

Practice Session: A Music Tutorial Software System

by

Daniel Garubba

Professor Paul M. Jackowitz

Submitted in partial fulfillment of the requirements of

the Honors Program at

The University of Scranton

25 April 2008

Table of Contents

Introduction.....	3
Comparison to Existing Software.....	5
Requirements.....	7
Analysis.....	10
Design.....	15
CompositionComponents Classes.....	15
AudioPlatforms.....	20
Implmentation.....	21
Resulting Product.....	22
Areas of Improvement for Next Iteration.....	22
Graphical User Interface Upgrades.....	22
Coding Style.....	23
Namespace Renaming.....	25
Exception Handling.....	25
Licensing.....	26
Conclusion.....	26
Appendix.....	27
API Documentation.....	27
Glossary.....	27
References.....	30

Introduction

In recent years the rhythm-based music video game genre, including titles such as Rock Band, Dance Dance Revolution, and Guitar Hero, has gained an incredible amount of popularity. In these games, users attempt to perform tasks dictated by on screen commands synchronized to the rhythm of a musical composition. The game applies a score the user's performance based on the timing and accuracy of the user's input. However, very few, if any, of these musical video games utilize an on-screen command vocabulary based on traditional sheet music notation.

The PracticeSession: A Music Tutorial Software System thesis project begins to bridge this gap between music games and sheet music notation. This project documents the creation of a software system inspired by popular musical games, but unlike the music games that have come before it, this software system will use an on-screen command language based on traditional western sheet music notation. The system will allow users to gain ability in sight reading, practice playing compositions, as well as reinforce mastery of the instrument used to play the composition. The software built by this thesis is not responsible for providing a course of study for the users, but a platform for which the musical compositions that make up a musical curriculum can be presented and a student's progress can be assessed.

The resulting software built in this thesis is the PracticeSession application. The first release of this application will be for Mac OS X 10.5 (Leopard). PracticeSession will implement the playing of compositions created from standard MIDI files (SMF) with user performances supplied by either a MIDI device or directly recorded input, such as a microphone or auxiliary input. The software architecture is built such that more platforms such as Windows or Linux can be easily receive ports of the software. The software architecture also allows the display and playback of future composition formats other than SMF. To attain this end, the software utilizes

platform independent libraries, such as OpenGL and Simple DirectMedia Layer (SDL).

Whenever platform dependent libraries are used, such as Apple's Core Audio library for audio input and output, the software uses the appropriate design patterns and object-oriented programming (OOP) concepts to decouple the central application from the platform dependent libraries.

Although PracticeSession lacks the polish in presentation of a commercial musical game or music education software suite, its initial state illustrates the promise of a larger, more capable software system. The scope of this thesis allowed for the design and implementation of the software project to carry it out of its nebulous state. New additions to the system can be road-mapped for future development. Additional developers can work on various aspects of the system, such as implementing a new audio platform, without knowledge of the system as a whole. Though the achievements of the existing system are substantial, the real accomplishment of this system lies in its extensibility and its potential growth in the future.

Comparison to Existing Software

The main inspiration for this software system is the popular music video game genre. The popularity of these games are undeniable. In January 2008, Activision announced that its Guitar Hero franchise had exceeded \$1 billion in world-wide sales¹. In these music games, users are prompted to perform on screen commands in rhythm to a synchronized musical composition. For example, in the Guitar Hero series, a series of five circles appear on the bottom of the screen, each representing a button on the guitar controller utilized by the user. Above this series of circles is a patterned strip with five parallel lines extending into the background

Although the main inspiration for this software system are the popular music video games, there are existing interactive music education packages. However, many of these software packages, like those available from ECS Media, Sibelius, and Alfred Music focus mainly on music theory, but not the assessment of a complete performance of a musical composition.

The software system that this thesis entails would have a unique niche if given public release. It's purpose is not to offer an alternative to the already established educational software, nor is it attempting to be the next Guitar Hero. The aforementioned music education software vendors offer large music theory curricula within their software suites, but the scope of this project does not allow this comprehensiveness. Music games aim to entertain, but the skills built by playing these games are largely most valuable within the game itself. That is, skills built playing Guitar Hero will not directly translate to skills playing a real guitar. Rather, the software created by this thesis project aims to create a platform by which users can play musical compositions represented in sheet music notation and have their performances judged. A music

¹ Graft, Kris. "Guitar Hero Breaks \$1 bln". Next Generation. 21 January 2008. http://www.next-gen.biz/index.php?option=com_content&task=view&id=8746&Itemid=2.

curriculum could be developed around this software as a series of compositions or exercises much in the same vein as music lesson books are constructed. The user experience of playing a musical composition should be similar to that of playing a one of the aforementioned music games.

Requirements

The created software for this project is conceivably large. The scope, if not controlled, could be beyond what would be reasonably attained in the time allotted for the project. Therefore, establishing a hierarchy of requirements is essential for managing the feasibility of project completion within the given time frame. This activity, requirements triage, is “the process of determining which requirements a product should satisfy given the time and resources available².” Although this project doesn't have the same concerns as Davis did in his article, such as easing the tensions with a marketing department, it is still important to prioritize and evaluate the feasibility of the requirements during the software system's development. There are many domains in which this software system requires some basic knowledge, like input handling, graphics rendering, audio playback, signal processing, etc. It is almost impossible to capture all of the requirements accurately initially.

Chief among these Requirements is the ability to play with a composition specified by a Standard MIDI File while receiving feedback from the system about the accuracy of the performance in the terms of a percentage. This is the central functionality of the system. The system will display this composition that is being played along with in terms of traditional sheet music notation. The system will indicate what notes of this sheet music representation are currently being played in the composition visually by indicating what the currently active notes and rests are and what is the current position of the of the composition being played. The software should also indicate what are the notes currently being played by the user.

The user can supply input either by using a MIDI device, like a keyboard or MIDI pickup apparatus for a guitar, or by an audio capture device for a pitch analysis over sampled audio.

2 Davis, Alan. “The Art of Requirements Triage”. IEEE Computer. March 2003.
<http://ieeexplore.ieee.org/iel5/2/26595/01185216.pdf?arnumber=1185216>

The menu system interface shall display menu items on staves so that the instrument used can perform menu navigation. The user can configure the input device such that staves drawn with the appropriate clefs. The input and as well as the displayed composition can be transposed as appropriate for the device.

The initial release of the system may support only one operating system platform, but it must the architecture must be built for portability. This is not really a functional requirement but more of a statement of a guiding principle of how the system should be built: the system's greatest strength should be its extensibility. The goal of the system should be the creation of the beginning step of a potentially greater platform by which users can play along with compositions in multiple formats with a richer user experience, more sophisticated performance analysis, and more customization for the playing session.

Because of the risk involved in working in so many various domains, some of the requirements specified in earlier requirements documents were dropped or modified as the breadth of the system became more realized and the time constraints prevented a greater initial realization of the system. One of these requirements was the establishment of a generic Performance Assessment Engine which the PracticeSession would utilize for the comparison of input to the supplied composition. Inspired by the similar functionality of music game software, this PAE was to provide for a frameworks on top of which other rhythmic applications were to be created. However during the design process, the timing of the system was determined to be based on the CompositionPlayer, not a generic Performance Assessment Engine. However, the Assessment package does reflect some of the concepts that were employed in the early planning of the the system, such as the Assessor class for comparing input to the composition.

Another early requirement that could not be implemented was some of the options that

the user could specify for an input device. The reason for this is that the custom MenuSystem, based on the display of information on staves, required the creation of a custom GUI. Since this interface was drawn largely from scratch, its data entry capabilities are limited, and the user customization was limited. For example, instead of selecting custom transposition values and specifying which staves to use for rendering compositions, the system instead uses presets due to the limitations of the user interface.

Another dropped function of the system was the users ability to supply his own MIDI file to play along with. This is due to the fact that browsing a file system is a platform dependent task, and time constraints prevented the implementation of the interface for browsing files from the user interface. However, if this functionality were to be eventually implemented, the MenuSystem would be capable of browsing the user files. Similarly, due to time constraints, the different modes for a the playing session, such as targeting specific bars of a composition or automatic repetition of failed sections of a composition were not implemented.

Analysis

After establishing the requirements, high-level organization of the software needs to be established. Subdividing the responsibilities of the PracticeSession application into smaller packages will aid in the design process. However, it is important to recognize any potential external libraries that may aid in the process of supporting these packages

In order to satisfy the requirements established above, it is also important at this stage to determine what external libraries would be necessary for the completion of the software system. Since building the system for maximal portability is a prime consideration, selecting libraries with this capability is essential. However, given the multimedia requirements of the project and the desire for performance, it is important that the software utilizes libraries that are close to the hardware, but still have a reasonable amount of portability.

One library, mainly used for games and multimedia applications stood out as being extremely popular for its portability and its . The Simple DirectMedia Layer (SDL) library, is utilized by emulators such as DOSBox, and games like Battle for Wesnoth, and multimedia players like VideoLAN Client (VLC) Player. Many game development tutorials, like those found on lazyfoo.net and the Game Programming Wiki use SDL in their exercises. SDL provides subsystems for handling audio, video, input, events, and threads. There are ports of SDL for all of the dominant platforms, like Windows, Mac OS X, Linux, and many flavors of Unix. SDL's popularity can be measured in the amount of applications featured on the site, including over 500 games. Using SDL would provide a familiar platform for many future developers who may want to contribute to the project, especially those familiar with game programming. Moreover its many services minimize PracticeSession's reliance on other external libraries. Furthermore, SDL integrates easily with another extremely portable library used for

graphics.

OpenGL is the most widely used 2D/3D programming API. It claims to be the industry standard, with its availability on all dominant platforms. High performance rendering with hardware acceleration would be attainable using OpenGL. Again, any potential future developers may have considerable OpenGL skills. Although the graphical presentation of the initial iteration of PracticeSession is expected to be minimal and rather spartan, this author's mastery of OpenGL as well any collaborations with future developers could make use of greater use the breadth of OpenGL's API for a more satisfying visual presentation. Furthermore, OpenGL does not require any licensing for use in any projects. Finally, SDL programs can also make direct OpenGL calls for hardware accelerated rendering.

With just SDL and OpenGL, much of the functionality of the system can be implemented. SDL can provide input handling for devices like mice, keyboards, and joystick, as well as handle events, use timers, and provide for an application window. Because it performs these functions, the software will not make too much use of the GLUT library's capability for supplying similar functions. OpenGL will provide high performance rendering, if only initially making use of the primitive rendering features. However even though SDL provides for some audio functionality it does not have any MIDI input, hardware MIDI synthesis, or sampled audio input. Being that this functionality is essential for meeting the most important requirements of the system, it is imperative that a suitable library is found that could provide this functionality. However, a search for a platform that provides for an easily portable and ubiquitous audio input/output interface proved to be in vain.

While performing a tutorial for using audio in SDL via the SDL extension `SDL_mixer` on lazyfoo.net, audio playback of a standard MIDI file sounded remarkably different (and in this

author's opinion, less pleasing) than the MIDI synthesis provided by the DOSBox emulator. The disparity for this difference in sound synthesis was caused by the fact that SDL_mixer provided sound synthesis by the TiMIDIty software MIDI synthesizer, while DOSBox provided a hardware implementation for General MIDI drivers within its DOS emulation. Looking at the source code for DOSBox revealed a dependency via conditional compilation (enclosed in `#ifdef __MAC_OSX__` preprocessor block) that MIDI synthesis was performed via the CoreAudio libraries in Apple's Mac OS X operating system. Researching the CoreAudio API available to Mac OS X revealed that CoreAudio contains all of the facilities needed to perform the audio input and output of the system. Although CoreAudio would be only used by the Mac OS X implementation of the system, this hardware dependency informs the architectural design of the system. All audio processing will be implemented by platform dependent libraries that should be wrapped to interfaces to separate the platform dependencies from the core, portable features of the system.

Because C is the native language of all of the APIs mentioned above and object-oriented programming would greatly benefit the extensibility of the system, C++ is the language of choice for this software system. Using an object oriented language should allow for the separation of the different features of the system into larger, dependent units. C++, however, does not have a package feature found in Java or Python. Instead, the namespace feature of the is utilized to illustrate the different domains and responsibilities of interrelated classes. Each set of classes sharing a domain are grouped into the same folder and share a namespace of the same name. Example namespaces include AudioPlatforms, Rendering, Input, and CompositionComponents. The following table lists the identified namespaces and their respective responsibilities.

Namespace	Responsibility
Application	Contains the application class, which coordinates with all of other classes in the system
Assessment	Contains classes for comparing current input to the current position of the played composition and maintains a score
AudioPlatforms	Creation of a group of platform dependent audio classes
AudioPlayback	Manages the playback of audio devices
CompositionComponents	Contains an object model for the visual representation of compositions
CompositionPlayback	Playing of Compositions and tracking the position of the composition during a playing session
CompositionTranslation	Classes for converting files of different formats into
Input	The processing of input from both audio devices and computer devices like keyboards
MenuSystem	The navigation between menus for configuring a playing session
Profiles	Keeps track of the configuration and status of a user
Rendering	Graphical representation of the system
Utilities	Miscellaneous classes for objects like container types and design pattern interfaces

Table 1: Namespaces for the PracticeSession system

The Preliminary Package diagram illustrates all of the namespaces utilized by the system. At the center is the Application namespace, which contains the central program logic. It is responsible for controlling all other theoretical “packages” of the system. The diagram illustrates the dependencies that were found prior to the complete design of the system, where more dependencies through the specification of classes were identified.

Preliminary Package Diagram

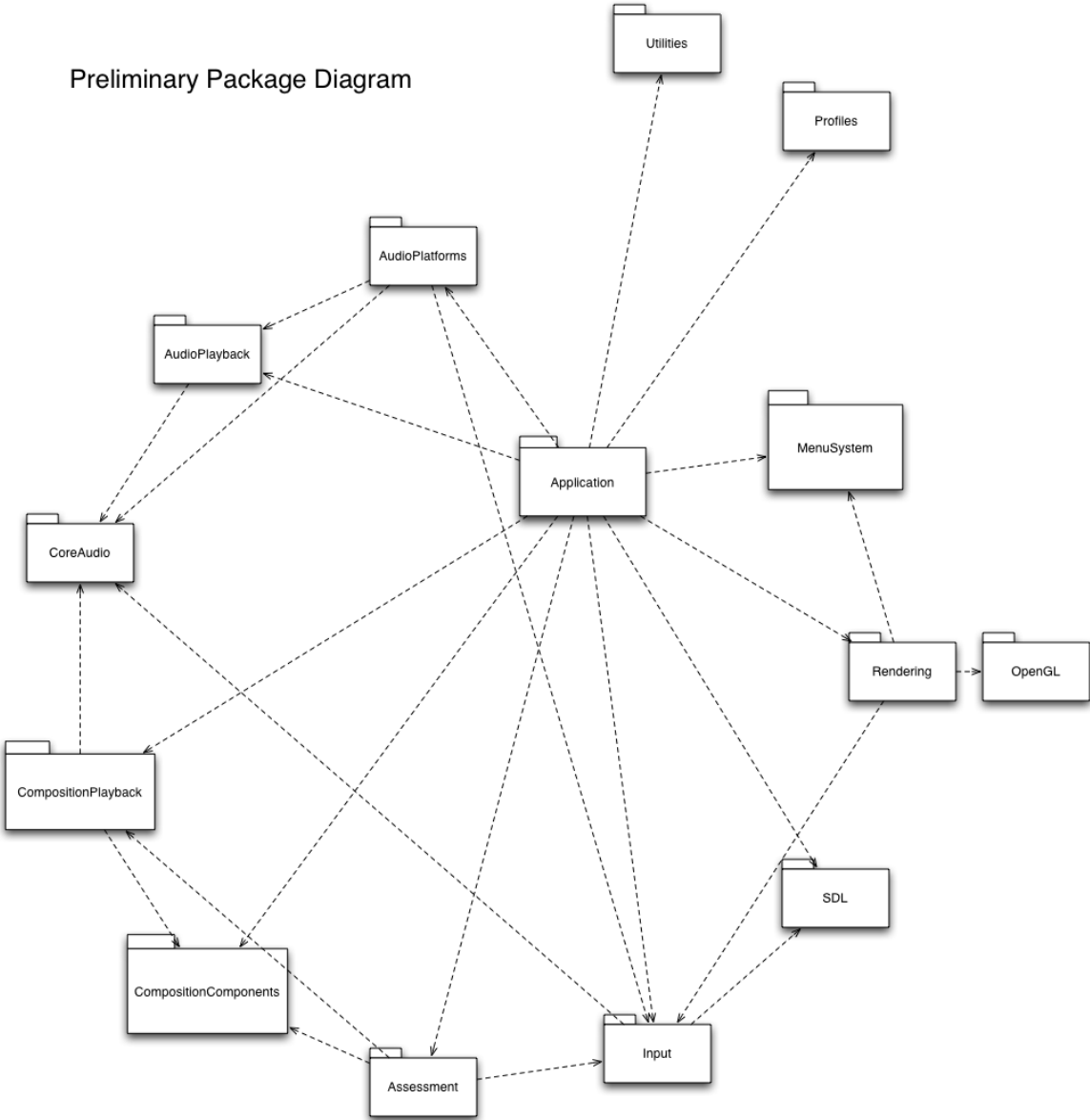


Illustration 1: Preliminary Package Diagram

Design

After the essential software components are identified from the requirements, classes must be specified for the design of the system. Note that in the following section words that have their first letters capitalized may be series of words without spaces and inner capitalization represent concepts that are modeled with classes. Any words that are in all caps are concepts that are modeled as constants such as an class attribute that is modeled as an enumeration (utilizing the `enum` keyword in C++).

CompositionComponents Classes

The first body of classes that were specified were the classes of `CompositionComponents` namespace. These classes specify the visual representation of the compositions. The `CompositionComponents` also provides a fixed hierarchy of classes with objects encapsulated by object composition. For example, at the top of the hierarchy is the `Composition` class. This class contains a set of `Parts`, as well as a singular `universalPart`, from which every added part copies its structure. The `Composition` class also contains a `CompositionInfo` object, that contains meta-data about the `Composition` object. Within each `Part` of the `Composition` there may be one or more `Phrases` of music. The `Part` object also keeps track of the name of the `Part` and other attributes, such as if the part is transposed. Each phrase of music has a name and contains a number of `Measures`. A `Measure`, equivalent to a bar of music, keeps track of its `TimeSignature` and its `KeySignature`. A `Measure` also encapsulates any `Tempo` changes or `DynamicIndications` may be present within the measure. The `Measure` class also contains contains a number of `Staves` (i. e. multiple `Staff` objects) onto which the actual composition is transcribed.

The `Staff` is most likely the most sophisticated object within the `Composition` object model, since it has to more work to maintain the consistency of its composite objects it contains.

The Staff contains a Clef, which keeps track of its position on the staff and what type of Clef it is (e.g. G Clef, F Clef). The Staff also keeps track of all of the NoteOrRest objects that are to be drawn on the staff. The Staff maintains the constraint that the Duration specified by its containing Measure's TimeSignature is fully covered by NoteOrRest objects.

When a Staff is first created, it contains only Rest objects. The Rest class, as well as the Note class and Tuplet class are all subclasses of the NoteOrRest class. Initially when the NoteOrRest class was first identified, its only subtypes were the Note and Rest classes. However, as it became apparent that keeping track of membership in a tuplet (e.g., which groups of Notes make up a full triplet) as well as doing the fractional arithmetic that maintains the constraint of fully populating the Staff, it was discovered to be convenient to maintain a Tuplet object.

Each NoteOrRest object maintains its own Duration, which is a fraction representing how long a NoteOrRest object is active. A Duration is nothing more than a fraction with an integer numerator and denominator. The valid Durations may vary for each NoteOrRest type. See the Table 2 which illustrates the valid NoteOrRest Duration values. The Durations in this table are represented as exponents rather than fractions to more clearly illustrate the importance that powers of 2 have for representing the Duration of NoteOrRest in a given composition. The NoteOrRest object also maintains a toDuration value which reports how much time has elapsed from the beginning of a Measure to when the NoteOrRest is active. For example, if a quarter note begins on the third beat of a Measure with a 4/4 TimeSignature, it would have a toDuration of 1/2 and a Duration of 1/4. Finally, a NoteOrRest has a function for the toEndDuration value, the Duration that had passed from the beginning of measure to the end of the NoteOrRest. It is the sum of the toDuration and the Duration of the NoteOrRest.

NoteOrRest Subclass	Valid Duration	Discussion
Note	$3^{0..1} * 2^{-\infty..+2}, 4$	A note can have a largest value of a dotted Quadruple note, which is a would be a Duration of 6/1 (i. e., $4 * 3/2$). The Duration may be a multiple of three for the representation of dotted notes. To represent a Note of a Duration that is not valid per this specification, it must be created by the placing notes whose sum of Durations equals that irregular Duration in a TIE Relationship. Notes must also be placed into a Tie relationship if a Note's Duration makes it end after the time allotted by the Measure's TimeSignature.
Rest	$2^{-\infty..+4}$	The Rest's Duration should only be a multiple of 2. Dotted Rests do exist in sheet music notation, but they are usually discouraged for readability and can be composed easily from the sum of smaller rests.
Tuplet	$2^{-\infty..+4}$	A Tuplet has specified the number of TupletMembers that it contains. The most common Tuplet is the triplet, which has three three TupletMembers. Each TupletMember has a Duration of the the entire Tuplet divided by the number of Members in the tuplet. For example, if a Tuplet represents a set of Triplet quarter notes, it the Tuplet would have a Duration of 1/2 (i.e. equivalent to a half-note's duration). Each member of this Tuplet would have a duration of 1/6. A TupletMember may either be a TupletRest or a Tuplet NoteGroup. A TupletNote group may contain a one or more Notes. However, because a Tuplet is a subclass of the NoteOrRest class, it is constrained not to span over more than one measure (i.e. the system currently does not support the

Table 2: NoteOrRest durations. Note that any values of ∞ seen above are strictly theoretical bounds and in implementation are constrained by the integer type used to represent the numerator or denominator of a Duration.

In addition to its Duration values, a Note encapsulates some attributes about itself. It contains a staffPosition value, which is an integer representing how many lines and spaces a Note is away from the central line on the Staff. It also maintains an Accidental value to indicate if the Note is SHARP, FLAT, NATURAL, or if it has NO_ACCIDENTAL at all. These items, in addition to the Note's encapsulating Measure's KeySignature determine what pitch a note represents. A Note may also have one or more Articulations, but this is more or less a

placeholder attribute for a more sophisticated iteration of the Composition Object model. The Standard MIDI File specification does not have any such information for the Articulation of Notes, although some extensions to the MIDI standard have been proposed to convey this information. Finally a Note encapsulates any Relationships that it initiates. A Relationship maintains a list of all the other Notes that are involved with the relationship. Such RelationshipTypes of the Relationship might be a SLUR or a TIE. The Relationship is the only member of this Composition object model that utilizes aggregation rather than composition with its encapsulated members, as a Note's may existence is not reliant on its membership in a Relationship.

Illustration 2 show the diagram for the Composition Object Model for the classes in the CompositionComponents namespace. Note that the member and function lists for the classes in this diagram have been omitted from this list for the sake of simplicity.

Composition Object Model

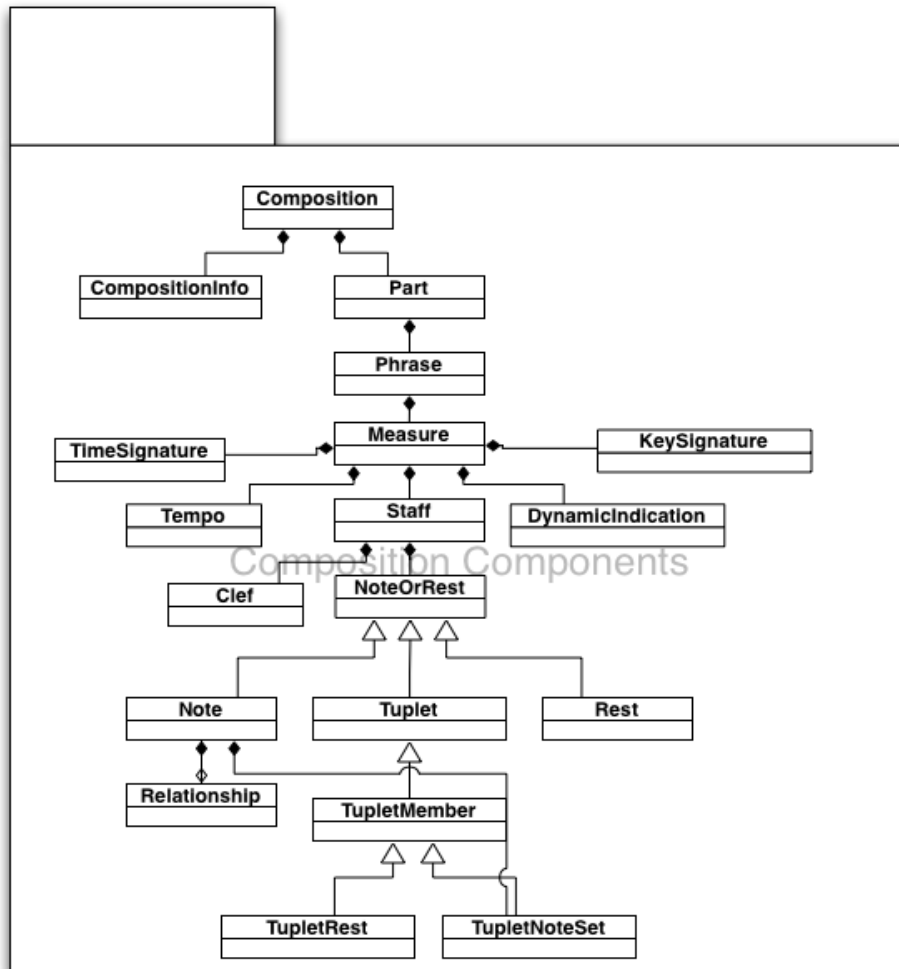


Illustration 2: The Composition Object Model illustrated by classes in the CompositionComponents namespace

AudioPlatforms

The Gang of Four's Design Patterns text recommends the use of the Abstract Factory design pattern for the use of classes that may be platform dependent. Being such, the AudioPlatforms namespace mainly serves as the container for the PlatformFactory abstract factory class and its realization class, the CoreAudioFactory. Figure

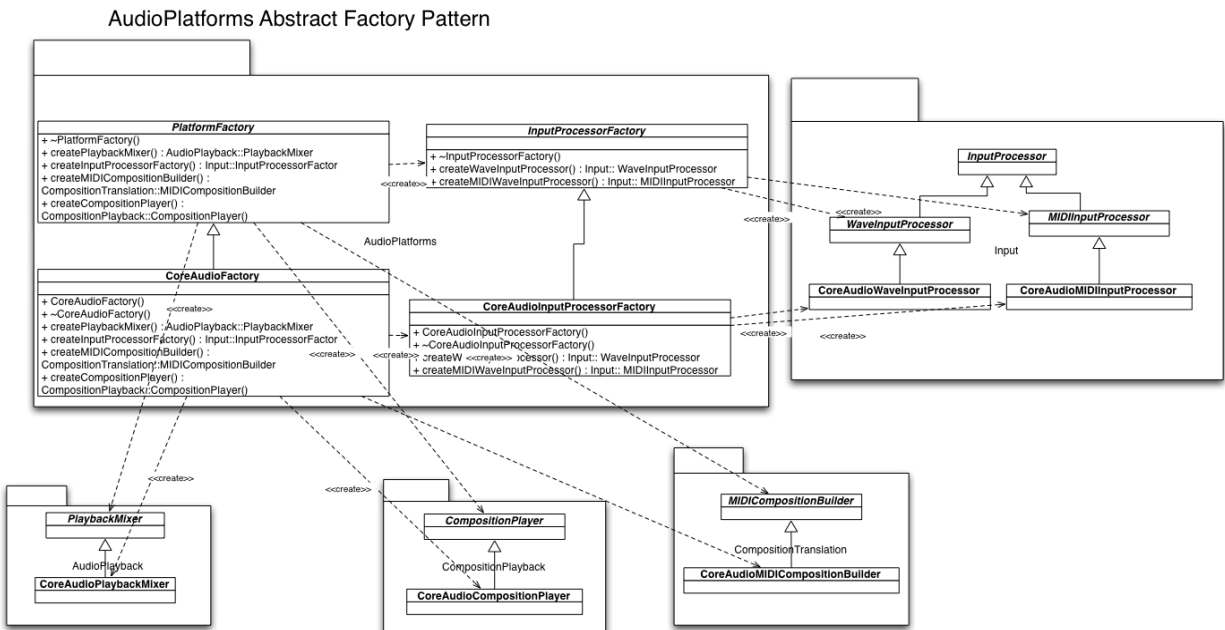


Illustration 3: The Abstract Factory pattern of the PlatformFactory

Implementation

This section will describe the process of coding the software and will describe what software tools were utilized and some of the informal Unit tests that were performed as the system was being implemented.

Resulting Product

This section will describe the functions of the system that were implemented at the completion of the Thesis and will provide some screen-shots of the software in action.

Areas of Improvement for Next Iteration

The PracticeSession application in its current state is not quite ready for commercial release. There are several areas of improvement that would expand the functionality of the platform greatly. The following sections identify the deficient areas of the system and state what possible steps could be made to make the system better suited for public release.

Graphical User Interface Upgrades

The menu system is limited in that you can only select MenuItems in a list format, which execute commands when executed. There is no means yet for the input of other data items, like numeric data entry, or text entry. This means that some of the flexibility of the system, such as setting a custom value for a transposition value or setting the number of staves on which the music is displayed, is hidden behind only selecting a finite number of presets, such as piano (two staves, not transposition value), guitar (one Staff with a treble Clef, transposed down an octave), bass (one Staff with a bass Clef,), trumpet (one Staff with a treble clef, transposed down two half-steps).

With the re-evaluation of the MenuSystem, there are several factors that need to be addressed. One is the extensibility of the component members of the MenuManager class. If the MenuManager were expanded to be a general GUI manager, with each member being a subclass of a generic Component class (much like Java's Swing or AWT framework) and following the Composite design pattern, there is much work that must be done to support this design. For example, in its current state, the Menus have no knowledge of their visual representation aside

from screen position, size, visibility, and which MenuItem is highlighted. Their visual representation is solely the responsibility of the Renderer. In order to make this system more extensible, components of a GUI manager must contain information on how to draw themselves so that a Renderer can display it without knowledge of which Component subtype the object is (à la the paint() method in Java's GUI frameworks). However, to support this feature, there must be a rich Interface of low level painting functions such that the Renderer can represent the general Components without any knowledge of the implementation of the Components. The Renderer must also be expanded to implement all of these painting functions.

Because of the complexity of creating a general GUI system, it may be worth investigating the possibility of integrating a widget toolkit like Trolltech's Qt (pronounced “cute”) to handle GUI functionality. However, there are many issues that would need to be resolved, such as if Qt can be used with the graphics rendering using OpenGL on an SDL_Video surface. Should the rendering be moved to an implementation in the Qt OpenGL 3D Graphics model? Are all of Qt's functions available in the open source GPL version? These are the issues that need to be addressed before Qt could be embraced as a platform for GUI features.

Coding Style

During the development of the system, this author became increasingly proficient with programming in C++. Coming from a primarily Java background, this software initially used reference return types for every getter method used. However, as the system continued to be implemented, more pointers types were being employed. The code should be evaluated such that there should be a uniformity of cases of when to use reference return types and when there should be pointer return types. For example, as a proposed practice, all getters which return

objects rather than built in types should return references, while all factory methods, which create objects with dynamic allocation, will return pointer types. This sounds like a simple principle, but what if an object member accessible by an accessor method can be validly represented as NULL. Would it be better to have the appropriate getter return a NULL type? Does this reflect any implications of using object aggregation vs. composition? These questions need to be considered when defining a guiding principle for deciding when to use pointers and when to use references.

C++ also offers some keywords that may be better utilized as the code is refined. For example, the `inline` keyword might be employed for greater optimization of the code. `const` correctness is also an aspect of the code that should be evaluated as soon as possible. According to the C++ FAQ Lite, it should be done at the “very, very, very, beginning.”³ Although the code is fairly large at this state, the `const` correctness should at least be implemented at as much as possible before public release.

Finally, the code should be modified for enhanced readability. Although it adheres to some conventions, limiting lines to 80 characters and careful use of ++ operators, there are plenty of areas for improvement. For example, the open bracket '{' character may be better placed on a separate line rather than maintained on the same line as a function or class declaration. Since the argument list of functions are long (most often because the arguments types are prefixed by namespaces), this would be useful for more easily indicating where the function declaration ends. Also because of these long function signatures, the argument list would be best separated on multiple lines, with one line per argument. The Xcode IDE does not feature the same code formatting features that Eclipse has (it can handle indenting, but there is no equivalent to the Source->Format command from the Eclipse's editor), so importing the project

3 Cline, Marshall. C++ FAQ Lite. <http://www.parashift.com/c++-faq-lite/const-correctness.html#faq-18.1>

into Eclipse for the sake of code formatting might be a worthwhile venture.

Namespace Renaming

As mentioned before, C++'s namespace feature was employed in the use of representing packages of software. Although the namespaces employed accurately portray what responsibilities each conceptual package does, the names are not specific enough to prevent the issue of “namespace pollution”. Namespace pollution could occur if the custom libraries created specifically PracticeSession utilize namespaces that other libraries may use. For example, if a developer were to port PracticeSession to another system and wanted to use additional libraries to support the port, a namespace with a generic name such as “Utilities” or “Rendering” from PracticeSession's set of libraries may provide a conflict with any additional imported libraries. Correcting this could be done by performing a find/replace script to replace all known namespaces in the system with a PracticeSession prefix, such that “Utilities” becomes “PracticeSessionUtilities”. After successful namespace renaming, the folders containing the classes of a namespace can be renamed to match the new namespace name and reimported into the IDE.

Doing this Namespace Renaming is simple in concept but its execution may become tedious. With longer namespace names, all of the source code readability may have to be reevaluated such that the lines are not too long (the source code generally adheres to the guideline that lines be no longer than 80 characters).

Exception Handling

Libraries used for the initial development of the system (namely, CoreAudio and SDL) report when errors occur when calling their respective functions. However, since these libraries were written in C, they do not throw any exceptions. Rather, they return error codes. Although

the software, which is for the most part a closed system at this state, does not evaluate the error codes returned by the SDL and CoreAudio functions, they serve

Licensing

Before the PracticeSession system makes its proper debut as an open source platform, the appropriate licensing issues must be addressed. SDL is a GNU Lesser General Public Licensed (LGPL) library, so there are obligations that I must fulfill to when distributing the software in the form of license duplication. However, there may be advantages to using the GNU General Public License (GPL). For instance, there may be a segment of the open source community who do not want to contribute to a project that may be used in commercial applications. These developers may take the aims of the Free Software Foundation very seriously and may never want to develop software

Conclusion

PracticeSession was built with the intention of being a constantly growing system. Even though it supports compositions defined in MIDI files, if more developers became contributors to the project, they could expand the functionality of the system greatly.

Appendix

API Documentation

This section is to be supplied after generating the HeaderDoc (Apple's Javadoc inspired automated documentation software) from the source code. This should account for roughly 40 pages of text.

Glossary

Abstract Factory – A design pattern that is used for the creation of a family of objects. In this pattern, an Abstract Factory interface specifies what Factory methods are to be used for the creation of this family of objects. In the PracticeSession, this pattern is employed for the creation of a family of objects based on the platform. In the case of the existing system, the platform dependent audio processing was handled by PlatformFactory implementation that supplied factory methods to create objects that with Apple's CoreAudio platform

Builder – A design pattern that is used for the construction of a complex object. The Builder pattern is employed by this software to construct Compositions from Standard MIDI files.

Composition – The data object that represents a musical composition. The nomenclature of “score” was avoided because:

1. This is a term used by music scoring software, i.e. software that allows users to create compositions.
2. So there would not be any confusion with the concept of a score as in the metric used to rate a users performance.

This is the top level of the object model for sheet music representation. All musical compositions represented in other file formats need to be converted to this format. Future

releases of the software should virtualize the functions of the Composition class so that compositions from different source formats, like the Amiga MOD format can be represented as a sub-class extension of the Composition object, rather than built concretely as the existing software does for Standard MIDI Files.

Composition Object Model – The hierarchy of classes that represent a musical composition.

These classes are specified in the CompositionClasses namespace. At the top is the Composition object, which contains the Parts of the composition. Each Part specifies a transposition value and contains a sequence of Phrases. Each Phrase contains a series of Measures. Each measure contains the Staffs onto which the Notes and Rests reside. This object model employs composition rather than aggregation for the creation so when a containing object is destroyed, so are all of its containing objects.

Factory Method – A design pattern that defines an interface for creating an object, but lets the subclasses determine what

MIDI Device – A hardware device that communicates with other musical devices via the MIDI protocol. A MIDI device is one of two methods by which users can input performances to the system.

Musical Instrument Digital Interface (MIDI) – Often a term with multiple meanings, MIDI formally is the protocol by which musical hardware devices

General MIDI – The specification that establishes a set of 128 universal instrument-number assignments. The software uses this specification for establishing Part names in conjunction with the the Meta-Data at the beginning of a track.

Simple DirectMedia Layer – A LGPL licensed library that is frequently used in open source game projects. It allows for easy portability between multiple platforms and interacts directly

with OpenGL. It is written in C and provides interfaces for video, input, event handling, threading, and semaphores, and has extensions for audio and font display.

References