ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

J - Moog

Project Report

Object Oriented Programming Course

student:
Renato Panebianco

Professor:
Mirko Viroli
Matteo Casadei

Assistant Professor:
Danilo Pianini
Angelo Croddai

March 1, 2016

Abstract

This report is related to the planned project for passing the object-oriented programming exam.

The project described below aims at the construction of a simulacrum of *Micromoog* synthesizer exploring the capabilities of java.sound.midi and java.sound.sampled packages.

The main purpose of this study is to develop a software that has on one side the bases for simple further developments, on the other, to provide cues to apply the design techniques acquired during the course.

Contents

Chapter 1

Analisys

The project Micro J-Moog aims to make a music synthesizer, specifically a simulacrum of MicroMoog that allows to apply the design techniques learned during the course, through the use of the main features offered by java.sound.midi and java.sound.sample packages.

1.1 Requirements

The J-Moog synthesizer shall provide the following basic functionalities, which can be classified mainly into two categories:

- **MIDI**(Music Instrument Digital Interface) where production of sounds using the existing samples in the sound card of a system, also using external devices (for example a keyboard) that interact with the software in question via the exchange of messages according to above MIDI protocol.
- **audio** where the sound is obtained through the algorithmic generation of specific digital signals with specific characteristics (for example, a sine wave) that are sent to the sound card of the host system.

Both of the above categories shall be available both through a proper visual interface (such as for example a Piano keyboard module that responds to mouse clicks) and either through external device (such as a USB MIDI keyboard).

1.1.1 MIDI requirements

- Open files in MIDI format and manage their listening, forward, stop.
- allow the selective listening for single (or multiple) track of these files in order to analyze the part performed by each individual instrument.
- generate new MIDI files, with the possibility of registration of a predetermined number of tracks, giving the possibility to choose, by means of an appropriate interface between the sounds (eg. Piano, violin, etc. Etc.) obtained from the set available on the sound card host system.
- ability to save files obtained.

1.1.2 AUDIO requirements

- generate "from scratch" *waveforms* among the most common in analogue sound synthesis (for example, sine, square, etc. etc.).
- change the sounds acting through a suitable interface (for example an equalizer) on the individual components of the considered sound (harmonics), in terms of phase of the signal or of its amplitude
- provide polyphony in the obtained sounds, or ability to generate multiple sounds simultaneously on different sequences (notes).
- manage the sounds obtained in terms of I / O giving the possibility of being able to save to disk and retrieve them when necessary.

1.2 Analysis and Model's Domain

In order to implement the above functionality (MIDI and AUDIO) J-Moog will consist of a core (Kernel) that will make use of some components.

About MIDI functionalities the J-Moog will need to use several of the host system's audio sound card resources.

According to this purpose, the main components are made of a *synthesizer* that interacts with the core of the host system MIDI sound card, and a *sequencer* that will be responsible for managing the operations on MIDI files (which are sequences of sound *events*, such as the execution of a note at a given time).

As regards as the AUDIO function, attention will be mainly aimed at ensuring that the produced sounds have a low latency, which consists in the potential delay between the request for enforcement of a sound (immediate) and its availability due to the computational cost that the algorithm that generates it might take. This aspect must also be deal with the requirement of polyphony (playing multiple notes simultaneously).

Quite a lot of time has been spent in search of of existing documentation (and then for studying it...) about the domain in question and in order to acquire the needed minimum necessary preliminary knowledge to implement key parts of J-moog. In this sense, indeed, the existing official documentation was found largely insufficient and too general relatively to the specificity of much of the application context (specially for MIDI sequencer functions).

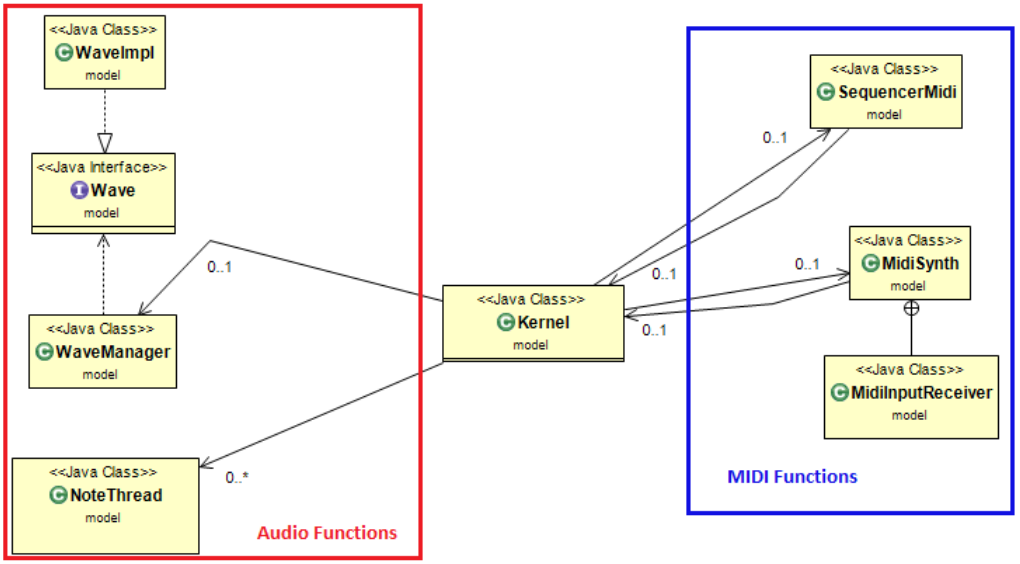

Figure 1.1: This picture shows main components of the system's core.

Chapter 2

Design

2.1 Architecture

In order to implement the above mentioned functionality it has adopted architecture *Model - View - Controller*'s architecture. This three parts have each the following function:

- **Model**: is composed of a Kernel that use mainly three components: a *synthesizer* , a *sequencer* and an *audio core*. the first gives access to the resources provided by the sound card in the host system (sound, MIDI settings, MIDI ports).

  The second manages the various possible operations (run, stop, record) of a MIDI file (which can be thought of as a sequence of sound events each with their own execution moment in a time line)

  the latter handles the features to produce a digital sound through a specific algorithm (related to wave form) and send it to the audio line of the host system sound card.

- **View**: is composed of several parts, the most important are: an equalizer and a player. The equalizer allows to produce sound by mean of different components (harmonics) , the player allows to manage some operations on a MIDI file (start, stop, update a progress bar , mute tracks, ecc. ecc.)

- **Controller**: is the interface between the View and the Model, and in addiction handle almost everything related with threads and I/O onto disk.

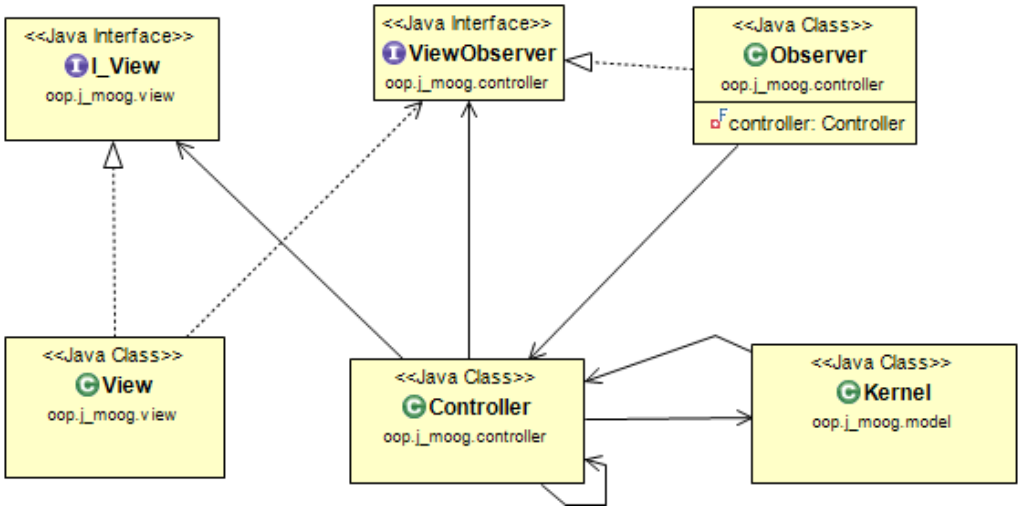

Figure 2.1: This picture shows the adopted *Model - View - Controller* architecture

2.2 Detailed Design

Preliminary considerations

According to the adopted MVC paradigm, and appropriate attention has been paid so that the part Model - Controller was always independent of the View, and in particular that the latter exchanged method calls with the Controller only. In addition to the classes / interfaces that represent main entities of the system some *Enums* (better explained in the next sections) have been adopted, in order to make easier to implement functionality such as file I/O and View building. Nested classes and patterns has been used wherever possible in the attempt to improve complex parts (e.g. the Wave Manager in the Model or the equalizer in the View).[1]

2.2.1 Model

As said in previous sections, the Kernel of the Model uses three components (synthesize, sequencer and audio core). It manages them in order to provide functionality of two types: *MIDI* and *AUDIO*.

[1] In the next sections of this chapter involved classes names will be highlighted with italic font.

MIDI functionality

This part of the system is shown in the next figure, where has been focused how the *kernel* interacts with the *MidiSynth* and *SequencerMidi*. It is worth to mention that starting from version 1.4 of the JDK the java.sound.midi package brings together into a single object the sequencer and synthesizer of the sound card. In other words, the Sequencer, in order to access the sound card's sounds (for example to execute a MIDI file) also it implements Synthesizer's Interface methods, in such a way that : Sequencer instanceof (Synthesizer) == true

However, under this project, they have been considered and implemented as two separate objects:

- first of all for architectural clarity in order to avoid to build a *God Class*, given the already numerous methods of each of the two classes separately taken, were many methods are required to handle the minimum functionality (MIDI channel selection, track, instrument, single track muting and multiple etc. etc.).
- secondarily to provide greater portability of the application.

Therefore, the *Kernel*, according to user choices manages the *MidiSynth* instance. This class wraps the *MidiSystem* class of *javax.sound.midi* package which through static methods allows access to available system audio and midi resources. Management is similar to the *SequencerMidi* class, which is a wrapper for the host system sound board sequencer (also provided by the above mentioned *MidiSystem* class).

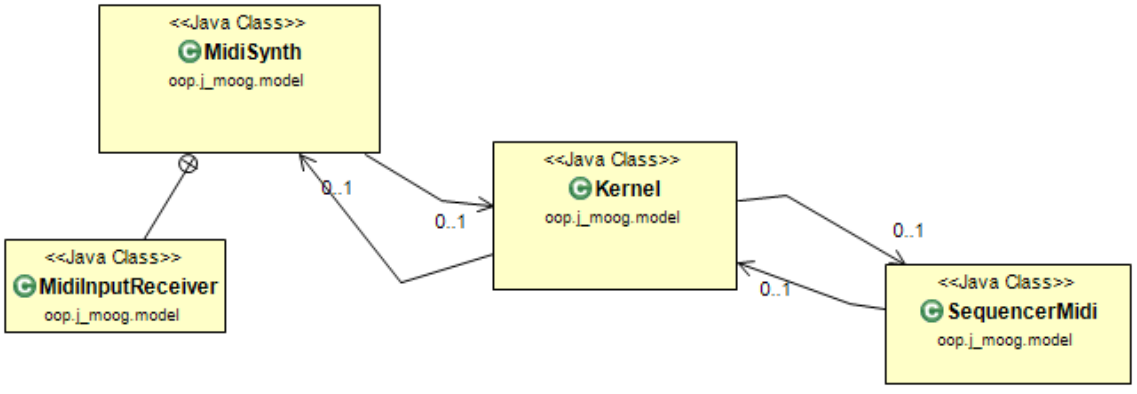

Figure 2.2: Main MIDI functionalities classes.

The class *MidiInputReceiver*: an application of decorator pattern to use an audio core other then MIDI through an external USB MIDI keyboard.

It deserves further explanation the class *MidiInputReceiver* nested in *MidiSynth* class (see 2.2). By default the sound board's synthesizer has its own MIDI message receiver, that catches midi message coming for example from other devices such as external USB keyboard.

Each of these messages is a *midi event* (such as a change of used instrument, pitchBand, keypress etc.), which can be intercepted by means of listeners the implement specific interfaces of the java.sound.midi package.

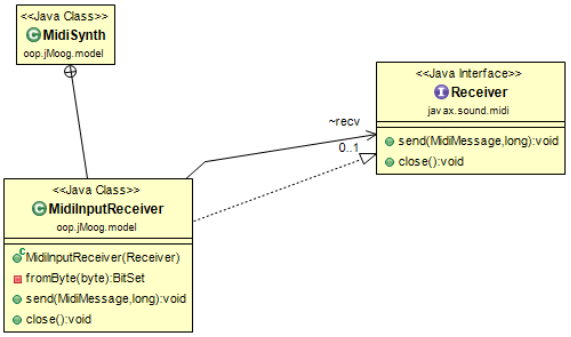

Figure 2.3: the decorator pattern adopted.

However, as reported in the javadocs on the subject, these listeners are implemented only for program changes (sounds), control changes (eg. Volume) but not for events such as *Note On* and *Note Off* messages, so it is not possible via standard listeners intercept pressed Keys events with a different device that uses a protocol other then MIDI (for example a third part audio core like the J-Moog when it produces its own sounds).[6]

After several attempts, the found workaround has been to implement a *decorator pattern* which it is possible to wrap the synthesizer receiver with. Each time a note is pressed, it triggers the *send()* method where you can nest a call to a method related with your own audio core.

This trick has made possible to use with an external MIDI keyboard the sounds related with the J-Moog's AUDIO functionalities below described.

AUDIO functionality

It 's the part that required more effort, first of all from a point of view of the minimum preliminary study of digital signal processing (DSP) that were necessary to implement even simple features like those of this software.

For this purpose has been implemented three main classes.

WaveManager: is the component that handles the production of sound samples according to through an algorithm based on a specific choosen a specific wave form

(*shape*).

NoteThread: it represents each single autonomous process created by playing a note on a musical instrument. In the specific case it is activated by the *Model*, and once running use the sample provided by the WaveManager and send them to the audio line of the system sound card.

Effect(I): modify the already produced samples according to an algorithm that give the sound a specific feature (e.g. fade). Is used directly from NoteThread during its execution, by calling its static method. Effects use parameters that are managed by a specific handler (*EffectParametersManager*).

Thus, these elements contribute to provide the following functionalities:

- **Generation of the waveform** A first problem to be solved concerned the mode to generate the different waveforms (sinusoid, square, etc. Etc.). The solution adopted has been to implement a WaveImpl object that taken as input a value of the possible options (Enum *WaveType*), generates the parameters needed to model a specific waveform (see source code *Wave.java* and produced javadoc). The object thus obtained is passed with a *strategy* pattern to the *WaveManager* that generates according to a specific frequency (received when by a thread that is triggered when a key is pressed - see next item) the needed samples to be sent to the audio line according to the *waveform*.

- **Polyphony**

  The solution adopted is to implement an array of threads. Each is initialized with an appropriate frequency related to the corresponding note on the keyboard. Their *start()* method calls methods (static) of *WaveManager* instance (to get sound samples) and sound effects (to process samples by an specific audio effect algorithm (eg. smooter)).

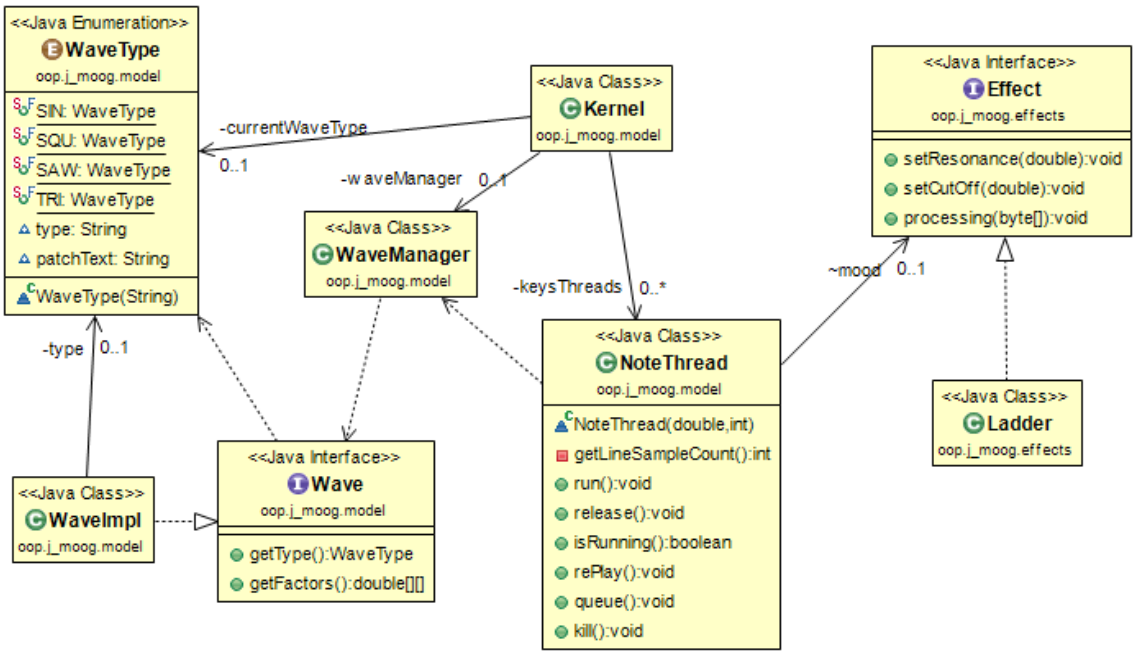

Figure 2.4: This picture shows the architecture and main components of the AUDIO functionalities

2.2.2 Controller

It acts as an interface in between the View and the above mentioned sound modules on the model (*Kernel*).

It has been implemented through a *singleton* pattern, in order to be conveniently accessed by multiple application parts (especially by various components of the view) and limit its instances to one at the same time.

In addition to the coordination functions between the View and the Model, it manages all the I/O operations[2], as well as to check about any external USB device connected to the system.

[2]using for these funcions a specific class for file extension filtering, in order to reduce wrong format file errors, and a Enum to classify all possible open/save options

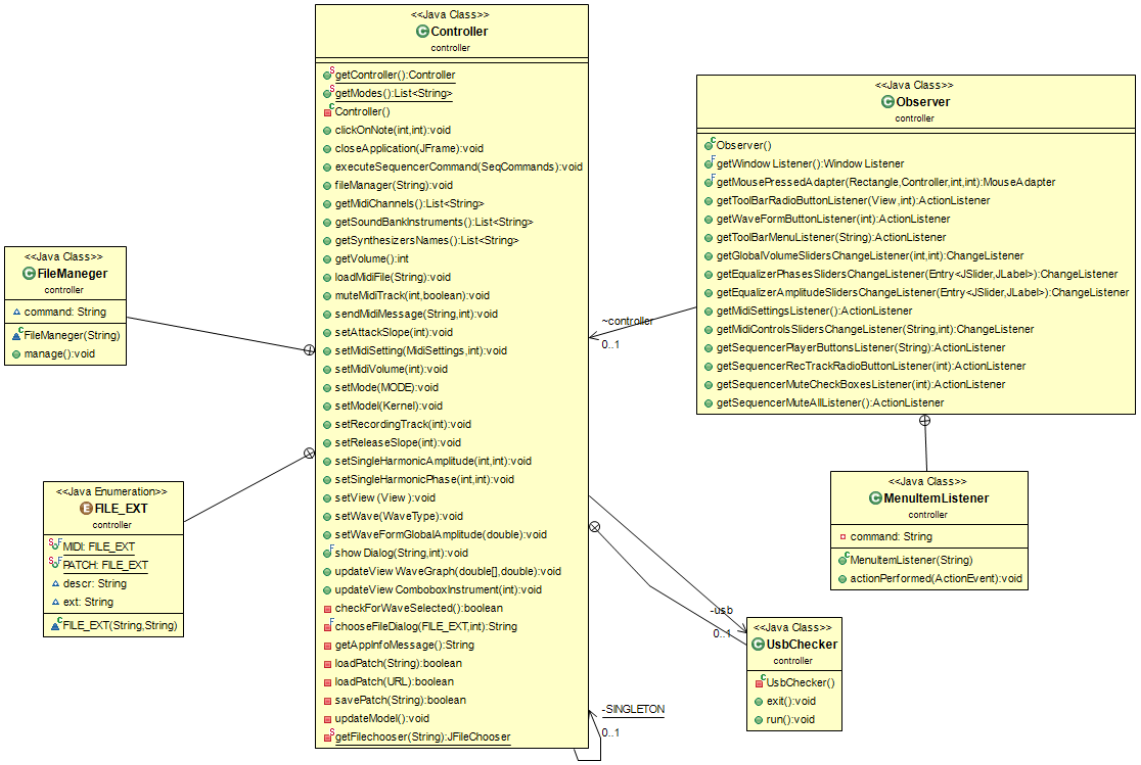

Figure 2.5: Controller and its main components.

2.2.3 View

It has been a quite laborious part to develop. It is build in order to be totally independent from the rest of the system. To achieve a better construction some *Enum* classes has been adopted to properly acquire certain parameters (eg.the number of buttons for specific functionality).

In addiction a specific class (*GUIFactory*) has been implemented in order to make easier (through calls to factory static methods) building certain parts of the *View* (such the equalizer, sequencer player and wave forms buttons) or to enable/disable a panel and its components.

Some among these methods return maps ¡Jslider, Jlabel¿ where each slider is coupled with its label, or in other cases buttons Groups (where in a collection of abstract buttons , is possible to select only one of them at a time). They have also allowed sothat, to *slim* the code in some parts.

The main Jframe of the View is based on a borderlayout so used:

- *North*
contains a x-boxlayout panel that holds the spectroscope display, the *MidiOption-Panel* and the *SequencerMidi* player.

- *West*
contains a y-boxlayout panel that holds the *WaveGraph* display, and the *Equalizer* (which is based on a y-boxlayout with nested x-boxlayout)

- *Center*
contains a gridlayout panel that holds the wave form buttons

- *East*
contains a y-boxlayout panel that holds a *PatchOptionPanel*

- *South*
contains a panel that holds the *Piano* keyboard (JPanel).

The piano keyboard code has been mostly taken from a demo of the official javadoc about the midi package, but has been improved so that at each *keypress* event, the keyboard is not completely redrawn every time, but only the concerned key and its "neighbors" . It need to be improved according to the MVC paradigm adopted as discussed better in chapter 3.

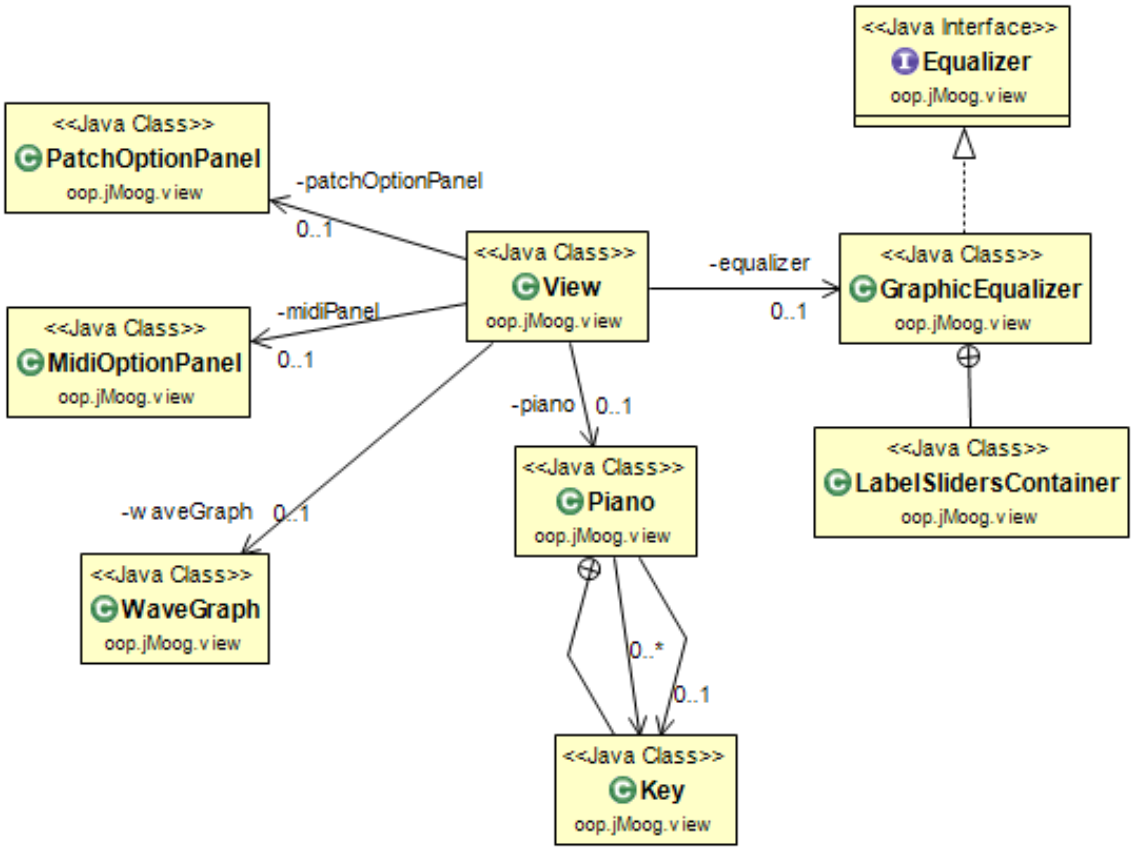

Figure 2.6: the View and its main components.

Chapter 3

Develop

3.1 Testing

Much of the application testing has been done manually, as the View has required a laborious testing phase to ensure that the various parts interacted with the controller (and in minimum part between them) in the proper way, trying at the same time to encapsulate each part of the system.

A part of testing has involved the generation of the waveforms, with test classes that bail on consoles the generated coefficients, in order to verify that under the strategy pattern every time passed to the *WaveManager*, this would generate the coefficients of the equations of the individual harmonics correctly.

3.2 Work Flow & Method

First of all, proper attention, in the context of the still few skills acquired, has been payed so that the various parts of the system were clearly distinct between them, and in particular, that the View (although a bit complex in some parts) was totally decoupled from the rest of the system.

Besides, it has been useful to prepare this report, parallel to the develop of some parts of the project.

DVCS systems as mercurial, has been used during the whole development, both for managing backups along the process, and also to try side develop tests on implementing some features, although it has not (unfortunately) been possible to work in

team within this project.

In addiction, the final code has been tested with suggested static source code analysis'tools (*findbugs* , *checkstyle*, and *PMD*).

3.3 Development Notes

First, wherever possible it is always privileged the use of *Collections*. With regard to the AUDIO functionality of the project, and taking into account the characteristics of the methods that allow access to the system audio line and its buffer features, after several tests it has for convenience opted for the use of arrays in buffer management (see class *NoteThread*) .

For I / O to disk as mentioned above, a filter class in the file management regarding patchs and midi file have been implemented in order to reduce wrong file's format errors.[1]

In this section, It is worth spending a few words about the logic adopted as regards as some of more interesting part of the system.

- *Key pressed/released Thread*: it has been the more challenging part of the whole system. Inside the thread's *run()* method are managed the audio buffer to send to the audio line, and a couple of boolean field used to manage in a thread-safe mode either the key release and an immediate re-play of the same key.[2]
- *Sequencer sliders*: here raised the need to combine on one hand a thread that would had to update the bar, on the other hand the displacement of the slider as consequence of an user action. The solution adopted is that the *Controller* starts a proper thread (nested Controller.*ProgressBarUpdater* class) after receiving a call from the *MidiSynth* that the file has properly run. The user action is attached to the mouse event.[3]
- *Equalizer sliders*: when a slider's change occurs, proper *Controller* > *WaveManager* methods are called. After the new wave shape computation is completed a call back method (*WaveManager* > *Controller* > *View* > *Graph*) update the graph.

[1] In addiction, not having found a way to show files without their extension in the open/save dialog window, specific method has been implemented to prevent undesired string concatenations in the file path.

[2] As it wasn't possible to call run() metod twice on the same thread [7]

[3] Notably a Mouse Adapter listener has been used, as just the release event had to be catched.

- *Wave Manager*: may be useful here to add few words about something that has been difficult to explain (and render) properly through the produced javadoc. The Wave Manager models the audio wave form selected, according to parameters provided by a WatyType object passed as argument with a strategy pattern.[4]

In addiction to all above, has been done an effort in order to make the application as user friendly as possible, also trying to reduce user interactions in terms of manual inputs insertion.

[4] For example a square ware equation is given by:
$sin(2\pi ft) + \frac{1}{3}sin(6\pi ft) + \frac{1}{5}sin(10\pi ft) \cdots \frac{1}{9}sin(18\pi ft)$.
In the *WaveImpl* class using three parameters i = 1,k = 2, e = 1 (and 8 default control sliders) the above equation has been obtained as: $\sum_{i=1(step=k)}^{8} \frac{1}{i^e}(2i\pi ft)$.
Combining this three parameters assuming $i = 1$ and $k, e \in [0, 1, 2]$ has been possible to obtain each of the four wave forms used.

Chapter 4

Final Remarks

4.1 Self-assessment and future developments

In conclusion, it remains primarily the satisfaction of having substantially implemented the original idea: namely, on one hand a midi file manager, on the other a digital synthesizer with all major components: a *Voltage Control Oscillator* (VCO), a *Voltage Control Frequency* (VCF) and an *Envelop Generator* (EG) starting from scratch.

However, in reviewing the work done, there is (*much*) room for improvement. Especially in terms of re-factoring the code in order to pursue a higher level of abstraction and increase code re-use.

In this sense further in-depth analysis will be made [1] in order to increased modularity of some features. According to this, it will deserve further studies the possibility of combining audio effects between them in difference sequences and also the of more advanced effects based (having higher computational cost), as well as the implementation of a spectrum analyzer via the Fast Fourier Transformation (FFT), which at the time of preparation of this document where not compatible with the total hours scheduled, already slightly exceeded.

Certainly, starting from the mistakes that will be covered during the discussion of this project, remains the intention of continuing its development, starting from

[1] E.g. try to divide the controller into two parts - one for audio and one for MIDI, or to slim somehow the Kernel.

Another topic in this sense will be the improvement of the Piano keyboard where further development will cover the mouse listener spin-off from the Rectangle class, which was not possible to add to a listener from the outside according to the M-V-C architecture.

exploring new features provided by Java 8, to explore then possibilities to realize a simple application which allows, for example, to use a mobile phone as a synthesizer connected directly to an external USB MIDI keyboard.

4.2 Difficulties Encountered and Suggestions for Teachers

The difficulties encountered can mainly grouped into two types:
- inherent lack of documentation
- arising from the characteristics of the chosen project

In the first group are mainly those related to the official documentation or other found from various sources, and notably too general and devoid of concrete examples that concern the majority of the sample applications.
As regards as this project, for example, the initial goal related to the implementation of real-time recording by a sequencer with capabilities to save the work on a midi file was in fact unfeasible within the available number of hours for the lack of sufficient documentation. In particular was low availability of examples (also on line) that would illustrate concretely how to combine the timing management.

In the second group can be identified difficulties
• derived from the need to use arrays with regards to certain audio features.
• to make algorithms work
• to use audio effects in real time and let them be used in thread-safe way by multiple threads simultaneously
• arising from the implementation of the interface, where the peculiarities of some features has led to inevitable difficulties
• in managing a USB device, testing all possible use cases: for example, an accidental disconnection resulting in an *Exception*.

Ed infine qualche riflessione ...

La prima: è stata una faticaccia...

Per il personale docente il presente progetto, con tutti i suoi errori e limiti, può ritenersi un obiettivo didattico raggiunto, tenendo conto di come sia stato realizzato da uno studente che prima di iniziare la frequentazione del corso non aveva mai adottato il linguaggio in questione, e tanto meno avvicinato la programmazione ad oggetti.

Quanto ad un feedback come studente, la principale considerazione riguarda le scadenze imposte da un corso di durata trimestrale, con un programma a dir poco vasto, nell'ambito del quale vengono trattati argomenti complessi (Reflections, Lambda, DVCS, ecc. ecc. ma l'elenco potrebbe continuare...) che non si ha il tempo di assimilare da una settimana all'altra. Questo fa si pertanto (ma è una considerazione necessariamente limitata a chi scrive) che l'impegno richiesto è stato tale da ritenere che, a fronte dell'impegno profuso dal personale docente in ogni contesto (lezioni, laboratorio, forum, materiale didattico, disponibilità), l'esame OOP sottostante il presente progetto (volendo usare un eufemismo) ben vale i 12 crediti relativi...

Bibliography

[1] Java Sound API
http://docs.oracle.com/javase/tutorial/sound/index.html

[2] Java Sound MIDI API
https://docs.oracle.com/javase/tutorial/sound/overview-MIDI.html

[3] Craig A. Lindley, *Digital Audio with Java*

[4] D. Flanagan, *Java Examples in a Nutshell*, 3rd Edition
http://www.onjava.com/excerpt/jenut3_ch17/

[5] P. Daly, *MSc in Acoustics and Music Technology*
http://www.acoustics.ed.ac.uk/wp-content/uploads/AMT_MSc_FinalProjects/2012__Daly__AMT_MSc_FinalProject_MoogVCF.pdf

[6] Javadoc, *Specifying Special Event Listeners*, Chapter 11: Playing, Recording and Editing MIDI Sequences
https://docs.oracle.com/javase/8/docs/technotes/guides/sound/programmer_guide/chapter11.html

[7] Javadoc, *Javadoc coding standards*
http://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html

[8] S. Colebourne, *Javadoc coding standards*
http://blog.joda.org/2012/11/javadoc-coding-standards.html

[9] R. L. DuBois and W. Thoben, *Sound*
https://processing.org/tutorials/sound/

[10] J. Burg PI, J. Romney, *Creating Sound Waves* , National Science Foundation CCLI Grant
http://csweb.cs.wfu.edu/~burg/CCLI/Tutorials/C/Chapter2/Creating_Sound_Waves.pdf

[11] P. Seifried, *Realtime audio processing, part 4: Comb filters, Flangers and Chorus effects – a bit of theory*
`http://philippseifried.com/blog/2011/11/01\\/`
`dynamic-audio-4-comb-filters-flangers-and-chorus-effects/`

[12] J.O. Smith, *Elimination of Limit Cycles and Overflow Oscillations in Time-Varying Lattive and Ladder Digital Filters*, IEEE Conference on Circuits and Systems, no. STAN-M-35
`https://ccrma.stanford.edu/files/papers/stanm35.pdf`

Appendix A

Quick User Guide

From the main toolbar → *mode* menu is possible to select the mode of use of the software.

- **MIDI mode**
  Selecting this mode will activate the MIDI player buttons. It is possible to launch the included demo in the software (*load demo midi* option) or open a midi file from disk. The radio buttons are used to select the recording track (functionalities Record/save under construction not completed) While the checkboxes allow to selectively listen (mute/on) to each track of the played Midi file.

- **ANALOG mode**
  By selecting this mode, you have access to the *synth* functionalities of the software.

  This option automatically activates the buttons to select the desired waveform and the equalizer to act on the components of the additive synthesis and the sliders to control the audio effects.

  All menu options are implemented.

  It is possible to load the patch demo (*load demo patch* option) and save/open sounds obtained onto disc (*patch*).