Idea Management for Composers

Independent Study Thesis

Presented in Partial Fulfillment of the Requirements for
the Degree Bachelor of Arts in the
Department of Computer Science and the Department of
Music at The College of Wooster

by
Grace Simonson
The College of Wooster
2025

Advised by:

Dr. Daniel Palmer (Computer Science)

Greg Slawson (Music)

THE COLLEGE OF
WOOSTER

Abstract

A challenge many composers face during the process of writing music is managing their early ideas. Not only does inspiration strike in a variety of places, but ideas themselves come in a myriad of forms: a melody, a texture, an image, words, rhythms, sounds, etc. However, current composition applications focus on notating music to create performance-ready sheet music and lack the ability to record early, inexact ideas. Utilizing knowledge of design principles, interviews with active composers, and research on the composition process and composition tools, I designed and implemented a software tool that allows composers to store, edit, and combine their early music ideas in one workspace. When testing the tool, composers enjoyed the ability to see all their ideas at once and the ability to switch quickly between different idea formats (which may be represented as notated music, drawings, images, audio recordings, or text in the workspace) and found that these features aided the creative process.

This project is dedicated to all composers who have had to fight with technology to express their ideas.

Acknowledgments

A project like this could never have happened without the help of the many people who kept me focused, gave me insight, and provided feedback throughout the process. First, I'd like to thank my two advisors: Dr. Daniel Palmer (from the Computer Science department) and Professor Greg Slawson (from the Music department). They helped me focus my ideas and plan the project start to finish and also provided so much help and insight into the process of doing an I.S. like mine (from coding a huge project to writing a thesis to setting realistic expectations). This project never would have moved past the brainstorming stage without their help.

I am not a composer myself, so I am extraordinarily grateful to all the composers who gave me their time and effort to make this software be the best it could be: Gracelyn Jack, Greg Slawson, Dr. Dylan Findley, Dr. Cara Haxo, Dr. Peter Mowrey, Ethan Yoder, Daniel Damiano, Colin Schrein, and Robert Thomson. All took time out of their busy semesters to help me with ideas for the software, tell me about the composition process, and give feedback on the tool I created. A very special thank-you goes to Gracelyn Jack, who was my first supporter of the project, came up with the idea for a drag-and-drop workspace, and provided invaluable feedback on the final product.

Finally, thank you to all my friends, family, and professors who helped me in a myriad of ways through a very busy year. Without your support, I never would have made it this far. Thank you for keeping me sane and listening to my I.S.-related ramblings!

Vita

Fields of Study Major field: Computer Science

Major field: Music

Specialization: User Interface Design, Composition

Contents

List of Figures

CHAPTER 1

Introduction

Being able to quickly record and retrieve ideas when composing music is important but tricky, since ideas and inspirations for music come from many different places. The ideas themselves vary in format–sometimes they are fully fledged music ideas, other times they are simply a texture or a rhythm or even a few words. However, most composition applications come in the form of music notation software–powerful for engraving sheet music to give to musicians to perform, but inflexible in its input. Notation software enforces the rules of music notation, forcing the composer to input exact pitches and rhythms to record ideas. For this reason, composers use a wide variety of tools to record their early ideas: voice recorders, staff paper, text editors, blank paper, sticky notes, or anything else that allows them to draw, sing, play, or write down an idea. When it comes time to flesh out those ideas into a full piece of music, the variety of formats makes it difficult to find, edit, and combine ideas into fuller compositions.

The goal of this software is to give composers a place to put their early music ideas for easy retrieval, editing, and combining: an idea management tool. Having many ideas in various formats in one place can facilitate creativity by allowing composers to see their ideas at the same time, develop those ideas, and discover new ones. Once ideas are developed to the point of having the pitches, rhythms, and other properties required by notation software, they may be exported to the composer's notation software of choice.

This software does not aim to replace all the tools composers use, rather, it provides a starting point for fresh ideas and a way to transition from early ideas to notation software.

CHAPTER 2

Background

This chapter discusses existing research on the composition process and current composition tools as well as interviews I conducted with composers about what features may be helpful in an idea management tool.

2.1 The Composition Process

The process of composing music is highly subjective and varies person to person. However, common trends and processes exist across the community. Through interviews with eight composers, Stan Bennett (at the Institute for Child Study at the University of Maryland) formulated six basic stages to the composition process: germinal idea, sketch, first draft, elaboration and refinement, final draft copying, and revision [1]. The germinal idea is the originating idea or concept for a piece of music, and can be a melody, chord progression, texture, outline, etc. Bennett explores the environment that may facilitate the development of a germinal idea and discovered that it varies greatly. A germinal idea may come to a composer while they are daydreaming, driving, taking a shower, walking down the street, or sitting in silence. A germinal idea may also be quickly forgotten if not recorded quickly (depending on how potent the idea is). The sketch is the germinal idea written down in some way (it may be notated, recorded, drawn, written about, etc.), which leads into the more fleshed-out first draft. Bennett notes there is a cyclical relationship between the

germinal idea, sketch, and draft, since working on any of these stages tends to spawn new germinal ideas (which in turn may develop into further sketches and first drafts). The elaboration and refinement stage further develops and adds to the first draft. After the final draft copying stage, the piece is ready to be performed. The final stage, revision, occurs after the performance of a piece and is less common. Bennett points out that many composers will leave a piece after it is performed to move on to a new project, skipping the final revision stage [1]. Bennett's classifications of each stage provide a useful framework when it comes to discussion the composition process.

The goal of the software developed here is to aid in the beginning of the composition process–recording the germinal idea and developing sketches of a piece of music. Emilia Rosselli Del Turco and Peter Dalsgaard (both at the Centre for Digital Creativity in Aarhus, Denmark) take a deeper look into these early composition stages by researching the question "How do music artists capture and manage their ideas?" [14]. When it comes to capturing musical ideas, Del Turco and Dalsgaard found that composers notate music (either by hand or in notation software), record themselves playing or singing the idea, write textual descriptions (or record verbal descriptions), draw graphics representing the idea (or find pictures that inspired the idea), and record environmental sound. The collection of ideas and sources of inspiration is known as an "idea archive," from which ideas may be organized, retrieved, and combined. However, due to the broad variety of formats and tools used in capturing ideas, an idea archive is not stored in a single location. This leads to difficulty in retrieving ideas when they are needed. Based on the results of the study, Del Turco and Dalsgaard offer three design recommendations for an idea management tool: the tool should have indexing functionality for easy retrieval of ideas, the tool should allow for various idea formats to be integrated into one environment, and the tool should allow composers to get as far as possible in the creative process (for example, allowing them to at least have a first version of a melody when starting from a germinal idea) [14].

2.2 Research on Composition Tools

Current composition tools focus on later stages of the composition process. Music notation software (such as MuseScore, Dorico, and Sibelius) aid in the final draft copying stage (by enforcing music notation rules and providing an easy way to generate performance-ready sheet music) and the elaboration and refinement stage (by allowing the user to easily edit notated music).

While notation software may be used in the early stages of composition, there are some challenges. By enforcing music notation rules, flexibility is limited and composers may have difficulty accurately recording their early ideas [3]. To input music into notation software, the composer must know the rhythm (and by extension, time signature) and exact pitches of an idea. However, a germinal idea may be only a rhythm, or only a melodic contour, or only the texture or instrumentation of a piece. Music notation software is less optimal for recording these types of ideas. Paper, whether staff, blank, or graph, allows composers the flexibility to record such ideas. However, to take advantage of notation software's engraving and styling strengths, the composer must spend time transferring ideas on paper into notation software [3]. This adds time and mental strain as composers must decide how far to take an idea on paper before transferring it to the computer.

For this reason, much research focuses on integrating paper (or the freedom of paper) into composition software. Garcia et al. developed and tested Polyphony, a composition software tool that integrates interactive paper, a piano keyboard, and inputs from a computer mouse and keyboard with composition software. In the study, 12 composers were given an hour to complete a specific composition task (writing an accompaniment to an existing piece of music). The various tools offered by Polyphony were used at different points in the composition process. The tools related to drawing were used more frequently in the early "ideation" stages of composing while the mouse and keyboard were used in later stages to refine ideas and set precise values. Many also utilized the copy-paste functionality of the mouse (which was not a feature of any other tool). There were two

general trends in how the composers adopted Polyphony into their composition style. Some improvised on the software, refining ideas depending on the software's capabilities. Others planned out the end result before moving to the software the implement their planned ideas. All enjoyed the immediate feedback (playback) offered by the software and the fact that the various input devices were synced together (presumably offering quick transitions from one format to another) [5].

2.3 Requirements Gathering: Composer Interviews

In addition to looking into existing research on the composition process, I had conversational interviews with seven composers to learn more about the composition process, tools composers use, and what features would be helpful to them in an idea management tool. Background research and interviews comprise the stage of the design process called requirements gathering. This is when the designer learns as much as possible about the goals of potential users, how they achieve these goals, and what improvements may be made to that process. Existing research on the composition process was a good place to start, but actually talking through both the composition process and what my tool could look like with composers greatly impacted the design, features, and functionality of the final product.

All of those I interviewed are currently active composers, a mixture of students and professors. All are either currently studying or teaching composition. There was also a mixture in compositional styles–some write notated music to be performed by musicians while others produce electronically mixed recordings or jazz charts. While each interview varied slightly (due to their conversational nature), each of the following topics were discussed:

- The composer's usual composition process
- Methods the composer uses to record early ideas

- The tools the composer uses to record musical ideas and the advantages/disadvantages of those tools
- Which features would be helpful to them in an idea management tool (focusing on the beginning of the composition process)

Based on the interviews, I generated a list of possible software features that would be helpful in the early stages of composition and organized them by popularity. When it came to designing and implementing my software, the features at the top of the list were implemented first.

There were two general sentiments common across most of the composers. First, they all found the idea of seeing all ideas at once (or having them present in the same space) to be helpful. Second, they wanted to be able to switch quickly between ideas and their formats. For example, the ability to go from notating music to entering text quickly would be helpful.

In terms of specific feature ideas, two were recommended by all composers: the ability to record audio (and play it back) and to enter text. Many composers also suggested being able to draw on images and notated music and to be able to play back any notated music entered into the application.

CHAPTER 3

Designing User Interface/User Experience

3.1 Principles of UI/UX Design

A well designed application helps a user fulfill their goal(s) as smoothly as possible. Once a user determines a goal, they plan (consciously or subconsciously) a series of actions to complete the goal. Finally, they evaluate the results of their actions to see if the goal was accomplished. Throughout the process, the application must communicate clearly with the user.

Don Norman explores how a system helps a user complete their goals in his 7 Principles of Design: discoverability, feedback, conceptual model, affordances, signifiers, mappings, and constraints [13].

3.1.1 Discoverability

Discoverability is what allows a user to determine what actions can be completed with the application, including the application's current state. Before an action is completed, and sometimes before a goal is even formulated, the user must be able to tell what actions are possible with an application [13]. For example, the communications app Discord has a small toolbar on its desktop application (seen in Figure 3.1) that shows the user a

few possible actions that may be done in the app [8]. As seen in the figure, when the

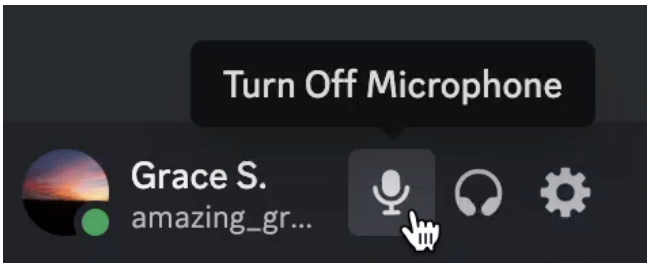

Figure 3.1: A toolbar in Discord's desktop application

mouse hovers over one of the labels, it is highlighted (showing a button) and a tooltip pops up to explain what happens when the button is clicked. From hovering over the microphone icon, the user learns that muting the microphone is possible (and presumably other microphone settings are adjustable as well).

Discoverability is linked to understanding (knowing what the controls and settings mean and understanding what the application's purpose is). This information can be conveyed directly through tutorials, manuals, and documentation, but often can be shown through the design of the application itself [13]. With the Discord example, the user can read the documentation or tutorials to learn how to change the microphone settings. The design of the toolbar, however, lets the user learn how to mute the microphone without needing a tutorial. Discoverability results from the remaining design principles below (affordances, signifiers, mappings, feedback, conceptual models, and constraints). A failure in one or more of these principles may lead to confusion or frustration on the user's part, as it hampers the discoverability of the application and prevents the user from successfully completing their goal [13].

3.1.2 Affordances

An affordance is the relationship between an object (in this case, the application) and an interacting agent (the user). For example, a chair affords sitting. A chair may also afford lifting. However, this is only the case if the chair is light enough and the interacting

agent is strong enough to lift it. Otherwise, the chair does not afford lifting. Affordances indicate possible interactions with an object or system. Conversely, anti-affordances prevent an interaction (for example, glass prevents passage of objects). Both affordances and anti-affordances must be perceptible or signaled to be useful to the user–if the user cannot tell what actions an object does or does not afford, they cannot use the object to complete a task [13].

3.1.3 Signifiers

A signifier communicates one or more affordances and/or appropriate behavior to the user. Signifiers show where an action must take place, as opposed to affordances (which show possible actions). Perceived affordances may act as signifiers–for example, a flat plate on a door affords pushing the door to operate it. However, perceived affordances are not always clear. If a door with a flat plate only opens by pulling, there is confusion. The plate signals a perceived affordance of pushing, so if the door does not push open, the user has received incorrect information. Signifiers give clear indications of where or how an action takes place [13]. For example, a date picker on an online input form has signifiers such as the label 'choose date' and a small calendar icon–together, they show the possible action (choosing a date) and where to do it (clicking the calendar icon). Signifiers must be perceptible, or the user never receives the information they may need to use the object.

3.1.4 Mapping

Mapping describes the relationship between the controls and the object(s) they control. When there is good mapping, the user should intuitively understand how to control and navigate the system, freeing their mind to focus on the activity at hand instead of how to use the system. Natural mapping is when the controls match the spatial layout of the object(s) they control, and is generally a good way to ensure intuitive mapping [13]. For

example, the arrow keys on a keyboard are laid out with the up arrow pointing up, the left arrow pointing left, etc. When the user presses one of the arrow keys, whatever the arrows are controlling moves in the direction of the arrow pressed. Grouping controls can also help in conveying mapping. Mapping conventions vary by culture. For example, when a control is a horizontal bar conveying an amount of something (such as volume or screen brightness), different cultures have different ways of interpreting the bar as full (conveying a higher number). Some think filling the bar left to right should raise the value and others think filling the bar right to left should raise the value [13].

3.1.5 Feedback

So far, the design principles discussed have pertained to communicating how and where an action should take place. Feedback is important for communicating information to the user after an action has taken place. Once a user tries something, feedback should immediately alert them to whether or not the action was successful and the current state of the system. Even if the system cannot fill the user's request immediately, the user must be made aware that the system received the request and is working on it [13]. For example, in the Mac OS, when the user selects an application to open, the app icon bounces in place to let the user know the app is currently booting up. If this was not communicated to the user, they might repeatedly click the app icon, thinking that something is not working. In some cases, repeating a command or request over and over again (when just once is needed) will cause issues.

Feedback must also be informative to be helpful for the user. Feedback containing too little or too much information is likely to be ignored. A short, annoying beep is not informative enough–does it mean a button has been pressed? Or a setting has successfully changed? Or does it mean an error has occurred? On the other hand, a lengthy text output after every tiny action, while informative, takes valuable time to read and is likely to be skimmed or ignored entirely. At best, feedback helpfully informs the user of the

current state of the system. At worst, feedback is annoying, distracting, or misleading [13]. Feedback can be as simple as displaying a message saying a form was successfully submitted or moving the text of a chat message into its own little bubble above the input field (to show the message was sent). Feedback is also helpful throughout the action process, not just after a task has been attempted.

3.1.6 Constraints

Constraints prevent the user from doing an undesired action. Physical constraints, such as a disabled button on a webpage, physically prevent the user from doing a specific action (pressing the button, in this case). Any physical constraints must be easy to see and interpret, otherwise may result in confusion or frustration [13]. If the disabled button does not have any of the usual signifiers to show it is disabled (such as the greyed out color or X-ed out cursor), the user will be confused when they try to press the button and nothing happens.

3.1.7 Conceptual Model

The conceptual model of a system is the user's understanding of how a system works. It is built from signifiers, affordances, constraints, mappings, and their own experience. Generally, the conceptual model for a system varies from person to person, as each one has their own interpretation of how something works. A user's conceptual model does not have to be entirely accurate or incredibly detailed to work–as long as it gives them an accurate understanding of the outputs of a system, it works. However, conceptual models are often inferred and/or erroneous. For example, many people misunderstand how temperature control systems in buildings work. From the affordances, signifiers, and results of a thermostat, people see that when you change the temperature on the thermostat, the temperature of the building gradually changes until that temperature

is reached. However, many people assume the air coming out of the vents is the same temperature as the one set on the thermostat–an incorrect conceptual model. Using this model, people go on to assume that in order to change the temperature rapidly, they should set the thermostat to a much higher or lower temperature than the one they want. In reality, the air coming out of the vents is always the same temperature. When one sets the thermostat, the vents blow until the desired temperature is reached, and then the air shuts off. When the temperature varies, the vents turn back on to maintain the desired temperature. A user utilizing the incorrect conceptual model would waste time and energy attempting to change the building temperature rapidly. This inaccurate conceptual model leads to frustration and confusion, as the results (slow temperature change and a higher energy bill) do not align with what was expected.

Conceptual models help people better remember how to use a system, since the sequence of actions needed to do something is not arbitrary. Conceptual models also help in novel and problem-solving situations, as they help users infer the results of a new action (or troubleshoot an unexpected result). Good conceptual models also help give the user a feeling of control–they know what to expect and how to get the results they want. Good design must convey a good conceptual model to the user to help them better understand how to use a system. An important thing to note is that when designing a user interface for a software application, this does not mean communicating implementation details to the user. Rather, the design must communicate a good way of understanding how the app functions for the user's purposes. For example, a file system on a computer contains folders where files are stored. Although this may not be accurate for how the computer actually stores the files, it provides a working conceptual model for the user [13].

There are several strategies to conveying a good conceptual model to the user. Mapping the controls as closely as possible to the task the user wants to complete helps the user understand the system quickly, as they can apply their knowledge (or conceptual model) of the task to the system [10]. When using a computer's file system, users use existing

knowledge of physical folders (places that store files) to understand the digital folders, which, just like physical folders, store files. Being clear and consistent with the objects users can interact with, including the actions that can be performed on those objects, the settings the user can change on those objects, and the relationships between the objects, also helps the user build a good conceptual model. Again, making these objects match the user's task and goal helps them understand it quickly. Also, using a consistent lexicon throughout the software communicates a conceptual model to the user while limiting areas of confusion, as consistency helps the user better navigate the software [10].

3.1.8 Location of Knowledge

Another aspect of discoverability is the placement of knowledge. Knowledge can exist either in the head or in the world. Knowledge in the head is anything that can be remembered or summoned up from within the mind. This includes facts ("Austin is in Texas") and skills (knowing how to play the clarinet). Knowledge in the head is very quick to retrieve, but is susceptible to memory errors. It also takes time and effort to learn in the first place. Knowledge in the world is any knowledge that is present in the physical world, such as a sign with a street name on it or an instruction manual. Knowledge in the world is always available (it doesn't vanish once the user has forgotten it), but may take time to interpret when encountered.

When it comes to design, putting knowledge in the world greatly aids discoverability, as the user can immediately utilize it without needing to remember knowledge in the head. However, too much knowledge in the world may be clunky or cluttered for an experienced user, who has internalized the knowledge and no longer needs it in the world. The key is putting enough knowledge in the world for a novice to get by (or help the expert who has forgotten) but leaving room for the expert to ignore if they would like. The perfect example of this is the computer keyboard. Novices start by looking down at the keyboard, hunting and pecking for the letters they need. All the knowledge they are using is in the world.

However, through effort and learning, many people develop typing as a skill, memorizing letters on the keyboard until they can type very quickly without looking down at the keyboard (or even thinking about using it). These people rely on the knowledge in their head to type. If, however, the typist forgets where a character is (perhaps it is a seldom used character), they can just quickly look down at the keyboard, find it, and continue on their way. The keyboard puts all the necessary knowledge in the world, but leaves room for the expert to rely on the knowledge in their head [13].

3.1.9 Human Error

According to Don Norman, there is no such thing as (unintentional) human error: human error occurs as a result of a poorly designed system. Specifically, human error is caused by forcing humans to act in unnatural ways, such as remembering precise concepts, being exact/precise in action, multitasking, or staying alert for long periods of time. These are all activities machines are good at, but humans are expected to do them. Norman argues good design should prevent errors by understanding the limits and strengths of people and machines, creating a collaborative environment to get tasks done instead of forcing people to conform to machine's (inhuman) standards [13].

There are several ways to create this collaborative environment and design for human error. First, making errors easy to discover and correct helps the user complete their task smoothly. Forcing them to restart every time they make a mistake is poor design, as it forces the user to be precise for the duration of the activity. Adding 'undo' and 're-do' functionality allows the user to easily correct any mistakes or missteps. If an action is permanent, it should be harder to complete. For example, adding a dialog asking the user to confirm the action may help them discover a possible error. Having sensibility checks also helps catch possible errors. For example, if a money transfer application flags (and asks the user to confirm) unusually large sums, it helps catch possible typing errors made by the user. An important thing to note–if the user is asked to confirm every action,

they may stop reading the notices and approve an erroneous action. Confirmation dialogs should be rare enough for the user to stop and read instead of ignoring. Another way to prevent errors is to add constraints preventing certain errors in the first place [13]. For example, if a form on a website needs all fields to be filled out before submitting, the 'submit' button at the bottom may be disabled until all fields are filled out. The disabled button is a constraint preventing the error of submitting empty fields on the form.

One of the most important ways to prevent errors is to provide the user with a good conceptual model. A good conceptual model helps with remembering how to do a sequence of actions, which helps to prevent errors caused by memory lapses [13]. Forcing the user to remember arbitrary rules and action sequences makes errors much more likely. A common cause of arbitrary rules and actions is forcing the user to do tasks that are not directly related to their goal. For example, in an application where the user plays chess, the only actions related to the user's goal (during the game) are choosing a piece to play and a place to move it to. If the user must remember (for example) to switch to 'move' mode or name the move, this interrupts the user's goal and opens them up to errors, as they must remember to do these arbitrary actions every time their turn comes [10].

3.1.10 Complexity

Complex tools are often needed to complete certain tasks. For example, cooking and kitchens are quite complex. A kitchen (usually) contains a multitude of tools and appliances, confusing to any novice or one unfamiliar with the kitchen's layout. However, one's own kitchen is rarely confusing to themselves–they know where everything is and manages the complexity without issues. This because they have a good conceptual model of where each tool, appliance, and ingredient can be found. However, poor design of complex machines leads to confusion. For example, a washing machine with 20 different settings (and only icons to indicate each setting) creates confusion [13]. The user, rather than going through the effort to learn all of the options, simply memorizes the one setting they know works

and never bothers to explore other features [10]. Good design manages complexity without generating confusion [13].

There are several strategies for managing complexity. First, while some tools and systems may need to be complex, keeping the tools as simple as possible (by having no objects or tasks unrelated to the activity at hand) helps by not adding needless complexity and confusion to an already complex task. Also, providing sensible default settings allows the user to jump straight into the task at hand while also giving them the freedom to adjust settings if needed. For example, text document applications (such as Microsoft Word) have default text formatting in place when the user creates a new document, allowing the user to start typing without having to choose a font, paragraph style, spacing, or any of the multitudes of options for formatting text. The user may still change any of those settings as they please, but simply sitting down to type something out has no extra tasks to complete beforehand. Providing templates also helps with complexity, as the user does not have to start from scratch every time. When a complex activity has many steps, having a guided path or wizard to help the user through the process limits confusion. Many desktop applications have a set-up wizard to help the user through the installation process. Many set-up wizards also include sensible defaults, so the user may click through the process quickly (if they do not desire to change or even understand some of the settings) and get started on their activity. Another strategy to managing complexity is to show only the most relevant information related to an activity. Complex optional information and settings may be hidden away behind "advanced" or "detail" menus or not shown to the user at all. Designing for the most common cases helps with this–making the most common or most used options readily available while hiding more detailed options helps prevent the user from being overwhelmed with information. Also, having a few generic commands (such as Create, Save, Move, or Copy) that apply consistently to objects across the application helps manage complexity, as it lessens the amount of arbitrary action sequences the user must remember. If these generic commands are consistent with common practice outside

the application (such as having a save icon in the top left corner that saves the current state of the application), the user may not even have to learn new action sequences. Finally, making an application customizable may help with complexity. Letting the user choose where controls are lets the user shape the application to match their expectations. However, customizability must come with sensible defaults–forcing the user to design the layout for the application adds needless tasks to the activity at hand. The base design that the user may customize should still follow good design practices, so the user may adjust if needed but can still complete their activity without extra hassle [10].

A good example of a system that manages complexity well is modern smart phones. Smart phones are quite complex, having many uses and purposes. However, most people have no trouble using their smartphone. Part of this is due to their customizability–the user may choose where to place different apps on the phone's home screen. Settings that are frequently changed (such as adjusting the screen brightness or turning on Bluetooth or airplane mode) are often just a swipe away, no matter what the user is currently using the phone for. Less frequently accessed settings, such as language and date/time, are still accessible through the settings app. By following several of the strategies for managing complexity, smart phones allow users to complete tasks efficiently.

CHAPTER 4

Designing an Idea Management Tool for Composers

As previously stated, the goal of this software is to aid the beginning of the composition process: recording the germinal idea and developing musical sketches. In terms of broad functionality, this means the software must be able to record ideas in the form of pictures, drawings, audio recordings, notated musical fragments, and text while providing functionality to manipulate, edit, and export these ideas.

There are many ways to design such a tool to adhere to design principles and provide a good user experience. However, given time constraints and the (in)flexibility of the code libraries used in implementation, compromises were made to ensure the software had necessary functionality. This chapter discusses where design principles are successfully applied. For details on implementation and discussion on the compromises made where the design choices are less ideal, see Chapter 5.

4.1 Applying Design Principles

4.1.1 Conceptual Model

Similar to how computer desktops and file systems use real-world analogies to convey a conceptual model (by linking objects in the system to physical objects the user encounters outside the computer), this software is analogous to a corkboard or the surface of a desk, with 'cards' of information the user may move around throughout the space. This includes

stacking cards on top of each other, not just moving the cards along the x-y plane. Each card represents a different idea the composer has, and may be an image, a drawing, a fragment of notated music, an audio recording, or written text. As with a real workspace, the user chooses what ideas to include (for example, choosing an image or writing down a notated music fragment). Besides simply moving the cards around the workspace, the user may perform different actions depending on the type of idea stored in the card. If the card shows an image, the user may draw on top of the image and erase all the drawings on a particular image. If the card contains an audio recording, the user may listen to it and adjust the playback speed. A card may contain a fragment of music–in this case, the user may listen to it and draw on top of the notated fragment. If the card contains text, the user may see and edit the text on the card. In a slight departure from the card analogy, the workspace may also include an audio recorder, where the user may record live audio and listen to it. To see all these cards as rendered by the software, see Figure 4.1. The red labels in the figure denote signifiers, discussed below in section 4.1.3.

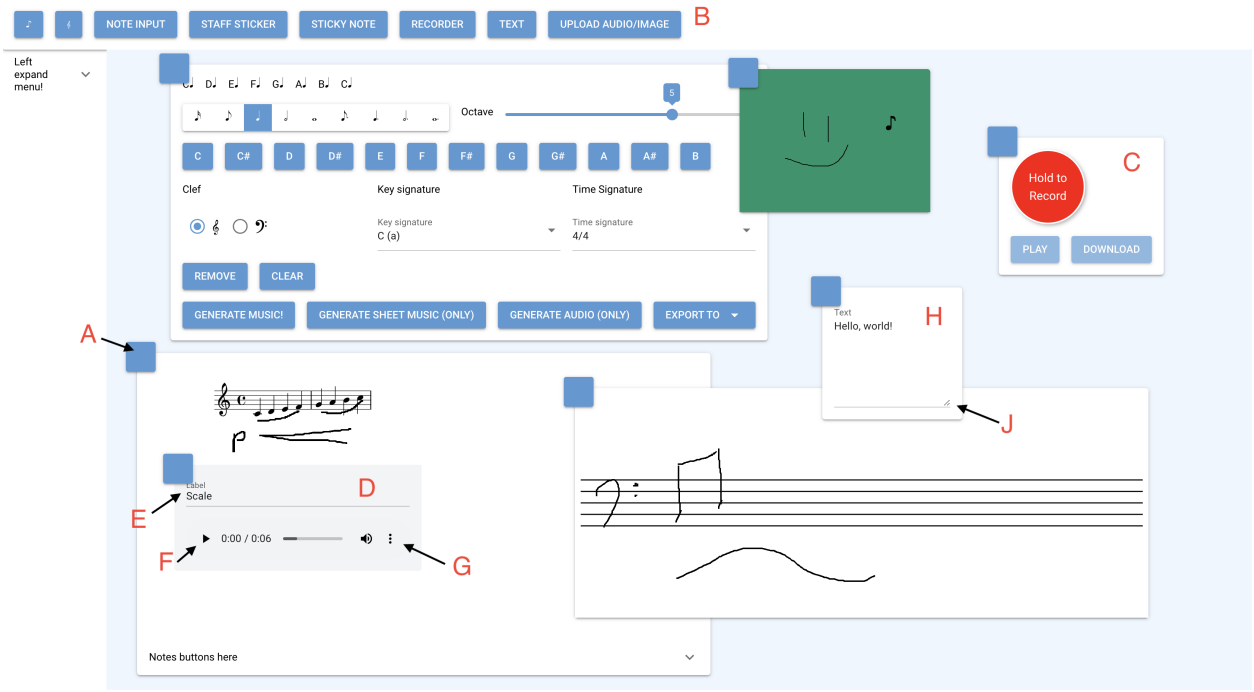

Figure 4.1: Workspace containing a note input card, sticky note, recorder card, notated music, audio player card, text input card, and staff sticker

When it comes to implementing a user interface for the various cards and objects that may exist in the workspace, all must have similar controls and common commands. For example, the controls to move a card throughout the space should be the same, whether it is a text card or music fragment card or a drawing card.

4.1.2 Affordances

The previous section on the conceptual model covered the high level affordances offered at an abstract level. Here, we will discuss the more detailed affordances offered by the application in the context of running on a computer.

The main workspace in the application affords dragging and placing objects ('cards' from the previous example), including files uploaded to the application. Uploaded files can be images, audio tracks, or MusicXML documents, which are rendered in the workspace as cards with an image, an audio player, or notated sheet music, respectively. Music symbol stickers may also be placed and dragged in the workspace.

There are several types of cards that afford drawing/making markings on them: an image (chosen and uploaded by the user), a fragment of notated sheet music (generated from a music fragment), a staff sticker (which shows a blank music staff), and a sticky note (a small blank card with a background color). These cards also afford deleting all drawings made, resetting the card to its original state.

Two card types afford handling audio, albeit in different ways. The recorder card affords recording live audio, playing the recording back, and downloading the recording made. The audio player card affords playing back music (either from an uploaded audio file or generated from a music fragment), adjusting the playback speed, downloading the music as a WAV file, and labeling the card (by typing in a short bit of text attached to the card).

Text input cards afford inputting and editing text input from the computer keyboard. They also afford resizing vertically (to show, hide, or match the size of any input text).

There are two card types that afford handling music fragments. The note input card affords notating music on a single staff (either bass or treble clef) in any key (up to 7 sharps or flats) in the most common time signatures and in the 7 most common octaves. Notes may be any duration between a 16th note and a dotted whole note. The note input card also affords removing the last note inputted and clearing the music fragment of all input notes. The notated music card affords seeing a music fragment as notated sheet music, drawing on the notated sheet music, and listening to the music fragment. Both types of cards afford downloading the music fragment as a MusicXML, PDF, WAV, or MIDI file.

All objects in the workspace afford having multiple instances–for example, a user may have several different music fragments or several copies of the same image.

4.1.3 Signifiers

A variety of signifiers (mostly labels on buttons) show the various affordances offered by the software. Many are labeled in Figure 4.1 with red letters and arrows. On all cards, a small blue button in the top left corner of each card signals where the user clicks and drags to move the card in the workspace (labeled letter A in Figure 4.1). Buttons in the top panel (above the workspace, labeled letter B) signal where the user clicks to insert or create a new card, including via file upload.

On cards that afford drawing, context menu options (options that appear when the user right-clicks the element) signify where the user may reset the image to its original state (the 'clear drawings' option) and bring the card in the front of all objects (look ahead to Figure 5.1 in Chapter 5 to see a staff sticker card with its context menu open).

On the recorder card (labeled letter C), the 'hold to record' button signals where the user clicks and holds to record audio. The 'play' and 'download' buttons on the same card signal where the user clicks to listen to and download the audio recording, respectively. The audio player card (letter D) contains several signifiers. The label (denoted by letter E) signals where the user clicks to type in their own label. Once clicked, the 'start typing' text

placeholder signals the user may simply type on their keyboard to insert text. The play button (denoted with the play symbol that changes to a pause symbol while the audio track is playing, letter F in Figure 4.1) signals where the user clicks to play and pause the audio track associated with the card. The menu to the right of the playback bar (letter G) signals where the user may download the audio track or adjust the playback speed.

The text input card (letter H) has many of the same signifiers as the label on the audio player card. In fact, the only difference is that the label says 'text' (instead of 'label') and a small triangle symbol (letter J) in the bottom right corner signals where the user clicks and drags to resize the text box.

The signifiers for music fragments are nearly all contained in the note input card–a card (with the same basic properties as other cards in the workspace) that holds all the buttons for manipulating a music fragment (seen in Figure 4.2). Each of the affordances related to

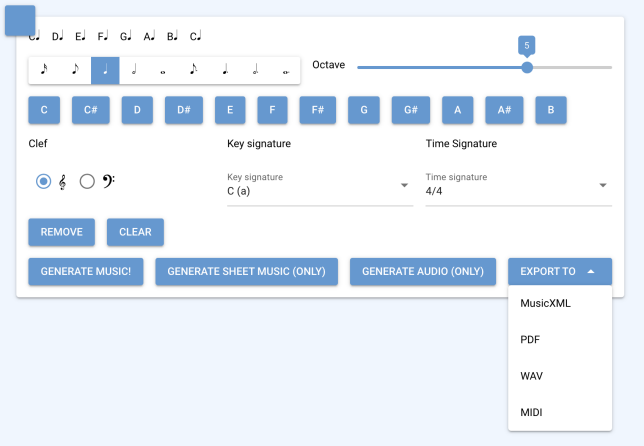

Figure 4.2: Note input card signifiers

note input is signaled through various buttons, toggles, sliders, and drop-down menus within the card. A toggle selector signifies where to select note duration, a slider signifies where to select a note's octave, 12 different pitch buttons signify where to click to add a pitch to the music fragment, two small (labeled) radio buttons signify where to set the clef of the music fragment, a labeled drop-down menu signals where to set the key signature, and another labeled drop-down signals where to set the time signature. There are also

buttons that signal each of the following: removing a note, clearing the music fragment of notes, generating sheet music and audio (both together and separately), and downloading the music fragment as a MusicXML, PDF, WAV, or MIDI file.

4.1.4 Mapping

In an ideal setting, all of the controls would directly map to the real-world activity they emulate. However, due to the time and coding constraints mentioned above, much of the mapping was less direct than it could be. However, mapping was designed to be as close as possible given those constraints.

For example, when dragging cards around the workspace, the card follows the mouse coordinates closely and lands where the mouse is released. Similarly, drawing marks map directly to the physical location of the mouse on screen.

The various toggles, buttons, and drop-downs utilize natural mapping for intuitive value selection. Note duration, pitch, and octave selectors are organized left to right from lowest to highest value: shortest to longest duration, lowest to highest pitch, and lowest to highest octave, respectively. The clef, key signature, and time signature selectors are organized left to right as they appear in standard notated music: clef on the far left, then key signature, then time signature. The key signature drop-down options are listed in order of sharp keys (from one sharp to seven sharps) and then flat keys (from one flat to seven flats), a common way of organizing key signatures.

4.1.5 Feedback

When the user adds a note to a music fragment, a visual representation of the note's pitch and duration is displayed. If other notes exist in the music fragment, the visual is appended to the end of the other notes' visual representations. This can be seen in Figure 4.3

Small pop-up notifications appear on screen for a short time when certain actions are

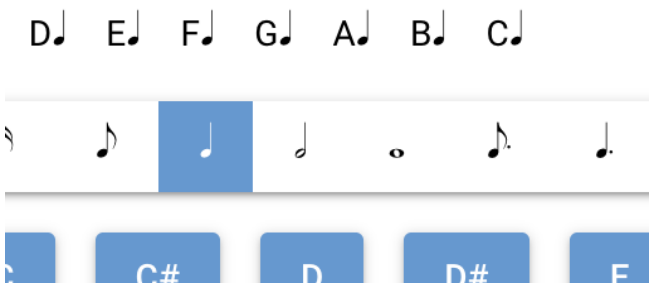

Figure 4.3: Note input card feedback: showing the current music fragment

completed to let the user know the action was successful. These notifications appear when a recording has finished, a file has been successfully uploaded (along with the file name and type), and the clef, key signature, or time signature has been changed for a music fragment. A notification is also set to appear if a file type the system cannot handle has been uploaded. However, constraints are in place to ensure this should never happen in the first place (see the section 'Constraints' below for more details) and the notification is there in case something unexpected happens.

4.1.6 Constraints

In this software, constraints are used almost exclusively to prevent the system from crashing or stopping. For example, the file upload dialog only accepts image, audio, and MusicXML files since the application is only able to handle files of those types. When the user opens the dialog box and browses files on their computer, only image, audio, and MusicXML files may be selected for upload. Another constraint may be seen in the recorder card. Before the user has recorded any audio, the 'play' and 'download' buttons are disabled. Both of these constraints prevent the application from crashing from trying to handle incorrect file types or files that do not yet exist. These constraints also allow the user to focus on the task at hand, since they do not have to remember which file uploads are allowed or if they have recorded something yet–the software reminds the user and guides them towards correct actions.

4.1.7 Location of Knowledge

With this software, location of knowledge is most important while the user is inputting a music fragment (that will eventually become notated sheet music and/or an audio track). Since each note has several properties that may need adjustment as each note is entered, the application shows the currently set values for these properties. This prevents the user from having to remember the current note duration, octave, clef, key signature, and time signature: all are visible at all times. This may be seen in Figure 4.4: the duration is set to quarter note, the octave is set to 5, the clef is set to treble, and so on. Also, the relative minor of each major key (the minor key with the same key signature as its major key) is listed alongside its major key in the key signature drop-down. This ensures the user may select a minor key without having to remember its relative major key. This may also be seen in Figure 4.4–the keys of both C major and a minor have no sharps or flats, so they are listed as a single option in the drop-down menu.

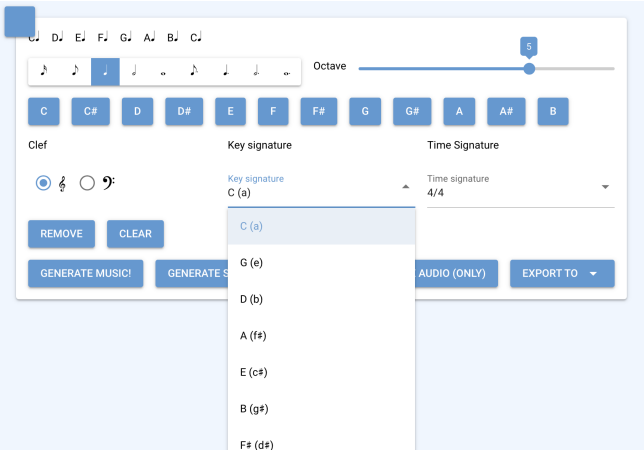

Figure 4.4: Note input card showing currently selected values and the key signature drop-down

4.1.8 Preventing Error

The placement of knowledge in the world also helps to prevent errors. The less the user needs to remember or memorize, the less likely they are to make an error regarding the information that would have needed to be memorized. For example, since this application

shows relative major and minor keys together, the user is much less likely to accidentally choose the wrong key signature if they are thinking in minor.

Allowing users to fix any errors they have made also makes the design more conducive to composing, since writing music is a highly exploratory process that often requires rewrites and edits to existing ideas. When it comes to note input, this software allows the user to delete the last note entered or even start over from scratch. On cards that afford drawing, the user may delete all drawings at once, resetting the card to its original state.

4.1.9 Managing Complexity

The main way this software manages complexity is by emulating existing formats, layouts, and defaults of software that composers are already accustomed to using. For example, most notation software defaults to notating music in the treble clef in C major in 4/4 time, so these same values are set as the defaults in the note edit card. Also, notation software tends to follow a general layout: macro items (such as parts and score settings) towards the top of the screen, micro items (such as dynamics and articulations) at the left of the screen, and the music itself in a large workspace in the middle of the screen. In this software, the top part of the screen contains buttons to insert ideas (the macro items in this case) into the workspace. The workspace itself takes up a large (central) portion of the screen, and new cards are placed towards the top left corner (a spot that is always prominent no matter the size of the viewing screen). Given that this software is focused on capturing general ideas and that many ideas may occupy the workspace at one time, there are no buttons or functionalities on the left–micro items are added and manipulated by interacting with an idea on the workspace (by drawing on it, clicking buttons on a card, or selecting an item from the context menu of a card).

The other way this software manages complexity is through consistency. All cards have the same controls for moving around the workspace and are created by clicking buttons that are grouped together at the top. Any card that affords drawing, whether it is a sticky

note, uploaded image, staff sticker, or a notated music fragment, has the same controls for drawing and deleting drawings. This minimizes the number of action sequences the user must learn to use the software.

CHAPTER 5

Implementation

5.1 Libraries and Dependencies

The combination of libraries used to implement this software led to a wide breadth of functionality almost entirely built on existing objects and functions present in the libraries. However, this limited flexibility and customizability, leading to compromises in the design of the user interface. The specific strengths and weaknesses of the foundational dependencies (NiceGUI, `music21`, and LilyPond) are discussed here.

5.1.1 NiceGUI

NiceGUI is a Python library for creating graphical user interfaces for the web browser [6]. Built for rapid web design, it offers a wide variety of pre-built objects with customization features and sensible defaults. For example, objects such as buttons, sliders, and drop-down menus could be generated in just a few lines of Python code (without needing any JavaScript, CSS, or HTML). The default styling for objects is generally sensible and easy to look at–I never needed to change any of the default fonts or colors and only adjusted sizing and padding to fit specific needs and functionality. There is also no front-end/back-end distinction. All the code needed for creating a GUI is contained in Python scripts.

NiceGUI was used to implement all the graphics of the application. Everything seen on screen in the application is some combination of NiceGUI elements. For example, all cards that afford drawing are made up of several NiceGUI elements: an interactive image (which handles mouse events and drawing functionality), a context menu (visible when the user right-clicks, this also contains 'menu items', another NiceGUI element), a button (to facilitate dragging around the workspace), and a card (which holds the other elements and has a drop shadow to provide depth). Figure 5.1 shows a staff sticker (one of the card types that afford drawing) with its context menu open.

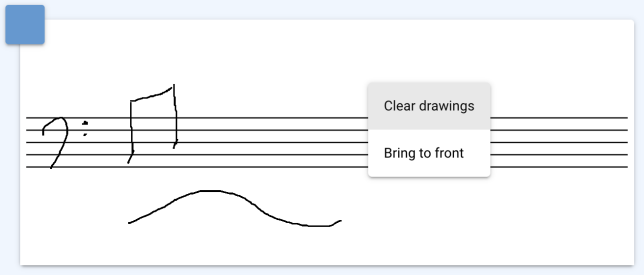

Figure 5.1: A staff sticker with context menu visible

5.1.1.1 Limitations

However, NiceGUI's customization features only extend to functionality provided by NiceGUI. Incorporating or implementing UI features outside what is included is difficult to do in a timely manner. Anything related to drag-and-drop functionality (dragging objects, layering and grouping objects, etc.) had to be hard-coded. For example, to help the user keep track of the many different objects in the workspace, the audio for a notated music fragment is grouped with that notated music fragment (see Figure 5.2). The user may move the audio player card anywhere in the workspace, but when the notated music card is moved, the audio player card moves with it. The two objects become linked when they are constructed–I was unable to find a way to link two already instantiated objects. This is a result of how NiceGUI handles parent/child relationships, that is, which objects are placed inside other objects. The movement of the parent (the notated music card, in

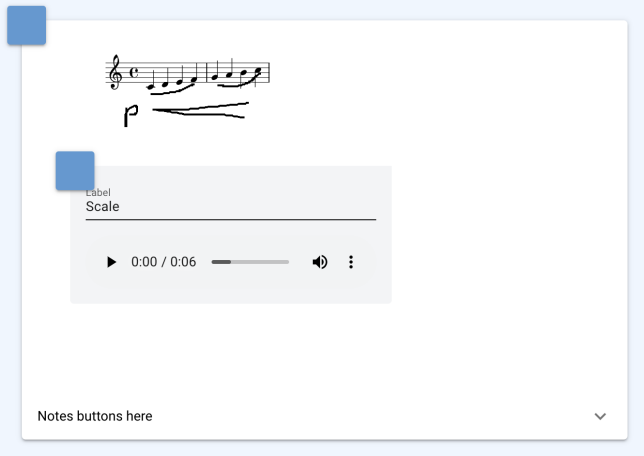

Figure 5.2: A notated music fragment and its associated audio player card

this case) determines the movement of the child (the audio player card), but never vice versa. The design may lead to some confusion on the user's part if the audio player card is far away from the notated music card but then moves when they try to move the notated music card. For this reason, the audio player card appears on top of the notated music card when they are generated in the workspace. This way, the two only appear separated if the user moves the audio player card a significant distance (despite the fact that they still move together).

The limited drag-and-drop features also led to poor mapping of note input. Many music notation applications allow the user to drag and drop music symbols directly onto the page of music. This almost directly emulates the act of notating music in the physical world, except instead of drawing symbols (which may be tedious), the application provides pre-defined symbols to be placed where the user pleases (as long as the placement follows notation rules). Implementing a similar feature in my software would require extending drag-and-drop functionality to include dragging for object creation, determining where the user would like to place a note or symbol on the staff, and translating where the user placed the note into computer-readable music (as well as enforcing any constraints needed for the whole system to work properly). Instead, I kept the graphics of music notation and representation of music behind the scenes completely separate. Unfortunately, this

was passed on to the GUI and note input is handled by various combinations of NiceGUI elements (toggles, radio buttons, sliders, and drop-down menus). Provided the various elements, however, I ensured the mapping followed sensible layouts and organization (see the section "Mapping" in Chapter 4 for details on how this was accomplished).

The difficulty of incorporating new objects and functionality into NiceGUI is also the reason the recorder card and audio player card look so different and have different controls (see Figure 5.3). Ideally, since they have overlapping affordances (playback and download

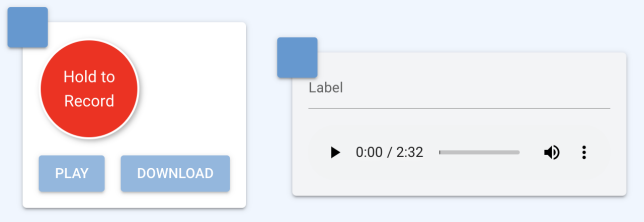

Figure 5.3: A recorder card and audio player card, compared

of audio), they should have the same controls for those affordances. However, each are sourced from two different locations. The audio recorder card is based based on the Audio Recorder object, developed by Mansar Youness and Falko Schindler in the NiceGUI community on GitHub [16]. The audio player card is a combination of elements from the NiceGUI library: an audio player, a short text input (for the label), and a card. Ideally, the recorder card should generate an audio player element based on the audio it recorded. This way, the source of the audio would not matter: the user uses the same controls to hear, download, and manipulate playback of the audio. However, the Audio Recorder object and audio player element are incompatible–the audio player element takes a link to a source file to play audio, but the Audio Recorder does not provide such a link. An alternative solution would be to create a custom audio player (based on NiceGUI elements such as buttons) to match the Audio Recorder. However, this would mean losing the playback control functionality and progress bar included with the audio player element. The compromise has been to keep the benefits of NiceGUI's audio player element while losing consistency of controls.

5.1.2 music21

`music21` is a Python toolkit designed for musicology (the study of music) [4]. It allows for easy representation of traditional Western music in Python, that is, music notated in the Western style. It is quite flexible in the music it can represent and has sensible defaults (similar to the ones discussed in the "Managing Complexity" section in Chapter 4). `music21` also comes with functions to read from MusicXML files and write to MusicXML, PDF, and MIDI files. In this software, `music21` is primarily used for storing music ideas and generating export files.

5.1.2.1 Limitations

Since `music21` is primarily designed for music analysis, there is limited functionality for editing notes in the middle of a 'stream' (a container of notes). This is the primary reason my software does not afford editing notes in the middle of a music fragment, despite its importance in preventing errors. However, `music21` has existing functions for clearing a stream and popping the last note, so these affordances where passed on to the note edit card.

Another slight drawback of `music21` is that the only audio format it handles and outputs is MIDI. MIDI files do not contain any sound data themselves–rather, they contain information about which instrument sounds for certain durations. It is impossible to listen to a MIDI file without any processing. In order to create audio files that could playback input music streams, my software utilizes FluidSynth (a command-line program) [7] and a SoundFont file [15] to add physical sounds to a WAV file (that is then downloaded by the user or used to create an audio player card).

5.1.3 LilyPond

LilyPond is a music engraving program that takes text as input and creates notated music [2]. The outputs are highly customizable, allowing for complex notation capabilities (including removing bar lines, clefs, and time signatures). My software uses LilyPond to generate SVG files of music fragments, which are then placed in music notation cards. `music21` can write to LilyPond text files and even uses LilyPond to generate PDFs. While `music21` also uses LilyPond for SVG generation, I found it necessary to adjust code on the LilyPond level before passing the output to my GUI (specifically to adjust page size and formatting settings). To create music notation cards, my software takes a `music21` stream, uses a `music21` function to write the stream to a LilyPond text file, makes small adjustments to the text of the LilyPond file, then uses LilyPond to generate an SVG file from that text.

5.1.3.1 Limitations

While LilyPond is highly flexible in its notation output, LilyPond text files are difficult to edit programmatically. The edits my software makes to `music21`-generated LilyPond files are minimal–commenting out a few lines and adding a line or two at the top of the file. Accessing more than high-level, full-document settings (as opposed to details on the note- and measure-level) would require a level of text processing akin to a tokenizer or compiler. Also, LilyPond is designed to generate ready-to-print music formatted for the performer–it only outputs single pages of music or single lines of music (formatted to fit the width of a page) at a time. These formatting constraints are not needed to represent music on a computer screen, which has the capability to scroll endlessly in any direction.

Both `music21` and LilyPond have one major flaw counterproductive to the goal at hand: they both enforce music notation rules. Although LilyPond has the flexibility to engrave music that looks as though it is breaking the rules (such as removing barlines), they are still present behind the scenes. Circumnavigating those rules (using these libraries) could

take as much processing as enforcing them in the first place. However, both are needed for the basic functionality of storing, editing, displaying, listening to, and exporting music fragments. Ideally, my software would combine the flexibility of paper with the power of notation software (or computers in general). Instead, I placed the functionality of both mediums in the same application but in separate objects. While this does not remove the issue of copying music from a flexible paper-like source to notation software, the app allows music ideas in various formats to be held in a single location.

CHAPTER 6

Results

6.1 Feedback From Composers

To see how well my software aids in the composition process, I observed five composers using my software. Four were composers I had interviewed in the requirements gathering stage, but one had not been involved in the process until that point and therefore had little prior knowledge as to the purpose and goal of the software. As with my initial composer interviews, they varied in experience and in style (although only one focuses on electronically mixed recordings rather than notated music). Each composer was prompted to try to compose something using my software with the goal of seeing which features were most helpful and which features needed to be added or expanded.

Every composer's first step was to familiarize themselves with how the software worked. Many did this by recreating familiar melodies in the note input card (such as "Twinkle Twinkle Little Star") and systematically exploring the functionality of the software (usually by clicking buttons in the top panel from left to right). Many also tried to push the limits of the application by inputting music with complicated notation or notes with extremely low or high pitches. Many spent most of their time on the note input card, but all explored the drawing and recording cards. Few composers made it to the point of

actually composing anything in the software, but their interactions with it show future areas of improvement as well as features that were helpful.

While many composers spent much of their time on the note input card, several remarked that they enjoyed the drawing functionality, text card, and recorder card. None asked for different formats–my software seemed to cover the possible formats ideas could take. However, most of the feedback pertained to extending the flexibility and clarity of those formats.

There were several common suggestions and areas of confusion (see "Future Work" below for a discussion of the possible reasons behind these and solutions). The most common suggestion by far was adding the ability to delete objects from the workspace. After creating more than three or four objects, the workspace felt cluttered and each composer wanted to delete objects they were done using. Many composers also mentioned this application would be most helpful to them on a tablet, since the drawing functionalities were difficult to utilize using a mouse or track-pad. The sticky note object was somewhat misleading–most composers assumed it afforded typing text instead of just drawing. Many also wanted to be able to expand the size of the sticky note to give them more space to draw ideas. Despite the feedback reminding users of the currently set note duration, octave, and current state of the music fragment (visible at the top of the note input card), many composers forgot all three while inputting notes and often had to backtrack to fix an error. Sometimes the error went unnoticed until after the notated music fragment was generated. This seemed to be less of an issue with the clef, key signature, and time signature. The discoverability of the drawing functionality was low–many composers discovered it by accident or were unsure of which cards afforded drawing. For example, many discovered that the sticky note afforded drawing (after figuring out they could not type into it) but some never realized they could draw on top of notated music fragments. Finally, many were unsure of how to edit music fragments. They all understood the 'clear' button on the note input card meant starting over from scratch, but some were confused as to what

the 'remove' button would do. One composer never even attempted to click 'remove' and started over from scratch every time he wanted to change something within the music fragment.

Several pieces of feedback were mentioned by only one or two composers, but are worth noting for their potential positive impact. First, being able to color code objects was suggested by two composers as a way to link cards belonging to the same idea. Also, while many enjoyed the playback controls offered by the audio player card, some recommended being able to change the tempo of the music fragment directly. Two composers also attempted to type note note names in from the computer keyboard (in addition to clicking the pitch buttons). Currently, my software is unable to represent flat notes and rests. Surprisingly, only two composers commented on the lack of flats and only one mentioned needing rests. Finally, two composers were distracted by the 'generate music!' button at the bottom of the note input card, noticing it (and wondering if they should click it) before discovering how to input notes in the first place.

6.1.1 Incorporating the Tool into the Composition Process

Besides simply getting a feel for the software and giving suggestions, one composer was able to utilize my software to help write a piece they are currently working on. Having already familiarized themselves with the software during the initial feedback stage, they were able to develop ideas drawn on paper into several measures of notated music (and export them to notation software) in an hour using my software with a plug-in drawing pad.

The composer started the session with several sketches of existing ideas written on a few pages of staff paper. Before using my software, the composer knew the piece would be for solo cello and composed using the twelve-tone technique (in which each of the twelve musical pitches are used equally instead of focusing on one key). Some ideas were copied directly off the staff paper and into the software (such as the main theme of the piece). To

develop a variation on the main theme, the composer drew a melodic contour onto a staff sticker, placed note stickers along the contour, and then entered the desired pitches and durations into the note input card to create the variation. See Figure 6.1 to see the sketch and its resulting notated music fragment (note the fragment is written an octave below where the sketch voices the pitches).

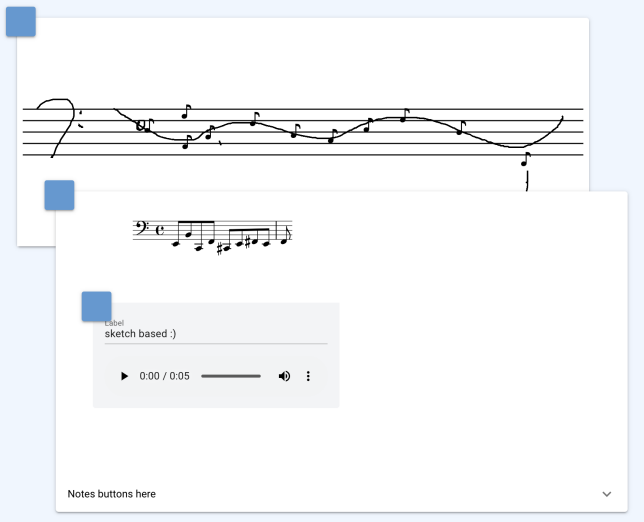

Figure 6.1: A musical sketch (towards the upper left) and the resulting notated music card (in the middle)

Several objects were created as reference points while composing this piece. The composer notated a tone row and imported (as an image) a tone matrix (both common tools when composing twelve-tone pieces), and also typed the general structure of the piece into a text card.

Once the main theme and variation were notated, both were copied into a single music fragment and exported as a MusicXML file. From there, the composer added some text about the structure of the piece in MuseScore (a music notation software) [11]. The final result is seen in Figure 6.2

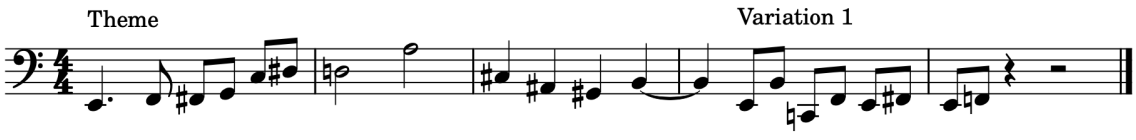

Figure 6.2: The result of one composition session (with processing by MuseScore)

The composer noted several strengths of the software. First, the note input card is well suited for the format of the piece they composed–only one staff was needed at a time and the note input card had all twelve pitches easily on hand. They also enjoyed being able to see all their ideas in one place–usually, ideas are scattered between loose leaf paper, a staff paper notebook, a computer, a binder, and even sticky notes. They especially enjoyed seeing the matrix, which is usually created on the computer despite being needed during earlier, notation-software-free stages of composing. This software allowed all the ideas and references for one piece to be condensed into a single spot. Finally, the composer found that my software "facilitates creativity" by allowing them to switch quickly between idea formats. Being able to switch from drawing to notating to listening was helpful and convenient, and allowed them to focus on how they wanted to develop their musical ideas.

The main suggestion this composer had for improving the application was adding functionality to manage clutter in the workspace. Figure 6.3 shows how this composer's workspace looked after their composition session.

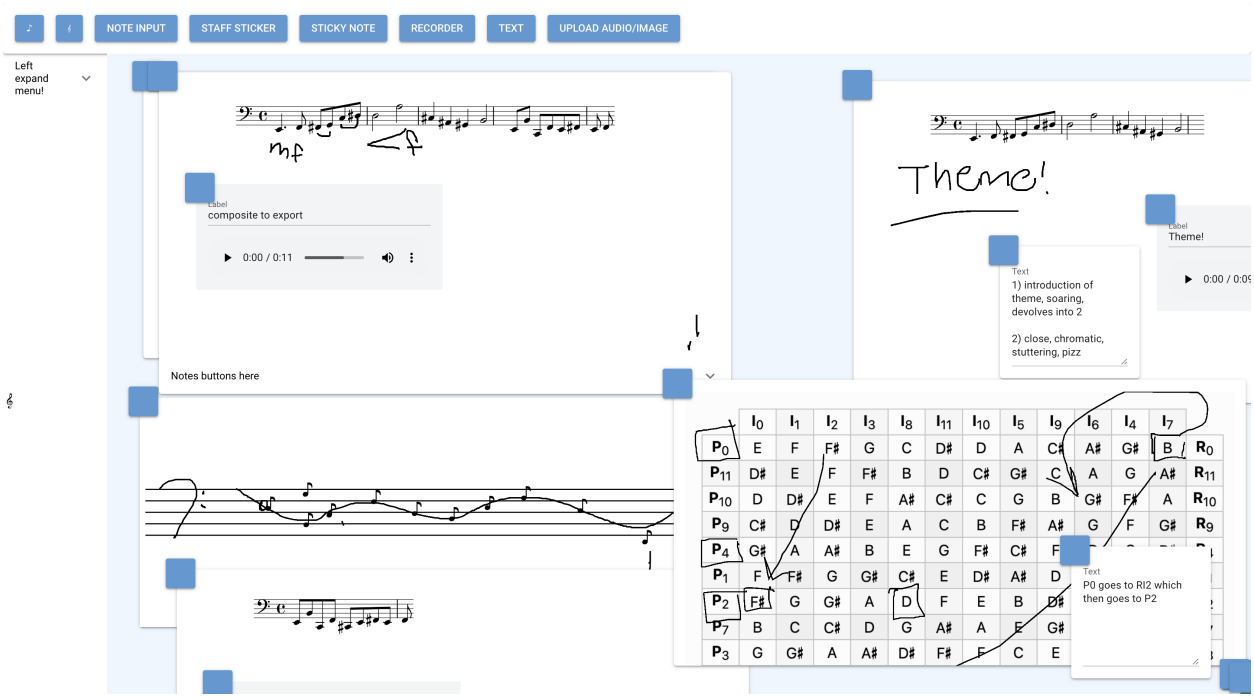

Figure 6.3: The composer's workspace

While the software was able to help in some areas of the composition process, there are

still many areas of improvement. The possible root causes of the issues and suggestions from all composers, along with possible solutions and improvements, are discussed below (in "Future Work").

6.2 Future Work

There are many areas in which the software could be improved. This section first addresses specific feedback given by composers (usually pertaining to specific features) before discussing broader functionality and how my software could better help the composition process.

Every composer, while using my software, ran into the issue of having a cluttered workspace after a short period of time. Many suggested deleting objects as a solution. Another solution would be to add more organizational functionality. This could come in the form of folders, multiple workspaces/tabs, and minimizing objects (either to hide from the main workspace or to include/group objects into a folder). Any and all of these features would aid in reducing clutter in the workspace, allowing the user to focus on the ideas at hand. Many composers also had difficulty utilizing drawing features with a mouse or track-pad and suggested using this app on a tablet (with a stylus) for easy drawing. One composer even suggested plugging in a drawing pad to the computer. Either way, expanding the application's compatibility with other devices would not only help with drawing but would also allow for greater flexibility and portability of the application.

Memory slips during note input occurred because the composers' eyes only focused on what they were looking for (usually pitch). Despite the information being close at hand, it was in their peripheral vision and often went unnoticed. The solution to this would be to provide better mapping to notated music (as discussed in Chapter 5). When reading sheet music, the pitch's octave, duration, and relation to the notes around it are visible just by looking at the note. By separating out these characteristics, my software limits

the speed at which users may pick up on and remember this information. This is also why composers generally had less trouble remembering the current clef, key signature and time signature. These three properties are not obvious when focusing on a single note–readers of sheet music are accustomed to keeping them in mind. For example, Figure 6.4 shows a small piece of notated music. When focusing on the first note in the second

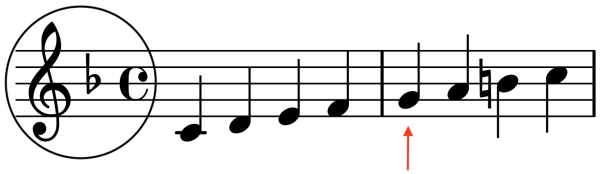

Figure 6.4: When focusing on the note indicated by the arrow, the circled material is difficult to parse using just peripheral vision

measure (denoted with a red arrow), the user (trained in reading sheet music) can tell the pitch name (G), octave (4), and duration (quarter note) as well as those three properties of the notes immediately before and after that note. However, the clef, key signature, and time signature (circled in the figure) are not legible when focusing on that note–the eyes must move to the far left to check. Composers (who utilize sheet music) are used to the individual note properties being visible just by focusing on the note. Matching this in my application would speed the note input process and help reduce memory errors. Another possible improvement to the mapping of my software would be to adjust the playback settings to reflect music tempo (which is denoted by beats per minute) instead of multiples of the default speed of playback (1x speed, 2x speed, etc.).

When using my software, many composers wanted more room to draw out ideas. This could be solved by making the starting canvases ('sticky notes') larger and allowing the user to resize cards that afford drawing. Also, adding a signifier to show which cards afford drawing (such as changing the cursor to a marker icon) would help discoverability. Sticky notes themselves were a sticking point–most composers assumed they could type into them and attempted to do so before drawing on them. Renaming the object to 'drawing' or 'drawing canvas' would help limit this confusion. There were two other labels that were

confusing for composers, both on buttons in the note input card. Many did not immediately realize that the 'remove' button would delete the last note in the music fragment. This would be solved by renaming the button to 'remove last note' or 'backspace'. Some were also confused by the 'generate music!' button on the bottom of the card. Due to its prominent placement (and exclamation point) it seemed to be the first button seen after glancing over the note input buttons. Disabling this button (including giving it the usual 'disabled button' signifiers) until notes have been entered may help remove some of the confusion.

Some composers suggested including color coding objects, showing that distinguishing cards in the workspace is needed. This could also be achieved by adding labels to more objects (as with the audio player card), but color may be more effective as it is more visually apparent. Typing in note names was also suggested by a few composers. In general, making the most common tasks (such as entering notes) easier to do would help with the speed of composition. Also, some music notation applications allow users to type in notes, so this would be another way to make my tool consistent with other music tools.

While few composers directly suggested adding rests and flats to music fragments, both are needed to represent some notated music ideas accurately and should be added in any future versions of the software.

In terms of broad functionality, there are three main improvements to be had. First, the main drawback of notation software in general is that it requires many aspects of music to be present in order to record an idea in the first place. For example, allowing the user to input rhythm alone or melodic contour alone would give them more freedom and flexibility when recording ideas. Users of my software can do this to a small degree (by using the drawing canvases to express ideas), but the format of those ideas (i.e. drawings) make them difficult to combine and export. The second main improvement is allowing recorded ideas to be more easily edited, manipulated, and combined. Currently, my software allows for many ideas in different formats to be seen at once, but the only way to combine those

ideas is to start with a new object from scratch. The editing capabilities for each format are also limited. By expanding the functionality to allow users to better edit and combine ideas, my software would be better suited to the composing process. Finally, the software currently does not have a way to save and reload objects from a composing session. Being able to return to previously recorded ideas is needed while composing, and having this ability in my software would make it more convenient and helpful.

CHAPTER 7

Conclusion

To better help composers record, store, and manage their early ideas, I developed an idea management tool based on interviews with active composers and background research on the composition process and composition tools. The GUI of the software is analogous to a desktop or corkboard, with cards containing each idea spread across the surface. While the tool follows many standard design principles, some compromises were made in order to provide basic functionality given the implementation constraints (such as providing less-than-ideal mapping for notating sheet music). Feedback from composers shows the app is strong in its ability to contain ideas of many formats: drawings, notated music, text, images, and audio recordings (including live recordings). The next step to improving the tool is giving it more organization features (such as deleting and resizing objects) to help when many ideas are present in the workspace at one time. The ability to contain all ideas related to a piece of music proved helpful for composers and aided the creative process.

References

[1] Stan Bennett. "The Process of Musical Creation: Interviews with Eight Composers". In: *Journal of Research in Music Education* 24.1 (1976), pp. 3–13. DOI: 10.2307/3345061. eprint: https://doi.org/10.2307/3345061. URL: https://doi.org/10.2307/3345061 (pages 3–4).

[2] Colin Campbell et al. *LilyPond*. Version 2.24.4. July 21, 2024. URL: https://lilypond.org/index.html (page 34).

[3] Vincent Cavez et al. "Challenges of Music Score Writing and the Potentials of Interactive Surfaces". In: *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*. CHI '24. Honolulu, HI, USA: Association for Computing Machinery, 2024. ISBN: 9798400703300. DOI: 10.1145/3613904.3642079. URL: https://doi.org/10.1145/3613904.3642079 (page 5).

[4] Michael Cuthbert et al. *music21*. Version 9.1.0. June 16, 2023. URL: https://www.music21.org/music21docs/index.html (page 33).

[5] Jérémie Garcia et al. "Structured observation with polyphony: a multifaceted tool for studying music composition". In: *Proceedings of the 2014 Conference on Designing Interactive Systems*. DIS '14. Vancouver, BC, Canada: Association for Computing Machinery, 2014, pp. 199–208. ISBN: 9781450329026. DOI: 10.1145/2598510.2598512. URL: https://doi.org/10.1145/2598510.2598512 (page 6).

[6] Zauberzeug GmbH. *NiceGUI*. Version 2.4.0. Oct. 16, 2024. URL: https://nicegui.io/ (page 29).

[7] Peter Hanappe et al. *FluidSynth*. Version 2.4.2. Dec. 30, 2024. URL: https://www.fluidsynth.org/ (page 33).

[8] Discord Inc. *Discord*. Version 0.0.337. Feb. 20, 2025. URL: https://discord.com/ (page 9).

[9] Jeff Johnson. *Designing with the Mind in Mind*. 3rd. Burlington, MA, USA: Elsevier Inc., 2014.

[10] Jeff Johnson. *GUI Bloopers 2.0: Common User Interface Design Don'ts and Dos*. Morgan Kauffman Publishers, 2008 (pages 13–14, 16–18).

[11] MuseScore Ltd. *MuseScore Studio*. Version 4.4.4. Dec. 11, 2024. URL: https://musescore.org/en (page 39).

[12] Joseph Malloch et al. "A Design Workbench for Interactive Music Systems". In: *New Directions in Music and Human-Computer Interaction*. Ed. by Simon Holland et al. Cham: Springer International Publishing, 2019, pp. 23–40. ISBN: 978-3-319-92069-6. DOI: 10.1007/978-3-319-92069-6_2. URL: https://doi.org/10.1007/978-3-319-92069-6_2.

[13] Don Norman. *The Design of Everyday Things*. Basic Books, 2013 (pages 8–13, 15–17).

[14] Emilia Rosselli Del Turco and Peter Dalsgaard. ""I wouldn't dare losing one": How music artists capture and manage ideas". In: *Proceedings of the 15th Conference on Creativity and Cognition*. C&C '23. Virtual Event, USA: Association for Computing Machinery, 2023, pp. 88–102. ISBN: 9798400701801. DOI: 10.1145/3591196.3593338. URL: https://doi.org/10.1145/3591196.3593338 (page 4).

[15] Frank Wen. *The Fluid Release 3 General-MIDI Soundfont*. 2013. URL: https://member.keymusician.com/Member/FluidR3_GM/index.html (page 33).

[16] Mansar Youness and Falko Schindler. *audio_recorder*. Apr. 10, 2024. URL: https://github.com/zauberzeug/nicegui/tree/main/examples/audio_recorder (page 32).