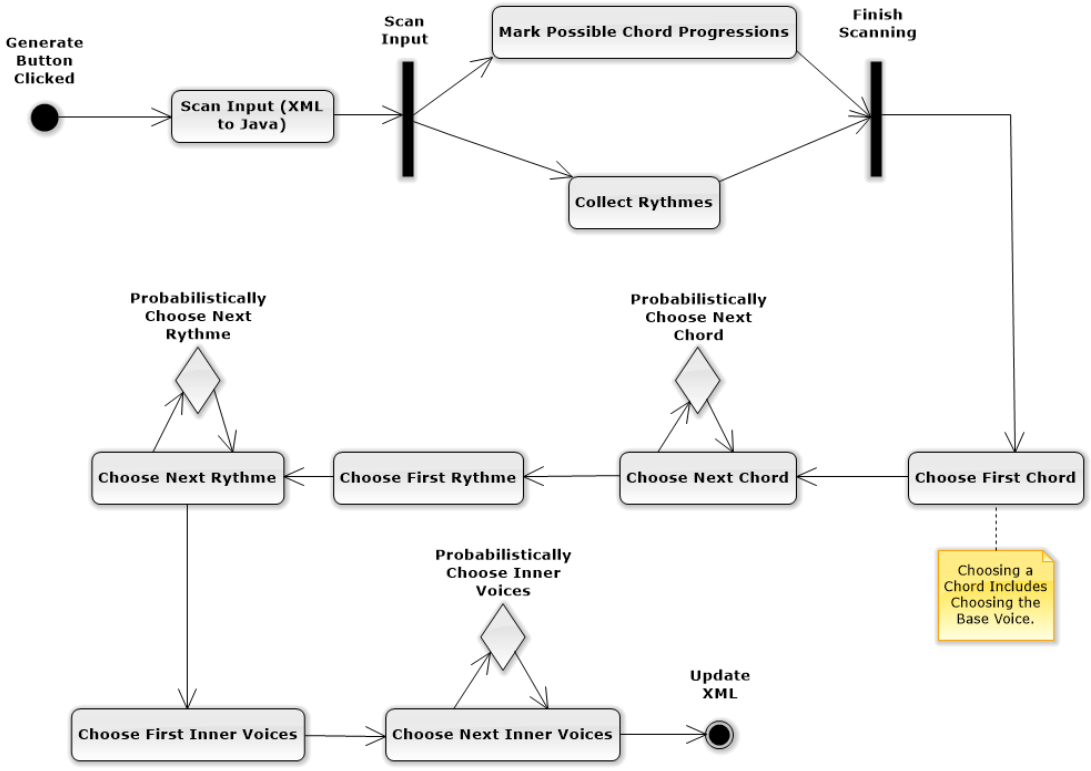

File
Edit
Input
Preferences
Bars x - y
Tempo: 120
Title
Composer
Genre
Key / Mode
Style
Chaos
0
10
Repetitiveness
0
10
Delete
Bars
2
5
Gen!

Sample Use Cases / Sequence of User Actions



Music Generation Model

The Composer will, provided an input being either an entire composition or chosen sections, generate accompanying voices. The Scanner component will parse this input, tagging as it goes, marking desired features of the input. In essence, the Scanner would build two structures: tonal and rhythm. On the tonal track, possible chord progressions - which will allow the Composer to infer (but not calculate at this phase in the pipeline) the following features: scales, arpeggios, the most likely roots of the above tonal features, harmonic/melodic motion, etc. Next, the Scanner will scan the rhythms, tagging repeating subsequences, which would allow consideration of where potential phrases could begin and end. For example, at the end of a measure on the middle beat of a triplet-sixteenth, it's fairly unlikely you're beginning a new arpeggio or some other defined phrase type. After parsing, the Composer will build a Markov model around each section, and probabilistically determine the following chord (and other dimensions as development progresses), and continue choosing the following chords until the phrase is processed. The Composer would then do the same for the rhythms. Based on the progressions and rhythms, the Virtuoso of the Composer generates complementary inner voicing as accompaniment. Below is a pipeline architecture diagram of the Composer Component, taking user input and returning an updated object.



3. Design Issues

Issue #1: *Which development platform should we use?*

Option 1: Java

Option 2: .Net

We chose option 1 due to more familiarity within the team and the availability of libraries for dealing with manipulating, creating, and storing music.

Issue #2: *Having decided on Java as the main development platform, which GUI platform will we use?*

Option 1: Swing

Option 2: AWT

Option 3: JavaFX

We chose option 3 due to it being the latest and greatest GUI platform and it has all the utility for our needs. It also allows the use of Cascading Style Sheets to design and apply to the GUI elements. Our team is familiar with creating Style Sheets for websites and can apply the same knowledge to the JavaFX layout. Also, the latest version of IntelliJ has built-in compatibility for manipulating JavaFX components.

Issue #3: *Should we try to create libraries from scratch or utilize pre-existing ones such as MusicXML-Viewer, Zong! and JFugue?*

Option 1: Write music handling libraries from scratch

Option 2: Utilize pre-existing libraries

We chose option 2. Obviously utilizing pre-existing libraries is a major advantage, but we also took into consideration learning the libraries and being cautious about them not having features that we would need to use or having compatibility issues further into development. Writing our own would have the specific features we would need but would be a monumental task and lengthen development complexity and time considerably.

Issue #4: *How will we be storing the work done and created within the program?*

Option 1: Use pre-existing format like MusicXML

Option 2: Use scratch created file format

Option 3: Use a hybrid format that includes MusicXML along with our own needed data

We chose option 1 because a MusicXML file will be the central data model for the entire project. Most actions will revolve around the data stored in the MusicXML file. Other necessary data can be created and used at runtime.

Issue #5: *Should we display multiple compositions on one screen?*

Option 1: Multiple open projects, same screen

Option 2: Tabbed open projects

Option 3: One open project limit

We chose option 3 to simplify the project, but may decide to implement a tabbed GUI to allow multiple compositions open at once if time allows.

Issue #6: *How should we display and edit the composition in terms of visual notation?*

Option 1: Common track based UI, more in the spirit of GarageBand / ProTools

Option 2: Display the piece in traditional music notation with staves and notes.

Option 3: Use a music sequencer similar to FL Studio

We chose option 2 because it is a well-established and standardized format. Users in our target demographic will be very familiar with how it works. This format is also more flexible as it can display many different types of rhythms and styles. It is also easier to render given the fact that we are storing our music in MusicXML format.

Issue #7: *Performance of accompaniment generation*

Option #1: Markov Model

Option #2: Harmonic State Searching

Option #3: Feature extraction combined with building Random Forest

Based on the assumption that however the Virtuoso engine works, the need for analysis of the music to be able to generate any music means that there will be a balancing act between performance and interesting/complex accompanying voices. To this end, as opposed to a complex harmonic state history searching algorithm, probabilistic models based on Sections or groups of bars will provide a performance feasible way to meet our goals. Between probabilistic modelling techniques, research shows Markov Models and Random Forests will produce similar results, though a strength of Random Forest is its ability to relate interdependent dimensions (consider tones and rhythm simultaneously with more accuracy). Markov chaining will have better performance, since many trees will need to be built for robust predictions. However RF has other strengths such as fuzzy decision generation models, which could provide a more human quality, but by the same token, could go that much more wrong in the other direction (unpleasant random noise). Also, presumably something of the human quality is present in the provided input for the Composer. Markov chaining more closely maps itself to our problem, given its nature and the fact that we'd be performing calculations on a relatively small set of musical bars at any one time. Sliding windows of notes meshes naturally with Markov models. Thus, between this and given the overhead of other approaches, Option #1 seems to be the logical choice.

Issue #8: *What libraries and how do we plan on implementing them into the project?*

Option 1: MusicXML, Zong!, JFugue version 4

Option 2: MusicXML, JFugue Music NotePad, JFugue version 5

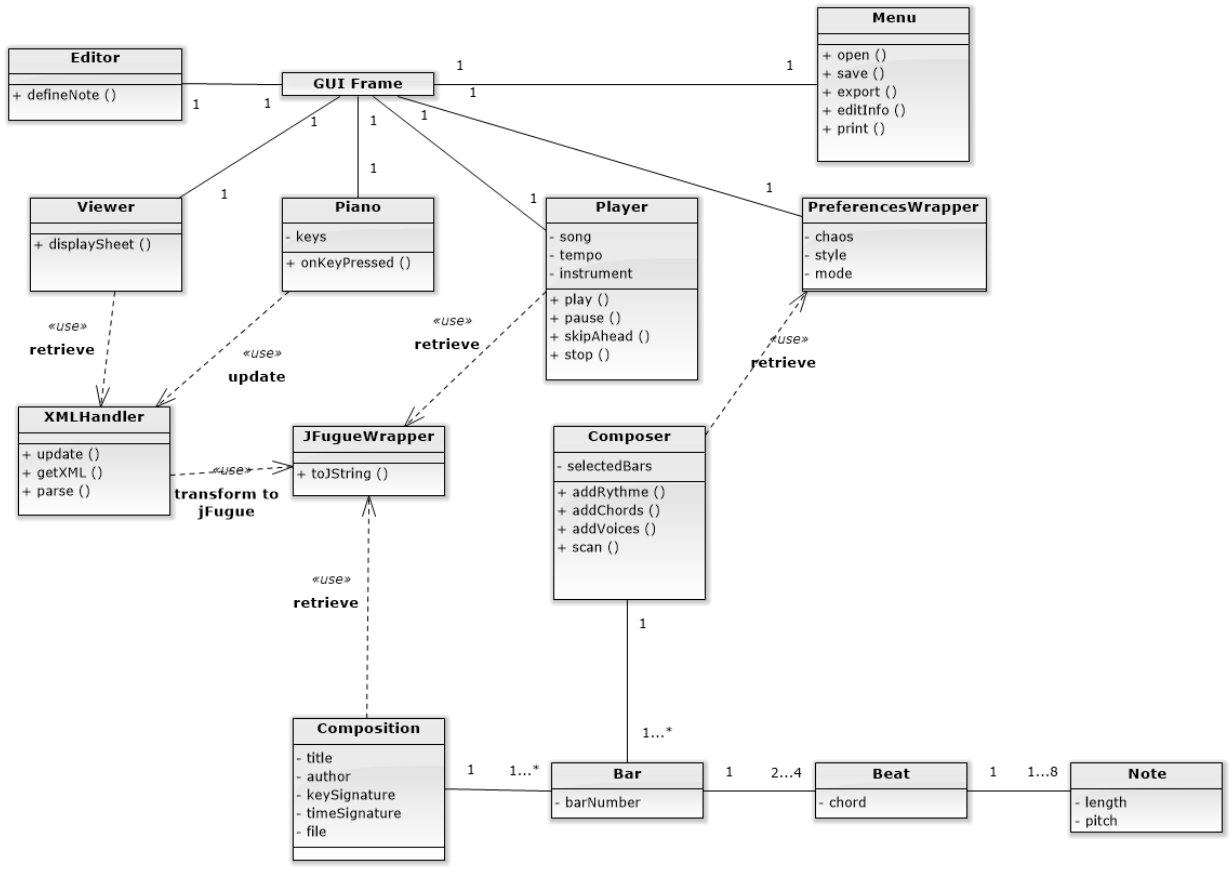
Option 3: Other combinations of libraries

We chose option 1 as the three major libraries that are utilized in Impromptune. MusicXML is a well-developed and mature file format standard and is compatible and well documented. Zong! is a currently developed Music Note Renderer which takes a MusicXML file and displays the notation as sheet music. We chose JFugue 4 over version 5, as version 5 is in beta, and may be unstable and not as fully featured as version 4. Also version 4 provides MusicXML input/output from its internal MIDI based structure.

Our program will read in a MusicXML file, which will then be passed to Zong! to render the file into visible sheet music. After modifying or creating a new composition, the manual input logic will pass the changes to JFugue, which will then allow the playback of the composition, and process the changes back into a MusicXML file. The dynamic generator will take in a JFugue object, manipulate it, and output a new JFugue object, to once again be parsed in MusicXML for Zong! to display.

4. Design Details

Class Diagram



Class / Layout Descriptions

GUI Main Frame:

- Menu Bar
  - Open/Close
  - Save/Save as
    - Project
    - Pdf
  - Edit meta-information
    - author
    - title
    - credits/copyright

        - date
        - comments
    - Print
- Player
    - Play, stop, and pause playback
    - Adjust the tempo
    - Define playback position
        - Skip ahead X bars
        - Go back X bars
    - Instrument/Synthesizer selection
        - Individually
        - Set
- Editor
    - Allow interaction
    - Frame that contains manual input options
        - type of note
            - sixteenth, eighth, quarter, half, whole, etc notes
            - triplet
        - time signature
        - key signature
        - tempo (for playback, and notation)
        - tie notes
        - insert chord
        - repeat
            - open repeat
            - close repeat
            - alternative repeats
- Viewer
    - Display composition
- Parameter Frame
    - Sends parameters to composition class for generation
    - Can change values from within
- Piano Keyboard Frame
    - A GUI keyboard for selecting which note to insert
    - Allows users to click which note to insert

Composition:

- Bars - set of Beats
    - Beats - set of Notes
        - Notes:
            - tone
            - length
            - dynamics

- Metadata (unique fields)
    - author
    - title
    - time signature
    - key signature
    - credits/copyright
    - date
    - comments

JFugue Wrapper:

- Acts as the in-between for the XML and Java data structures
- Can turn retrieved XML elements into JFugue classes or MIDI data
- Can turn data structures back into jFugue music strings

XML Handler:

- Will house the XML data
- Will handle any interactions between the XML and other processes such as update the XML upon request
- Can retrieve desired XML data

Composer (generative algorithms):

Corpus (selected bars):

- References the JFugue MIDI data
- Will return the Virtuoso's generated data
    a. calls the XML update event,
    b. sends JFugue output to MusicXML
    c. re-render the view with the generated accompanying voices

Scan:

- Extracts relevant data for the Virtuoso

- Performs feature selection on extracted data augmented with:
    - Predefined parameters (system constants)
    - Provided optional user parameters (such as tonal roots, genre, modes/keys, styles)
    - Composition key/time signature
- Decomposes into the relevant layers for analysis (such as tonal strings, rhythms, and longest repeating subsequences)

Virtuoso (Fuzzy Constraint Satisfaction Optimizer):

- This will be the AI of the composer engine
- Performs a Markov process based algorithm to consider each group of bars
- Builds the following layers as Markov chains based on Scan extraction:
    - Rhythm
    - Chords & other tonal features
- Generates inner voices based on above
- Validates layers' weighted compatibility and optimizes based on given parameters from input
- Returns generated music data back to the composition via the XML handler
    - Displays suggested accompaniment greyed out or some other intuitive mechanism implying to user the distinction

Tools and Libraries

Tool/Library	Version	Source	Description
Java	8 Update 31	http://java.com/en/download/manual.jsp	Development Platform
JavaFX	8	http://www.oracle.com/technetwork/java/javafx/overview/index.html	GUI Library
JFugue	4.0.3	http://www.jfugue.org/	Sound Library
MusicXML	3.0	http://www.musicxml.com/	File format
Zong!	0.1-a.70	http://www.zong-music.com/	Notation Renderer