Facultade de Informática
UNIVERSIDADE DA CORUÑA

TRABALLO FIN DE GRAO
GRAO EN ENXEÑARÍA INFORMÁTICA
MENCIÓN EN COMPUTACIÓN

MIDI-based music score editor

Estudante: Daniel Pena Docampo
Dirección: Óscar Fresnedo Arias
José Pablo González Coma

A Coruña, febreiro de 2020.

To my parents

Acknowledgements

First of all, I would like to thank to my project directors Óscar Fresnedo Arias and José Pablo González Coma for all their attention, dedication and help during these last few months. Without their help I would not be able to carry out the project successfully.

I would like to thank my family. To my parents, who have always encouraged me to be to work hard, to be a good person and helped me to overcome any problem. To my brother, for all his support and love, always believing in me.

I would also like to thank to the great friends met during these four years. Specially to those who supported me in the good and not so good moments.

Abstract

Music has always played a fundamental role in society. It is a basic human function and, as a consequence people have always found music significant whether it is for enjoyment in listening, its emotional response, performing or creating. Music is represented using music notation. Music notation is the transcription of music into music scores. Music scores have been a way of teaching and sharing music with others. In addition, it was a key contributor to evolving music language and allowing the works of composers to be preserved over time.

The continuous advancements in technology have influenced both the creation and recording of music. One of the milestones for music creation and recording was Musical Instrument Digital Interface (MIDI). MIDI is a protocol that provides a standardized and efficient means of conveying musical performance information as electronic data. Then, it is possible to obtain musical notation information by just processing the electronic data inside a MIDI file.

The objective of this project is to develop and implement a MIDI based music score creator. Users will be able to load MIDI files, process them and generate the correspondent music score. To achieve this, a combination of musical knowledge and information obtained by the MIDI protocol will be used to create the accurate scores.

Resumo

A música sempre xogou un papel fundamental na sociedade. A música é unha función básica nos humanos e polo tanto, as música sempre foi importante para as persoas, xa fora para disfrutar escoitándoa, pola resposta emocional, para interpretala ou para creala. A música é representada pola notación musical. A notación musical consiste na transcipción de música en partituras. As partituras músicales sempre foron unha forma de enseñar e compartir música entre as persoas. Ademais, as partituras foron un contribuinte clave na evolución da linguaxe musical e permitiron preservar as obras creadas por diferentes compositores a través do tempo.

Os avances continuos en tecnoloxía influenciaron tanto a creación como a grabación da música. Un dos hitos máis importantes para a creación e grabación de musica foi a creación de MIDI. O protocolo MIDI proporciona un medio estandarizado e eficiente para transmitir información musical como datos electrónicos. Polo tanto, é posible obtener información sobre notación musical simplemente procesando os datos electrónicos dentro dun archivo MIDI.

O obxectivo deste proxecto e desarrollar e implementar un creador de partituras musicales baseado en MIDI. Os usuarios poderán cargar archivos MIDI, procesales e xerar a partitura correspondiente. Para logralo, utilizarase unha combination de conocimientos musicales e de información obtida polo protocolo MIDI para crear partituras precisas.

Keywords:

- MIDI.
- Music.
- Score.
- Note.
- Pulse.
- Time signature
- Key signature
- Beat.

Palabras chave:

- MIDI.
- Música.
- Partitura.
- Nota.
- Pulso.
- Compás.
- Armadura.
- Ritmo.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

Music has always played a fundamental role in our society. It has been present from the beginning of humanity to the modern era and will surely continue to be important in the future.

There are various reasons why music has such a big impact in people. The most remarkable is that music is a basic human function, as the interaction with sound is inevitable, either to make it or to take pleasure in it. As a consequence, people have always found music significant, whether it is for enjoyment in listening, its emotional response, performing, or creating [1].

Its importance can also be reflected in the number of famous quotes intellectuals of different eras have dedicated to it. Some examples are ”Without music, life would be a mistake” from Friedrich Nietzsche [2] or ”music is a moral law. It gives soul to the universe, wings to the mind, flight to the imagination, and charm and gaiety to life and to everything” from Plato [3].

Moreover, scientists have demonstrated through many studies that children with music studies tend to develop better emotional, physical and academic skills with respect to the ones that do not study this discipline [4].

The music industry has always been one of the biggest business in the global market, making an average of 17 million dollars of revenue per year. Its impact can be noticed in the number of famous artist and superstars with thousands and millions of fans that are willing to travel around the world just to see them perform live.

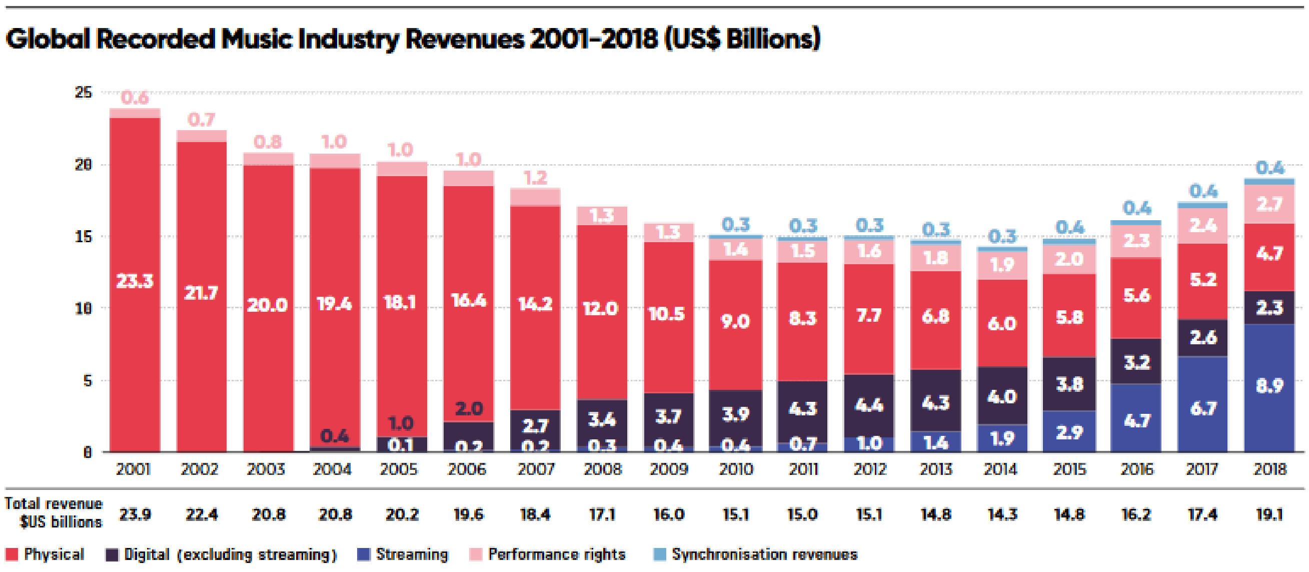


Figure 1.1: Music Industry Revenue separated in categories [5]

.

A change in the way that music has been recorded for the past few years can be appreciated In Figure 1.1. While in the first lustrum of the twentieth century all the music was recorded to be played in physical platforms, from then on, taking advantage of the continuous advances in technology, the digital recording has experienced a exponential growth over the years surpassing the physical recording and becoming the most important in the industry.

Such advancements in technology have influenced both the recording and the creation of music. One of the milestones was the creation of the MIDI. MIDI is a protocol that provides a standardized and efficient mean of conveying musical performance information as electronic data [6]. This protocol enabled a lot of new techniques to add, create, store, treat and modify music as electronic data that revolutionized the whole industry.

The advancements that MIDI brings to the analysis of the data in music added to the importance of music in the society are the main reasons why this project has been chosen.

1.2 Motivation

The aim of this project is to develop a software that receives a MIDI file recorded with a piano, analyses it and prints the score sheet corresponding to that file.

This software is meant to be really useful for many people in the music industry, and education. Some examples in which this software makes work easier are:

Composition. The process of composing music is quite large and complex. First, a composer has an idea of a melody or a song. Usually, just by playing the piano without a specific idea in mind some good melody or base for a song can be discovered. When this happens, sometimes is really difficult to remember what exactly was played before and the process of recovering and remembering the exact notes can become really frustrating. Afterwards, if they successfully remember this base they usually record it. Once this happens, then they enter the process of writing the score. This is a time consuming task, as they have to write down all the information in a score by hand.

On the other hand, if they use the software that is going to be provided in this project, only by recording everything that they play, they would not have to worry about anything else, as all the calculation and transcription would be done automatically by the program. Thus, just by using this software their workload would diminish considerably.

In addition, many current artists of the music scene do not have music studies. Therefore, when these artists find a melody they like, they need the help of people with higher musical knowledge to try to put it within a context needed to create a song such as key signature, tone etc. With this software, all of these operations needed when creating the representation in the score are done automatically. While the representations calculated by the software in these cases might need additional refinement due to MIDI limitations, they are going to be a really good approximation of the structure of the melody.

Education. At all levels of education, teachers usually ask their students to do some exercises at home such as specific music scales, intervals, notes, rhythms, etc. With this software, they would only need to play it once, having the exercises written automatically and making it easy to share them with their students. This can be expanded into different situations, as every time the teacher wanted the student to practice a specific musical piece, they would only need to play it and share the score obtained with them.

Furthermore, this software can be used as a study tool as it makes easy to detect errors when playing a specific song. This can be checked comparing the output generated by the program with the original score of the song. With these two scores, the errors made can be detected and can help to understand what is wrong and what is needed to do to solve it.

1.3 Objectives

As stated before the main objective of the project is to develop a software that receives MIDI files and creates the correspondent music scores. The software has to fulfil the following requisites:

- The amount of information asked to the user has to be minimal. The idea of the software is that it has to be able to represent the score with the minimum available information and, therefore, being as useful as possible to a wide variety of users.

- Represent MIDI files recorded with piano. This requirement is motivated because the piano is the most complex instrument and thus, the instrument that is going to create the most complex and difficult combinations to analyse. The piano is the only instrument that has two music staffs representing the notes played by the right hand and the left hand. This allows artists to play several notes at the same time and make lots of different combinations, resulting in thousands of different scenarios. As a result, if the software is able to analyse and process these music pieces correctly, it will be very easy to extend its usability to easier instruments like clarinet or saxophone, that only need one staff and can only play one note at a time.

- Create a simple and use-friendly UI which makes easy the loading and processing of MIDI files as well as displaying the score created.

Chapter 2

Fundamentals and main concepts

In this chapter the main concepts needed to understand the project are going to be explained. First, an explanation of some basic musical language concepts are going to be exposed. Subsequently, the MIDI protocol is analyzed, the features provided and how they can be used to obtain the desired result is going to be showed.

2.1 Musical Language

2.1.1 Staff

The staff is a set of five horizontal lines and four spaces used for the representation of notes of different pitches [7]. The staff is the basic structure of a music score as all of them are composed by a set of staffs containing notes that recreate the music.

A specific type of staff is the grand staff. The grand staff is a combination of two staffs joined by a brace, with the intention of being played at once by a single performer. This staff is the one used for piano scores, where the upper one is intended for the right hand and the lower for the left hand. This staff is the one that is going to be used in this project.

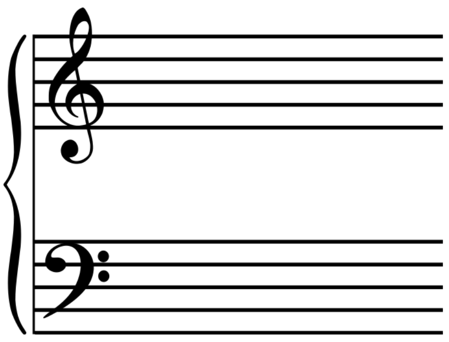


Figure 2.1: Grand staff representation

However, staffs by themselves have no precise meaning without the addition of some symbols at the left-hand side that are going to add information about the notes in that staff and the structure of the music score. These elements are going to be the Clef, Time Signature and Key Signature. Every one of these elements will be explained in the following sections.

2.1.2 Note

A note is as symbol used to represent a single musical sound in the music score [7]. Notes are the basic elements in a music score as the different combinations between them creates the different songs and musical pieces.

In order to explain the different note attributes, the term **beat** has to be explained. The **Beat or Pulse** is the basic unit of time, the regular rhythmic pattern of the music.

The different attributes that add information about every specific note in a score are:

- **Note figure.** Symbol used to indicate the duration of a note. There are several figures used in score, each one representing a specific duration. This duration is specified in terms of the number of pulses or beats that notes last.

  Considering that the quarter note figure is set to have a duration of 1 beat, the more common musical figures representation, their figure note name, the duration in beats and its representation in musical notation are defined in Table 2.1.

Figure	Note Figure	Duration in beats	Musical notation representation
𝅝	Whole Note	4	1
𝅗𝅥	Half Note	2	2
♩	Quarter Note	1	4
♪	Eighth Note	1/2	8
𝅘𝅥𝅯	Sixteenth Note	1/4	16
𝅘𝅥𝅰	Thirty-Second Note	1/8	32
𝅘𝅥𝅱	Sixty-Fourth Note	1/16	64

Table 2.1: Note Figures

  Besides, each one of these figures may have a dot added next to them. This dot means that the duration of that figure is the duration of the figure plus half of the duration of such figure. Some examples are:

  – ♩. This figure is called dotted-quarter and has a duration of 1 + 0.5 = 1.5 beats.

  – 𝅗𝅥. This figure is called dotted-half and has a duration of 2 + 1 = 3 beats.

This applies to every figure showed above.

- **Name or note pitch.** Each note has a name corresponding to a specific sound. Thus, the same sound will always be called by the same name. There are several alphabets for this purpose. The one that is going to be used is the English naming convention.

  Pitch is a perceptual property of sound that allows their ordering on a frequency-related scale. More commonly, it is the quality that makes judging sounds as higher and lower possible [8].

  The scale of music used is the so called chromatic scale or twelve-tone scale, which is a set of 12 musical sounds ordered by pitch. The chromatic scale is composed by 12 sounds that are a semitone above or below their adjacent pitches. This scale is composed by seven natural notes and five altered notes.

  Tone and semitone are measurements that indicates the difference between different note pitches. Semiton e is half the value of a tone.

  The natural notes of the scale are represented with A, B, C, D, E, F, G. The interval between these notes, excepting B-C and E-F is of a tone. The interval between the B-C and E-F is of a semitone. The altered notes are represented by the name of the natural notes plus the Flat (♭) symbol or the Sharp (♯) symbol.

  - The ♯ symbol increases a semitone to the note that is represented with.
  - The ♭ symbol decreases a semitone to the note that is represented with.

  Therefore, the altered notes are going to be: A♯/B♭, C♯/D♭, D♯/E♭, F♯/G♭. The result is five pairs of altered notes, each having two different representations for the same sound. Also, it can be appreciated that there is no altered note between B-C and E-F. The reason why they do not appear is because B/E ♯ are the same notes as C/F and C/F ♭ are the same notes as B/E.

  The chromatic scale is:

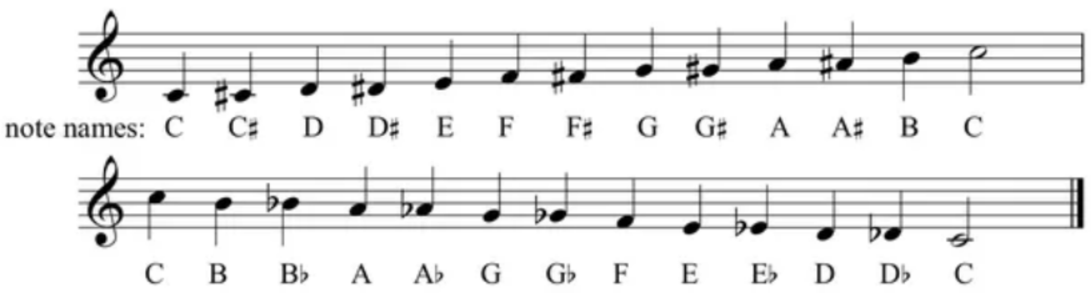

note names: C C♯ D D♯ E F F♯ G G♯ A A♯ B C
C B B♭ A A♭ G G♭ F E E♭ D D♭ C

Figure 2.2: Chromatic Scale [9]

- **Octave.** The octave is an interval between one musical pitch and another with its frequency doubled, known as interval of 8 tones [7]. As can be seen in figure 2.2, notes separated by an octave have the same letter name and belong to the same pitch class. A pitch class is a set of all pitches that are a whole number of octaves apart [10].

  The octave is a fundamental piece of information used to determine a specific note. Without its information it would be impossible to establish the exact position of a note in the score. Only when both the information of the name and the octave are known, then is possible to locate the position of the note.

2.1.3 Clef

The clef is a musical symbol that specifies the exact position of a specific note and, consequently, the positions of all the notes within the score.

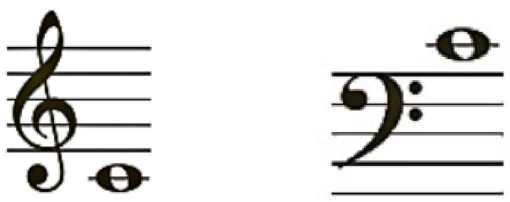
There are three different clefs used in modern music notation:

Clef	Name	Note	Octave	Note Location
𝄞	Treble Clef / G-Clef	G	4	Second line of the staff
𝄢	Bass Clef / F-Clef	F	3	Four line of the staff
𝄡	Alto Clef / C-Clef in third	C	4	Third line of the staff
	Tenor Clef / C-Clef in fourth	C	4	Fourth line of the staff

Table 2.2: Music Clefs

In table 2.2 can be observed that the C-clef symbol can be two different clefs depending on how it is positioned in the staff. In any case, these two clefs are rarely used in today's music notation, being relegated to very specific scenarios.

Moreover, it can be noted that the F-Clef scale is one octave below G-Clef scale. This is the reason why in most piano scores the right hand notes are written using G-Clef and the left-hand notes in F-clef. In figure 2.3 the note C in octave 4 is going to be represented in terms of both of these clefs to notice the differences between them.



(a) C4 with C-Clef (b) C4 with F-Clef

Figure 2.3: C4 with C and F Clefs [11]

2.1.4 Time Signature

Before explaining what time signature is, an explanation of two musical concepts have to be done.

- **Tempo.** The tempo is the speed at which a piece of music is played. It can be specified by digits representing the number of beats per minute or a word that represents a more flexible tempo. These words imply a range of beats per minute in which the song is supposed to be played [7].
- **Bar.** A bar is a segment of time corresponding to a specific number of beats. Bars are the segments of the score. They are separated between them using bar lines. There are different bar lines, the most common are:

Bar	Name	Use
𝄀	Single Bar Line	Indicates end of a measure
𝄆 𝄇	Repeat Line	Indicates to repeat everything inside those brackets once
𝄂	End Bar Line	Indicates the end of the score

Table 2.3: Music Bars Lines

Time signature is a notational convention used in musical notation to specify how many beats or pulses are contained within a bar and which note figure is equivalent to a beat [12]. Thus, it is the element that defines the structure and the rhythm of the song.

It is composed by two numbers, the numerator and denominator. The numerator defines the number of beats per measure or bar and the denominator defines the note figure which corresponds to a beat in the score. There are three types of time signatures: simple, compound and complex.

The simple time signatures are those where the beat can be divided into two equal parts. They are the most common type. The most usual are 2/4, 3/4 and 4/4. These time signatures define a measure with a length of 2, 3 and 4 beats respectively as well as setting the duration of a quarter note (♩) as the beat length. In Figure 2.4 an example of a 4/4 time signature is displayed.

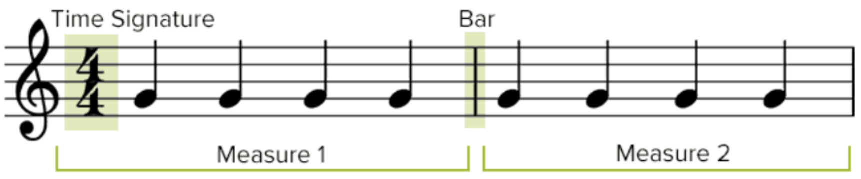


Figure 2.4: Representation of a 4/4 time signature bars [12]

The compound signatures are those where the beat can be divided into three equal parts. In this type of time signatures the numbers express a slightly different information. The most common are 3/8, 6/8, 9/8, 12/8. The numerator defines the number of divisions there are in a measure and the denominator the note figure that represents the division, which is an eighth-note (♪).

As in the compound signatures the beat is divided into three divisions, it will be calculated by joining those divisions. Therefore, the beat defined by these signatures will be a dotted-quarter (♩.). Hence, the previous signatures define a bar with a length of 1, 2, 3 and 4 beats, being the beat a dotted quarter.

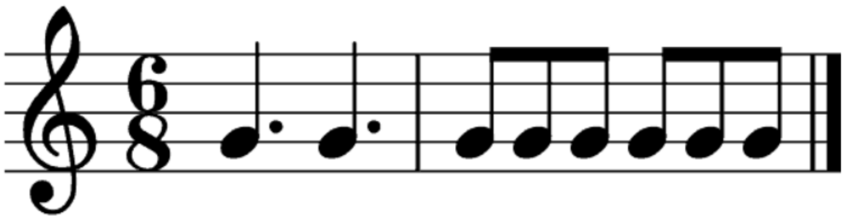


Figure 2.5: Representation of a 6/8 time signature bars [12]

The complex time signatures are those measures that contain both simple and compound beats. These signatures are the less common so no deeper analysis is going to be made about them.

2.1.5 Key Signature

A key signature is a set of sharps or flats placed together at the beginning of the staff after the clef sign. These elements are going to affect every note on the line or space they are located throughout the entire composition [13]. It represents the key in which the piece of music is written.

A key indicates the base scale of a music composition and, subsequently, the base chords. Every key is defined by a key signature which determines them. Key may be in two modes, major and minor mode. The differences between them are complex and are not needed for the development of this project so no deeper analysis will be made.

There are several key signatures and they may only contain sharp or flat elements but not both of them. Besides, these elements have to appear in a specific order. Accordingly, it is possible to know the altered notes of a score just by knowing the number of sharps or flats in the key signature.

If the key signature is composed with sharp elements ♯ the order of appearance is F♯, C♯, G♯, D♯, A♯, E♯, B♯. From then on it would start again from F creating the so called double sharp (𝄪) that increases note by a whole tone.

If the key signature is composed with flat elements ♭ the order of appearance is B♭, E♭, A♭, D♭, G♭, C♭, F♭. From then on it would start again form B creating the so called double flat (𝄫) that decreases a note by a whole tone.

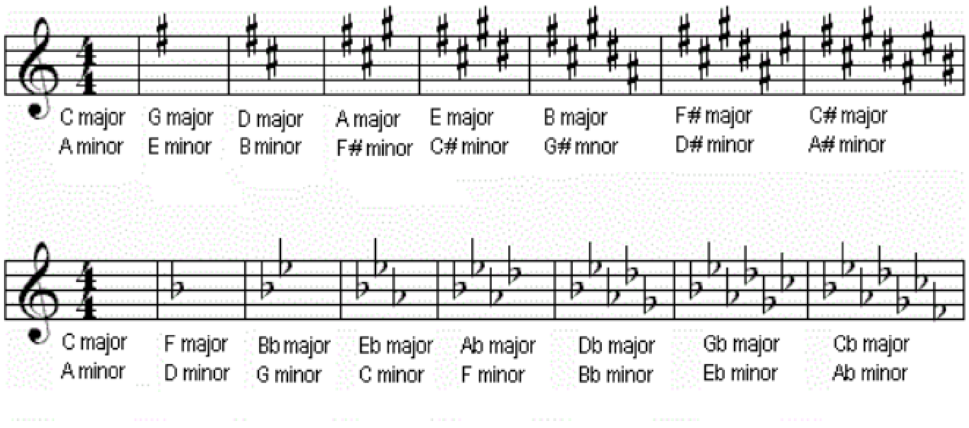


Figure 2.6: Representation of key signatures and their respective keys [13]

In figure 2.7 an example of a score with a key signature can be seen.

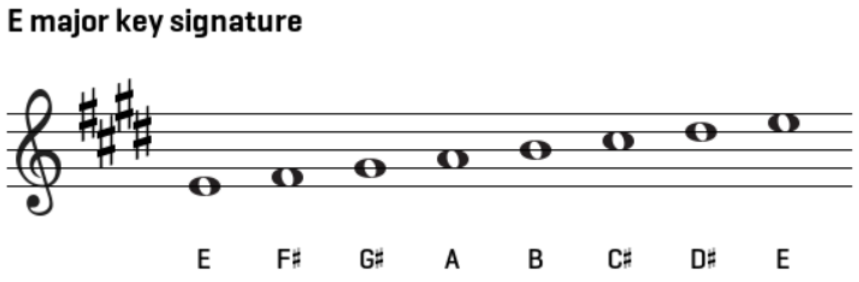


Figure 2.7: Representation of the E Major scale [14]

It can be realized that while the notes are written as natural notes, the ones matching the notes represented by the sharp elements are altered by them. If one of these notes was wanted to be natural, then the symbol ♮ has to be written before the desired note. Then, the desired note and all of notes in the same pitch class within a bar will be natural, with the exception of another sharp or flat symbol appearing afterwards.

2.2 MIDI

2.2.1 Introduction

MIDI is a protocol that provides a standardized and efficient means of conveying musical performance information as electronic data [6]. It was originated as a real-time protocol to enable communication between separate hardware devices. Specifically, the main objective was to interpret the frequency and the sound wave information made from playing an electronic keyboard [15].

MIDI files do not contain the sampled audio, they contain the instructions needed by a synthesizer to generate the sound in form of MIDI messages. These messages tell the synthesizer which sounds to use, which note to play, and how to play each note. This introduces various advantages when compared with the sampled audio saved in disks or CD-ROMs. The first advantage is storage space. While high quality stereo sampled audio requires about 10 Megabytes of data per minute of sound, a MIDI sequence might consume less than 10 Kilobytes of data per minute of sound. Besides, it includes capabilities to easily edit the music inside a file such as the tempo, the key or the pitch of the sounds independently [6].

The MIDI hardware interface on a MIDI instrument will generally include thee connectors, which are IN, OUT and THRU. A MIDI data stream, which is an unidirectional asynchronous bit stream at 31.25 Kilobits/second with 10 bits transmitted per byte, may be originated by a MIDI controller o a MIDI sequence. A MIDI controller is a device which is played as an instrument and translates its performance into a MIDI data stream in real time. A MIDI sequencer is a device which allows data sequences to be stored, edited, combined, and replayed. The data stream originated is transmitted via the OUT connector [6].

The data stream is then received by a sound generator or sound module, which will receive the data through the MIDI IN connector to perform the desired operations. The data received through the IN connector can be transmitted back out using the MIDI THRU connector. Therefore, many sound modules can be connected as a chain by connecting the THRU output of one device into the IN connector of the next one [6].

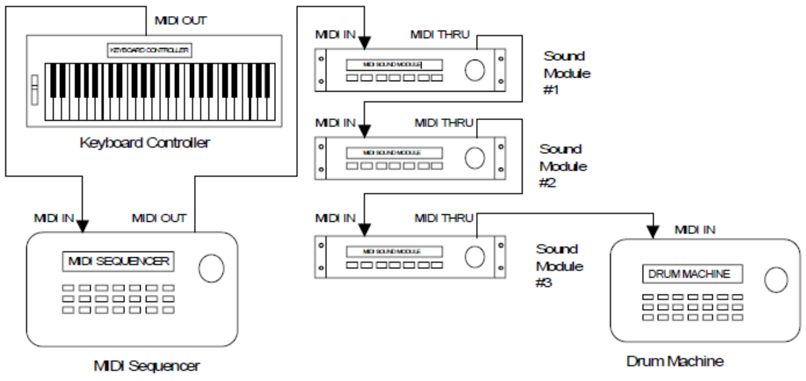


Figure 2.8: Representation of a MIDI system [6]

A composer might use a system like the one in Figure 2.8 to write a piece of music consisting of several parts where each on of them is written for a different instrument. Each part would be played one at a time and captured by the sequencer in different channels. Afterwards, each sound module would be set to receive a different channel. Finally, all sound modules would play these parts at the same time [6].

As explained before, MIDI as a hardware interface is concerned with real time processing where music consists of a string of events. These events happen one at a time and from moment to moment, creating a stream of events. On the other hand, a musical score or musical recording is an static object that contains all the events information statically stored, not as a stream of real time events. [15]. As a result, MIDI has to facilitate a two-way traffic between these static objects and the dynamic processes controlled by hardware, supporting two kinds of processes [15].

Firstly, as a recording device, a MIDI keyboard captures key strokes and converts them into digital data. Secondly, as a broadcast service, a MIDI file sends the data as one event at a time [15]. This traffic between both directions is regulated by a MIDI clock. The clock will time-stamp every musical event and then record the event in disk [15]

2.2.2 MIDI files

There are three different types of MIDI files for organizing multi-track music [15].

- **Format 0**. This format uses one track and is vertically one-dimensional conforming one time-line. This is the one used to represent monophonic music. When multiple voices are represented under this format, they are all saved in one track.

- **Format 1**. This format is intended to be used for a multi-voice music with different tracks that are melodically different. Thus, each voice will occupy one track. All of the tracks are vertically one-dimensional conforming one time line.

- **Format 2**. This format is intended for music with different tracks that are temporally different. It is horizontally one-dimensional.

Every MIDI file consists of a series of 8-bit byes and contains two sections or chunks, the header chunk(MThd) and the track chunk(MTrk). These chunks concept and syntax are outgrowths of the Interchange File Format (IFF) architecture first offered by Electronic Arts. Each chunk as a 4-character American Standard Code for Information Interchange (ASCII) type and 32-bit length.

The header chunks specify basic information about the entire MIDI file. Its syntax is:

<HeaderChunk> = <chunk type> <length> <format> <ntrks> <division>

The <Chunk type> and <length> are the four ASCII characters "MThd" and the 32-bit representation of the data that follows. The <format> represents the format of the file (0, 1, 2). The <ntrks> identifies the number of track files in the chunk and <division> specifies the meaning of the delta times. The <division> has two formats, one for metrical time and another for time-code base time. [6].

The track chunks are where song data is stored. They contain a stream of sequential events preceded by a delta time. Its syntax is:

<Track Chunk> = <chunk type> <length> <MTrk event> [6]

The <Chunk type> and <length> represent the same information as the tags in the header chunk. In this case, the <Chunk type> is going to be "MTrk". Then, the syntax of a <MTrk event> is:

<MTrk event> = <delta-time> <event> [6]

<delta-time> is stored as variable-length quality. It represents the time at which an event has happened. Delta time is measured in ticks as specified in the header chunk. The <event> is any MIDI channel message. It can be:

<event> = <MIDI event> | <sysex event> | <meta-event> [6]

These are the different type of MIDI events that can occur. They will be explained in the next section.

Format 0 files have a header chunk followed by one track chunk. Formats 1 and 2 have a head chunk followed by one or (usually) more track chunks [15].

In this project, format 0 files are going to be used. The reason is that in most cases a song is going to be recorder playing both hands at the same time and, therefore, saving all the notes played in the same track. In order to work with format 1 files, the right-hand and left-hand parts of the song would need to be played separately one after the other and saved in different tracks. While using format 1 files would ease the task of creating the score as the left-hand and right-hand notes would be separated, in practice every recorded song will have both parts played at the same time.

2.2.3 MIDI event types

MIDI events

These events contain musical performance information. The messages in these events are:

- **Note_on, Note_off, velocity.** These are the events that correspond to the activation and release of a particular note. MIDI note_on and note_off events are independent, considered as two separate events. When a key is pressed on a MIDI keyboard or controller, a note_on event is sent through the MIDI OUT port. The note_on event contains two data bytes that specify the key number pressed in the keyboard and the velocity, which is a number that represents how hard the key was pressed. This information is used then by the synthesizer to select which note to play as well as its amplitude. Once the key is released, the note off message will be send, containing also the key number and the velocity of the note, which is probably going to be 0 as it is normally ignored [6].

  These events are going to be critical in the development of the software as these events will provide the information about every note in the song and this is going to be the base to build all the functionality of the program.

- **Aftertouch**. Some MIDI keyboards have the ability to sense the amount of pressure which is being applied to the keys while they are being pressed. This information is used to control some aspects of the sound produced by a synthesizer [6]. This event is not going to be processed in this project as it does not give useful information to create a score and not every keyboard can send it.

- **Pitch Bend**. Message send when a keyboard instrument changes the position of the pitch bend wheel used to modify the pitch of sounds being played on a given channel [6]. It will not be processed in the development of the project as it does not include useful information.

- **Control Change**. Used to control a variety of functions in a synthesizer. One of these functions is to inform when the keyboard pedal has been activated and when it has been deactivated [6]. While this information is useful for the synthesizer it cannot be used in the calculations of the score as many times the pedal is just used to make the transition between notes smoother that might confuse the program when calculating the duration of the notes. One example may be when three sequential eight notes are wanted to have their sound mixed using the pedal. If the information of the pedal is taken into account the result will be that the three notes perform a chord with a duration of a dotted quarter note. On the other hand, if the information of note_on and note_off is taken into account, the three eighth notes are successfully detected.

- **Program Change**. This message is used to specify the type of instrument which should be used to record the music of a single channel. Once again, this event is not useful for the purpose of the project.

Meta events

These events represent more general features of the music that are not conceived in real time. They contain information about the track names, tempo indications, lyrics, copyright notices and so forth. Many of these events are optional, programs do not have to support every meta_event. The messages in these events are [6]

- **Sequence Number.** Event that occurs at the beginning of a track. I specifies the number of a sequence. This event is really useful for format 2 files to determine which sequence is the following. For file formats 0 and 1, which only contain one sequence, this number is contained in the first track.
- **Text Event.** Event used for description of the song. It can contain the name of the track, description of intended orchestration, lyrics etc.
- **Copyright Notice.** Contains a copyright notice as printable ASCII text.
- **Sequence/Track Name.** Contains the name of the sequence or track in file formats 0 and 1 (for track in format 1 only the name of the first one).
- **Instrument Name.** Contains a description of the instrumentation used in the track.
- **Lyric.** Contains the lyrics to be sung. Generally, each syllable will be a separate lyric event.
- **Cue Point.** Contains a description of something that happens on a film, video or stage at a specific point of a musical score.
- **End of Track.** Mandatory event. Indicates the exact point in which the track ends.
- **Set Tempo.** Event that indicates a tempo change in a specific point of a music score.
- **Time Signature.** Event that indicates the time signature of the score expressed as an hexadecimal number. After some calculations that number can be translated into the numerator and denominator of the key signature of the score. If no time signature is specified, then the default value will be 120 ticks per quarter note.

- **Key Signature.** Event that indicates the key signature of the score. Depending of the value of some digits of the hexadecimal number indicates the number of sharp or flats in the key signature of the score. If no key signature is specified, then the default value will be key without any accidental elements.

These meta events are not created automatically by recording a song. Instead, they have to be add manually by the user. As one of the objectives of the project was to create the score in a way asking the user the least amount of information, it will be tried to use only the events that are strictly necessary to create a good score.

System exclusive events

These events are used to convey a message that pertains to a particular hardware system. These events are not important for the purpose of the project so no further analysis will be made.

2.2.4 Conclusion

The analysis performed has clarified that, from a software perspective, MIDI was exclusively design for sound applications and digital communication between electronic instruments, not as a page-description language for musical scores. As a result, the information required to print music efficiently greatly exceeds what MIDI provides, both quantitatively and qualitatively [15].

It does not exist a simple way to compensate for information that is simply not there. This might be the reason why there are not many software that perform the operation of transforming a MIDI file into a music score as very complex operations are needed in order to obtain an acceptable result.

Chapter 3

State of the art

In this chapter a review of the available software similar to the one proposed in this project is performed. First, a general view of the features provided by the existing software tools will be done. Finally, an analysis fo the good and bad aspects of the software when creating the music score will be carried out.

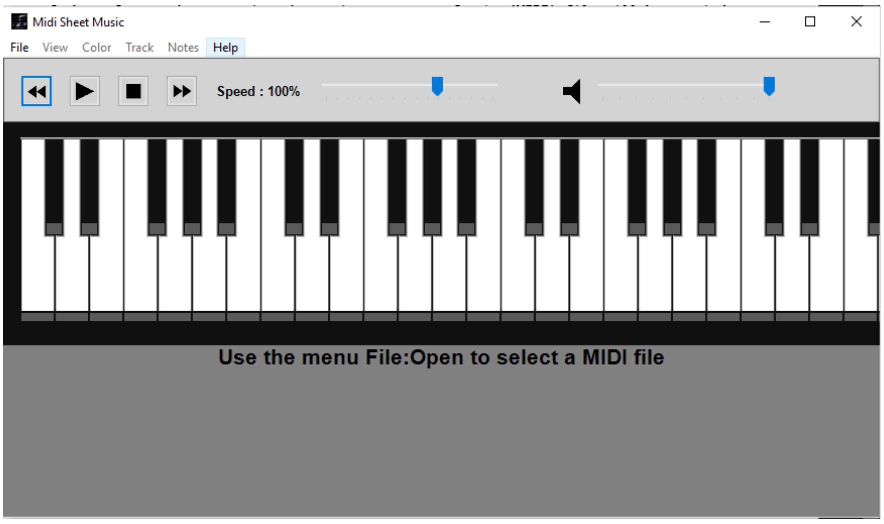
3.1 Midi sheet music

Midi sheet music is a free software that receives a midi file and displays the correspondent music score. The author of this software is Madhav Vaiyanathan.

Some of the features of this software are [16]:

- Display note name next to each note.
- Save the score in Portable Network Graphics (PNG) images or Portable Document Format (PDF) file.
- Display notes in different colors.
- Selection of which MIDI Tracks to display.
- Combining MIDI Tracks into two staffs (left hand and right hand).
- Adjusting key signature, time signature, and measure length.
- Transposing notes up or down.
- Playback of the music.
- Highlight piano keys and score notes as the music is played.
- Adjust the speed of the playback

The UI can be seen in Figure 3.1. It is a simple UI composed by a tool bar with buttons to play the song, a piano representation and the score created when a file is processed.



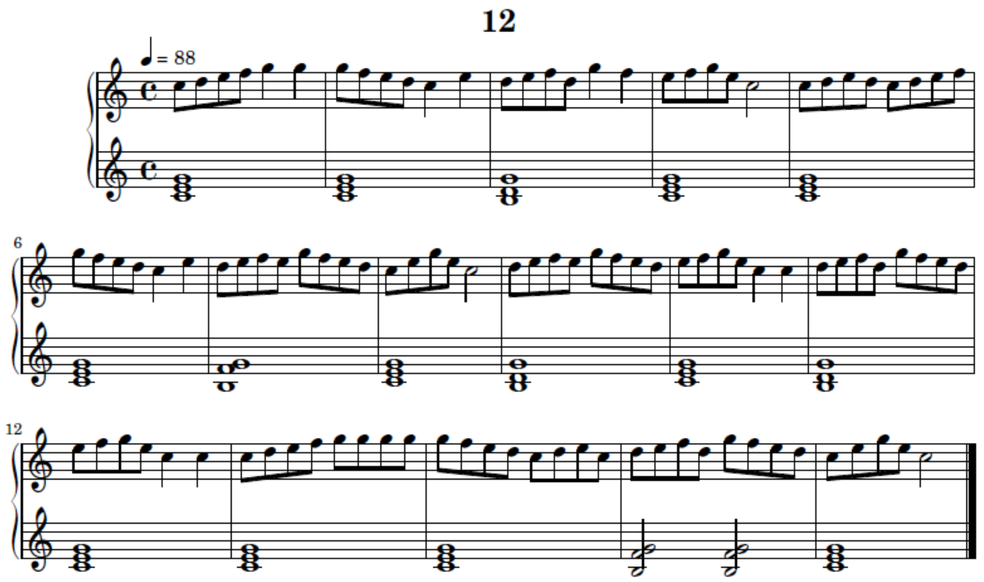
Use the menu File:Open to select a MIDI file

Figure 3.1: Midi sheet Music UI [16]

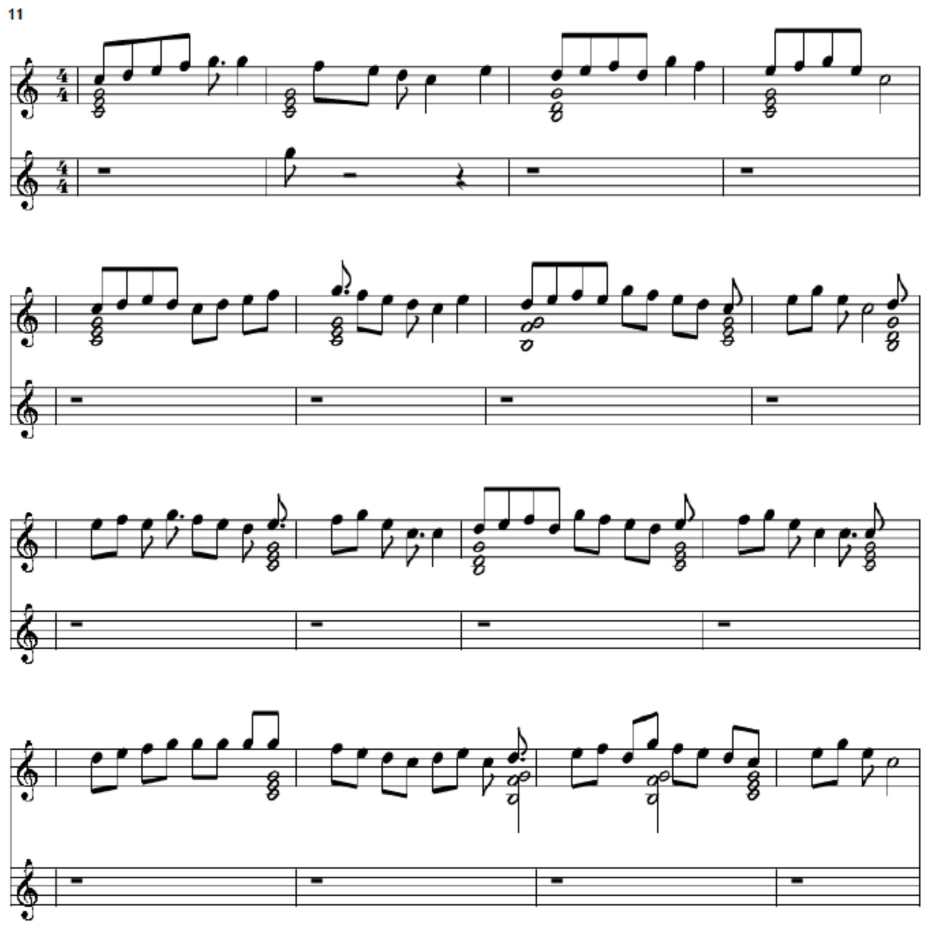
This program has a lots of visual features that makes it very attractive from that point of view. The problem comes when an analysis of the creation of the scores is made.

First of all, this software cannot calculate the time signature so it needs to be included as a meta event in the midi file by the user (which is non-trivial as will be explained in MIDI recording. In addition, it can only process 3/4 and 4/4 time signatures, leaving out the $\genfrac{}{}{0pt}{}{2}{4}$ and all of the compound time signatures. As a result, a lot of music cannot be processed by this software. The same problem occurs with the tempo meta event. If it is not determined by the user then the program does not function properly.

Besides, it has some problems being able to separate the left-hand and right-hand notes when the left hand is positioned in a higher octave than usual. To conclude, it also has a problem of adding notes into a bar with a total duration bigger than the duration admissible by the time signature of the score. Some of these problems can be appreciated in the following comparison.



(a) Real Score



(b) Score created with Midi Sheet Music

Figure 3.2: Comparison between real and Midi Sheet Music scores

3.2 MuseScore

MuseScore is an open source and free music notation software. It has full support for playing scores and importing or exporting MusicXML and standard MIDI files. This software was launched in September 2002. From then, 17.277 commits have been made by 168 contributors [17].

Some of the features of this software are [17]:

- What You See Is What You Get (WYSIWYG) design, notes are entered on a ”virtual notepaper”.
- TrueType fonts for printing and display that allows for high quality scaling to all sizes.
- Easy and fast note entry.
- Many editing functions.
- MusicXML import/export.
- MIDI (SMF) import/export.
- MuseData import.
- MIDI input for note entry.
- Integrated sequencer and software synthesizer to play the score.
- Print or create pdf files.

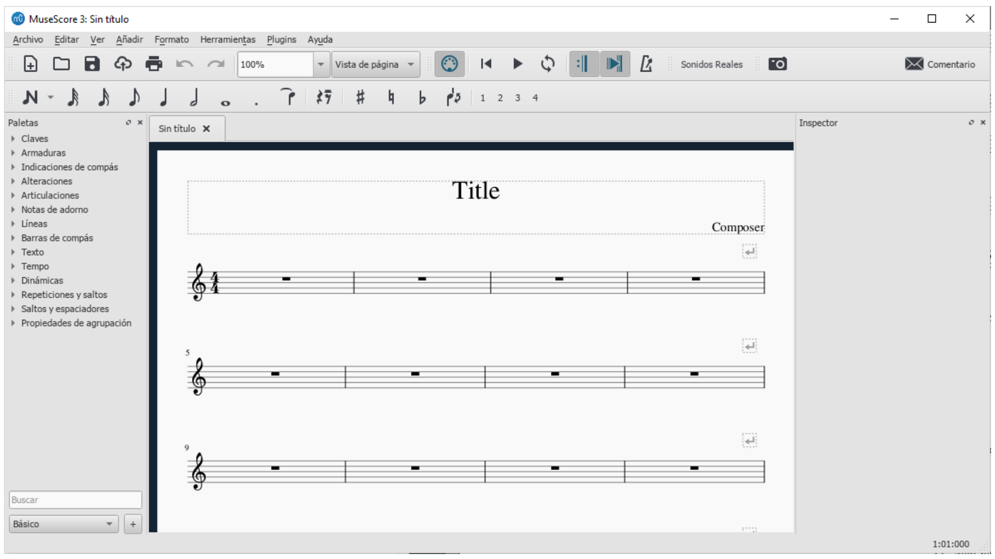
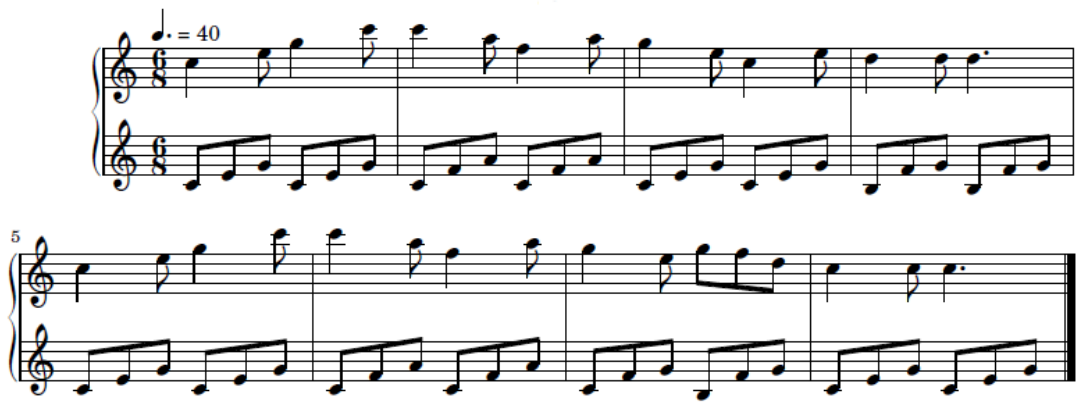


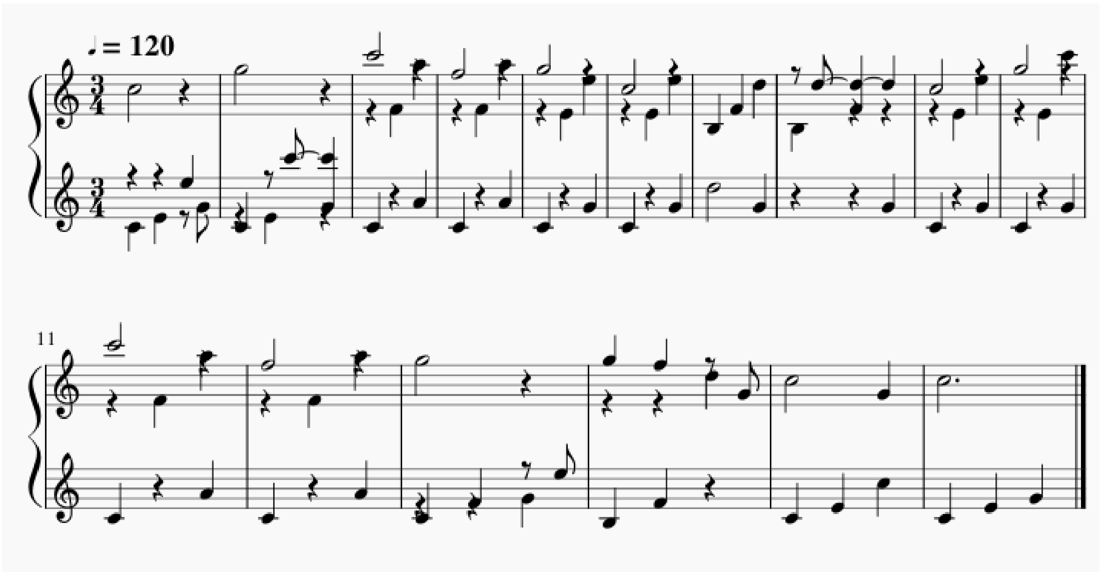
Figure 3.3: MuseScore UI [17]

The UI can be seen in Figure 3.3. It is a very complete UI that gives the user a large amount of tools to write music scores by hand. Everything that a composer would need to transcript a song is available in this software. Moreover, it is very intuitive, which makes the learning process short and smooth.

This program is really good in terms of music transcription. However, it has several problems when processing a MIDI file to automatically create a score. First, while the software is supposed to understand and process compound time signatures, whenever a song with a compound time signature is processed, museScore returns a simple time signature which makes the score completely wrong and unreadable as it can be seen in Figure 3.4



(a) Real Score

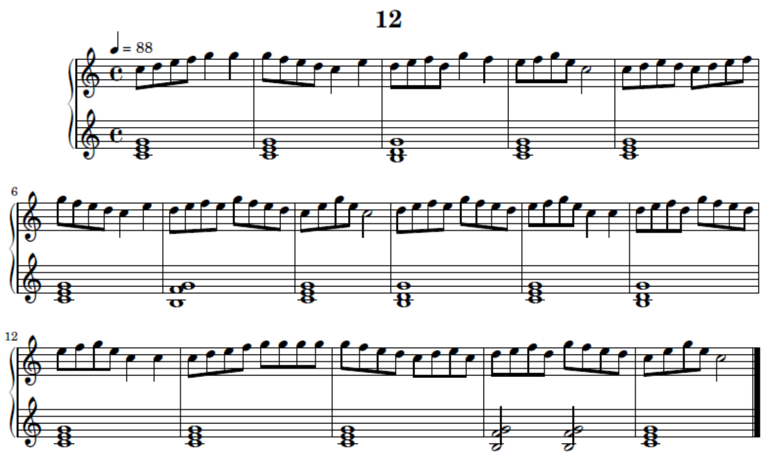


(b) Score created with MuseScore

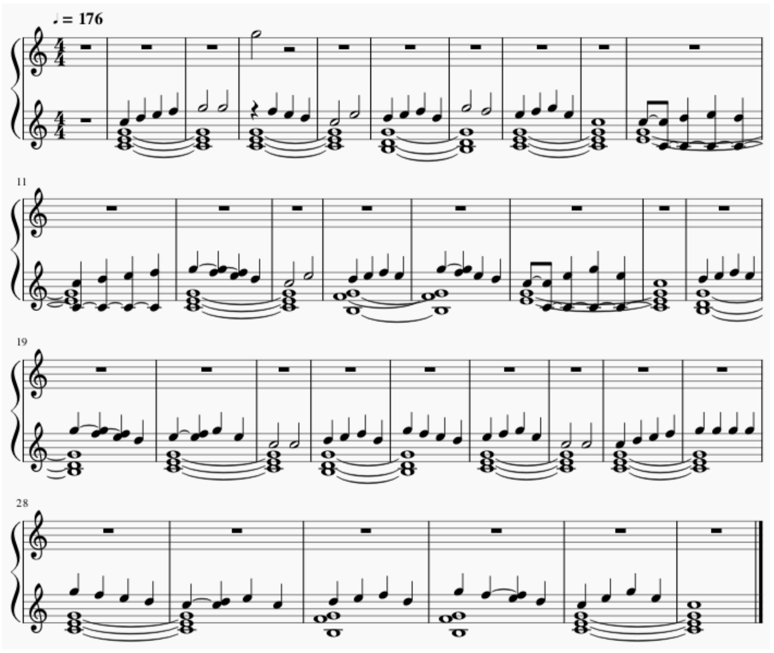
Figure 3.4: Comparison between real and MuseScore scores for compound time signature

Furthermore, the separation of the music notes in both hands is another task that museScore is not able to perform reliably as it can be observed in both comparisons made in this section.

The results obtained for simple time signatures are finer in most case scenarios. It is able to calculate the tempo and the key signature some times. The resultant score is satisfactory when the first note of the score is a quarter note as museScore usually gets the duration of this note and assigns its value to the duration of a quarter note. Nonetheless, when the first note figure is not a quarter note, the score then will be incorrect as it can be observed in Figure 3.5.



(a) Real Score



(b) Score created with MuseScore

Figure 3.5: Comparison between real and MuseScore scores for simple time signature

After analysing the two most known and used free software available, it can be stated that the results obtained have been unsatisfactory. Neither of them obtained acceptable results in general. This issues appear largely by means of the difficulty to work with the little information that MIDI provides, which obliges software to make too many difficult decisions with a strong impact in the final result.

Chapter 4

Tools and Technology

In this chapter, a justification of the different decisions taken in terms of the technology and tools used for the development of the project is going to be made.

4.1 Programming Language

The programming language chosen for the development of this project is Java. Java is a general-purpose, concurrent, class-based and Object Oriented (OO) programming language. It is strongly and statically typed. It clearly distinguishes between the compile-time errors that can and must be detected at compile time and those that can occur at run time. Java source-code is usually compiled to the bytecode instruction set and binary format defined in *The Java Virtual Machine Specification* [18].

Java was the programming language selected for the project development for two main reasons. First, because it includes packages which provide an interface for developers to work with MIDI messages. Finally, as a consequence of the experience the author has in this programming language which is crucial for the satisfactory development of the project.

4.2 Libraries and Development tools

4.2.1 MIDI parsing

Java Sound

The Java Sound Application Programming Interface (API) is a low-level API for effecting and controlling the input and output of sound media, including both audio and MIDI data. It provides mechanisms for installing, accessing, and manipulating system resources such as audio mixers, MIDI synthesizers, file readers and writers, sound format converters or MIDI devices. The Java Sound API comprises 4 packages. The package used in this project will be *javax.sound.midi* which provides an interface for MIDI synthesis, sequencing and event processing. With this package mostly every MIDI, meta and system exclusive event can be translated and processed. It will be used to parse the MIDI file and process the different events [19].

4.2.2 Score creation

Score creation is a critical task in the development of the desired software. The different analyzed options are:

Abc4j

Abc4j is a java library which provides an API to handle abc musical notation using java. It also provides a tool to parse tunes written in abc notation, and other classes in order to support midi playback, music score display, and other functionalities. The software can be extended to add or modify different features [20].

Abc notation was created in 1991 as a means of sharing folk songs and traditional tunes using only ASCII characters. It has become very popular in certain communities, and collections of tunes using this notation are available. However, the downside of this notation is that it does not support multiple voices nor defining instruments [21]. As a result, this library is discarded as it does not fit into the project requisites.

JFugue

JFugue is an LGPL-licensed open source library for programming music without the complexities of MIDI. It has its own notation to represent music using only ASCII characters, provides I/O to MIDI files and allows manipulating music programmatically [21].

JFugue differs from other Java music APIs in two significant ways. First, the music programming is done using strings to specify notes, instrument changes, controller messages and other musical data. These string are quick to create and easy to understand in contrast with other APIs that use objects to represent each note in a the musical score, which makes music programming tedious. Finally, JFugue lets users create ”patterns” of music that can be manipulated, recombined, and transformed, which adds a lot of functionality [22].

However, issues appear when the creation of the music score has to be done. All the programming in JFugue is done with ASCII text that has to be processed to create and display the score to the user. While JFugue musical notation is fairly complete it is not standard and few other applications use that notation. An interesting project called *JFugure Music NotePad* is under way to create a GUI fronted for JFugue [21]. Currently, it only supports a subset of the overall facilities of JFugue, and it is not valid for scores with multiple voices. As a consequence, this library is discarded as it does not fit into the project requisites.

LilyPond

LilyPond is a cross-platform computer program and file format for music engraving, devoted to produce the highest possible quality sheet music. It translates the aesthetics of traditional music scores to computer printouts. It is a free software and part of the GNU Project [23].

LilyPond is a text-based application, so it does not contain its own graphical user interface. It uses a simple text notation for music input that strives to be simple, smoothing the learning curve for new users. Then the file is interpreted and processed in a series of stages [24]. In the final stage, the music notation is output to PDF via PostScript or to other graphical formats as Scalable Vector Graphics (SVG) or PNG. Lilypond adheres to the What You See Is What You Mean (WYSIWYM) paradigm, being the work-flow of writing the music notation similar to that of preparing documents with LaTeX.

LilyPond has support for every possible element needed to create a musical score. Besides, it has very complete manuals and documentation explaining how the program works and how to perform every possible action as well as some features to produce an output score comparable to professionally engraved scores. Some of these features are [25]:

- **Optical font scaling.** Depending on the staff size, the design of the music font is altered. As a result, note heads become more rounded, and staff lines become thicker.
- **Optical spacing.** Stem directions are taken into account when spacing subsequent notes.
- **Special ledger line handling.** Ledger lines are shortened when accidental elements such as sharp and flat appear enhancing readability.
- **Proportional spacing.** Notes can be positioned in such a way that exactly reflects their duration.

As a consequence of all of these features as well as its ease of use and understanding, Lilypond will be the software used to create the music score in this project.

4.2.3 MIDI recording

There are many software available for MIDI recording. Some examples are Reaper, Anvil Studio, MixPad Midi Editor, MultitrackSttudio, Gig Performer and SynthFont 2. Every one of them have a common attribute, they are payment software. One of the aims of this project is that the software has to be able to process MIDI files recorded by everyone. Therefore, these programs are discarded as not everybody can afford the licenses price.

MidiEditor

MidiEditor is a free software that provides an interface to edit, record and play MIDI data. This editor is able to open existing MIDI files and modify their content. New files can be created by either recording the data from a connected MIDI device or by manually creating new notes and events. Tracks, channels and MIDI events can also be easily modified using MidiEditor [26].

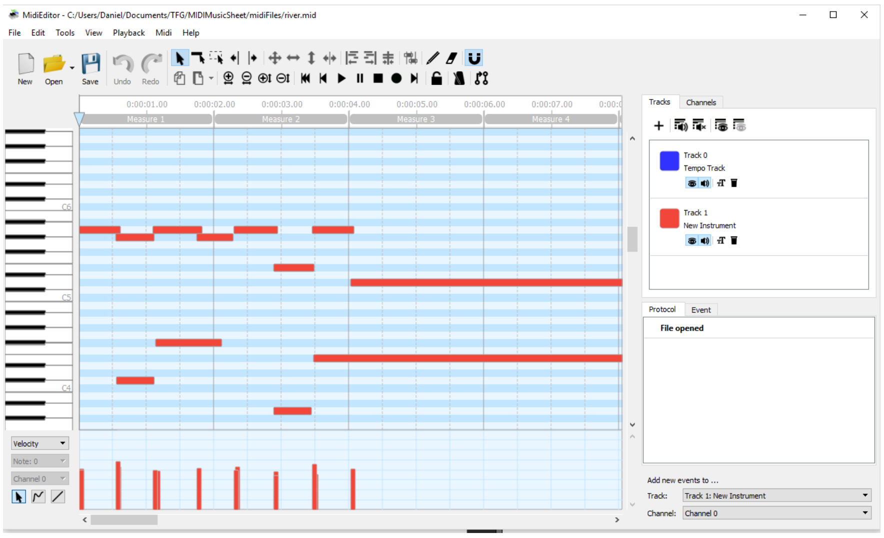


Figure 4.1: Midi Editor UI

MidiEditor has a quite simple UI that is easy to use. To record a MIDI file, the procedure to follow is:

1. Connect the electronic music keyboard to the PC.
2. Press the record button.
3. Play the song on the keyboard.
4. Save the song as a MIDI file.

As it can be noticed, the process to record a MIDI file is straightforward. Moreover, the edition of the file is pretty simple as well. The bars represented in Figure 4.1 are the notes of the recorded song. The user can change the length and position of every bar, changing the duration and note pitch in the song. Finally, new notes and tracks can be added if it was required.

It needs to be taken into account the factor that every song recorded with this program will have the default values for tempo and key signature, which are 120 and 4/4. The tempo can be changed, but the unit used by the program is Ticks Per Beat (TPB), which is different from the unit used in real life scores which is Beats Per Minute (BPM). In order to set the exact value some calculations would have to be done to translate both units but the average user will not carry out those operations.

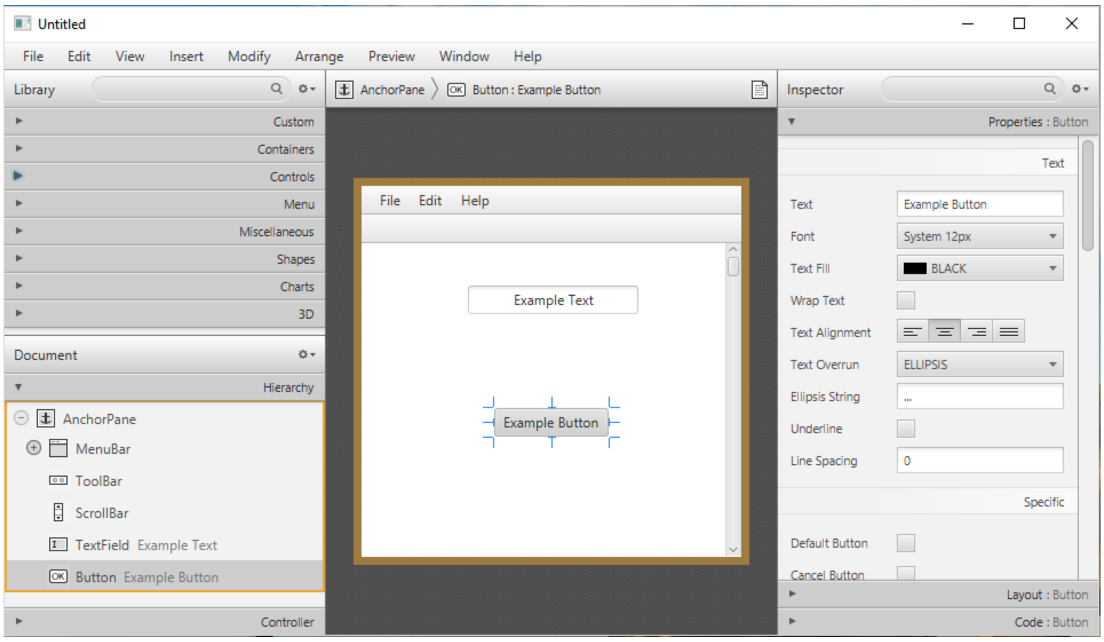
This simple piece of software is chosen for MIDI recording in this project.

4.2.4 User Interface

JavaFX 13

JavaFX is a set of graphics and media packages that enables developers to design, create, test, debug, and deploy rich client applications that operate consistently across diverse platforms. Generally, applications build with JavaFX are network-aware applications that are deployed across multiple platforms and display information in a high-performance modern UI that features audio, video, graphics and animation [27].

It follows the structural pattern Model View Controller (MVC) in which the view is managed by a *.fxml* file that is written in Extensible Markup Language (XML). For the creation of the UI, the JavaFX Scene Builder will be used to help with the design process.



Text	Example Button
Font	System 12px
Text Fill	BLACK
Wrap Text	
Text Alignment	
Text Overrun	ELLIPSIS
Ellipsis String	...
Underline	
Line Spacing	0
Default Button	
Cancel Button	

Figure 4.2: JavaFX Scene Builder UI

JavaFX Scene Builder is a visual layout tool that lets users quickly design JavaFX application user interfaces without coding. Users can drag and drop UI components to the work area, modify their properties, apply style sheets and other operations. Besides, the code of the

FXML file is automatically generated in the background [28]. As it can be seen in Figure 4.2 the UI is assembled with 4 different views.

- The Library view, which contains the UI components.
- The Hierarchy view, which contains the structure of the components that constitute the UI.
- The Work view, which contains the UI representation.
- The Inspector view, which contains modifications that can be performed to the properties, layout and code operations of each component.

FontAwesomeFX

JavaFX does not include icons as UI components. So, in order to display icons the FontAwesomeFX library is used. These icons are SVG in format and can be customized using Cascading Style Sheets (CSS). FontAwesomeFX is compatible with SceneBuilder which makes it a perfect fit for the project.

Apache PDFBox

JavaFX does not include interfaces to perform operations with PDF file and, therefore, not being able to display the PDF score created by LilyPond. The Apache PDFBox library is an open source Java tool for working with PDF documents. It allows the creation of new PDF documents, the manipulation of existing ones and the ability to extract content from them [29].

This library is used to get the resultant PDF content and transform it in a set of images that will be displayed in the UI.

4.3 Support tools

IntelliJ IDEA

IntelliJ is a multi-language Integrated Development Environment (IDE) written in Java for developing computer software. The entirety of the code has been written in this platform. Some features of this IDE are [30].

- **Coding assistance.** It provides code completion by analysing the context, code navigation and code refactoring, giving options to fix inconsistencies via suggestions.

- **Built in tools and integration.** It supports integration with build and packaging tools like grunt, gradle and SBT. In addition, it supports version control systems like Git, Mercurial, Perforce and SVN.
- **Plugin ecosystem.** It supports different plugins to add additional functionality to the IDE.
- **Supported languages.** It supports many different programming languages that can be added via plugins.

It is available as a free community edition and in a proprietary commercial edition. If the user is a student or a teacher, the commercial edition is free of charge.

Balsamiq Mockups

Balsamiq Mockups is a graphical user interface mockup and website wireframe builder application. A mockup is a scale model of a design or device, used for teaching, demonstration, design evaluation, promotion and other purposes. It is a commercial software that has a 30 day free trial. Balsamiq was the tool selected due to the experience of the author with the tool.

Git

Git is a distributed version-control system for tracking changes in source code during software development. It is designed for coordinating work among programmers as well as tracking changes in any set of files. The git repository will be stored in Bitbucket, which is a web-based version control repository hosting service for source code and development projects that use either Mercurial or Git revision control systems. It offers both commercial plans and free accounts. Free users can only create a limited number of private repositories. Bitbucket was selected because every team member had experience working with it.

Chapter 5

Planning and methodology

Planning is an essential process in every engineering project. Through planning, an estimation of the critical tasks, the amount of time required, the costs and the resources needed to carry out the project successfully are done. Besides, a specific development methodology needs to be defined so as to structure the work flow of the software development.

First, the methodology selected for the development of the project is defined. Afterwards, the planning of the project is exposed, describing the tasks, costs and resources involved.

5.1 Methodology

The methodology chosen for the project development is iterative and incremental. The iterative and incremental software development methodology was created as a response to the weaknesses of the traditional waterfall model.

An iteration is the repetition of a process in order to generate a sequence of outcomes. The sequence of iterations is intended to produce the final product by refining it at each iteration. Each repetition of the process is a single iteration and the outcome of each iteration is then the starting point of the next iteration [31].

The iterative and incremental development methodology is a combination of both iterative design and incremental development. The general idea of this methodology is to develop a system through iterations (repeated cycles) and incrementally (small portions at a time). Working through iterations means that the development of the application is split into smaller chunks. Each iteration follows the same structure of a typical process in any software project development which is analysis, design, implementation and testing. At each iteration, the development team tries to prepare just what is needed for the successful delivery at the end of the cycle [32]. This methodology allows team members and product stakeholders to take advantage of what was learned during the development of earlier parts of the system for the following iterations.

It is a common practice that each iteration is finished with a demo to the clients. The demo is used as a learning process in order to detect problems in the software, in the iteration planning or even the scope of this project. Since the a functional version of the software is available much earlier, it will be easier to detect problems in the early iterations of the development process.

All of these procedures makes the development methodology flexible to adapt to changes which is ideal for the development of the project to compensate with the lack of knowledge of the domain by the author. Some requisites and objectives of the project had to be refined or modified and the use of this methodology has allowed the detection of problems fast and efficiently.

5.2 Planning

5.2.1 Resources

Three different type of resources can be analysed in this project:

Human resources

The development team of the project is assembled by three workers. As a consequence, the author will take different roles depending the type of work done. The different types of human resources in this project are:

- **Director**. The project director is an expert in the filed that sets the main lines of the project and revises that everything is being executed as expected. This role is going to be assumed by the project directors Óscar Fresnedo Arias and José Pablo González Coma who will help the author obtain the requisites of the application as well as supervising every aspect of the development process.
- **Analyst**. The Analyst is in charge of analysing the system to develop, obtaining the functional and non-functional requirements and writing down the Software Requirements Specification (SRS). This role will be assumed by the author.
- **Developer**. The Developer is responsible of identifying, producing and testing the software system following the specifications made by the analysts. This role will also be assumed by the author.

Material resources

The material resources used in this project includes the author personal computer, a electronic keyboard an a MIDI cable to connect both devices.

Software resources

These resources would correspond to the software mentioned in the Tools and Technology chapter. Every tool used was either open source or had a free student license.

The costs of the different human resources per hour in euros can be seen in Table 5.1

Team Members	Director	Analyst	Developer
Daniel Pena Docampo	-	26€	20.8€
Óscar Fresnedo Arias	45.5€	-	-
José Pablo Gonzalez Coma	45.5€	-	-

Table 5.1: Cost of the human resources per hour in euros

The cost assigned for each human resource in the project was based in the report *Estudio salarial - Sector TIC en Galicia 2015-2016* [33]. Note that the salaries calculated in the report also include social security costs.

5.2.2 Project planning

The iterations and tasks planned for the development of the project are:

- **Iteration 1.** General Analysis of the project.

  T1.1 Study of the MIDI specification.

  T1.2 Analyse software requirements.

  T1.3 Develop the Software Requirements Specification SRS.

- **Iteration 2.** Basic version of the program. It prints on the screen the notes located in the different pulses and the key signature of the score.

  T2.1 Parse MIDI events.

  T2.2 Calculate pulse duration.

  T2.3 Divide the score in pulses.

  T2.4 Match the note_on note_off events to obtain notes information.

  T2.5 Place each note in the correspondent pulse.

  T2.6 Calculate the key signature.

- **Iteration 3.** LiLyPond output parser implementation and reation of PDF file of the score.

T3.1 Study of the LilyPond text notation.

T3.2 Detection of the different scenarios to print (Chords, combinations of different note figures in a pulse, etc)

T3.3 Implement the LilyPond adapter parser class.

- **Iteration 4.** Implement an advanced version of the software. The score created is composed by a grand staff with their respective clefs and time signature, where the notes of each pulse are separated in both hands and they are join by ties when necessary.

T4.1 Detect which note figure is the beat of the composition.

T4.2 Calculate the key signature.

T4.3 Separate notes in right and left hand notes.

T4.4 Calculate the clefs for upper and lower staffs.

T4.5 Introduce ties between notes where needed.

T4.6 Adapt the LilyPond parser class to these new features.

- **Iteration 5.** Creation of the UI.

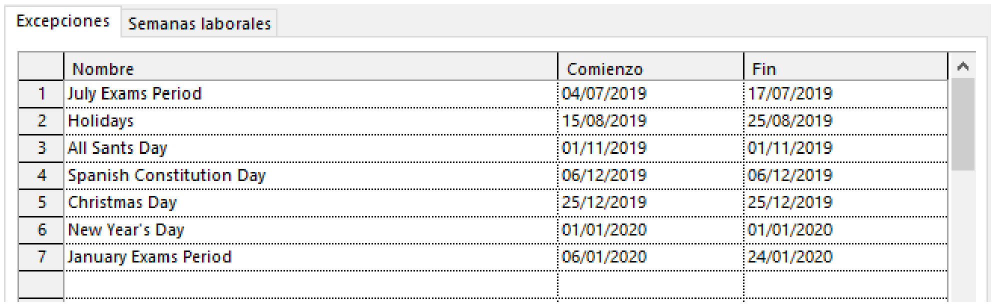
T5.1 Study JavaFX and SceneBuilder.

T5.2 Create the representation of the UI.

T5.3 Add functionality to the UI elements.

The duration of each iteration has been established as a month and one week, looking for a balance between quantity and quality of work. Since the project is being developed in parallel with undergraduate studies, the author will be working on this project 3 hours a day. Thus, the amount of time spent in the project development each iteration will be 75 hours.

This project is expected to begin on 17th January 2019 and finish on 27th January 2019. Besides, the project development is expected to be paused during some weeks or days. The days and reasons can be observed in Figure 5.1.



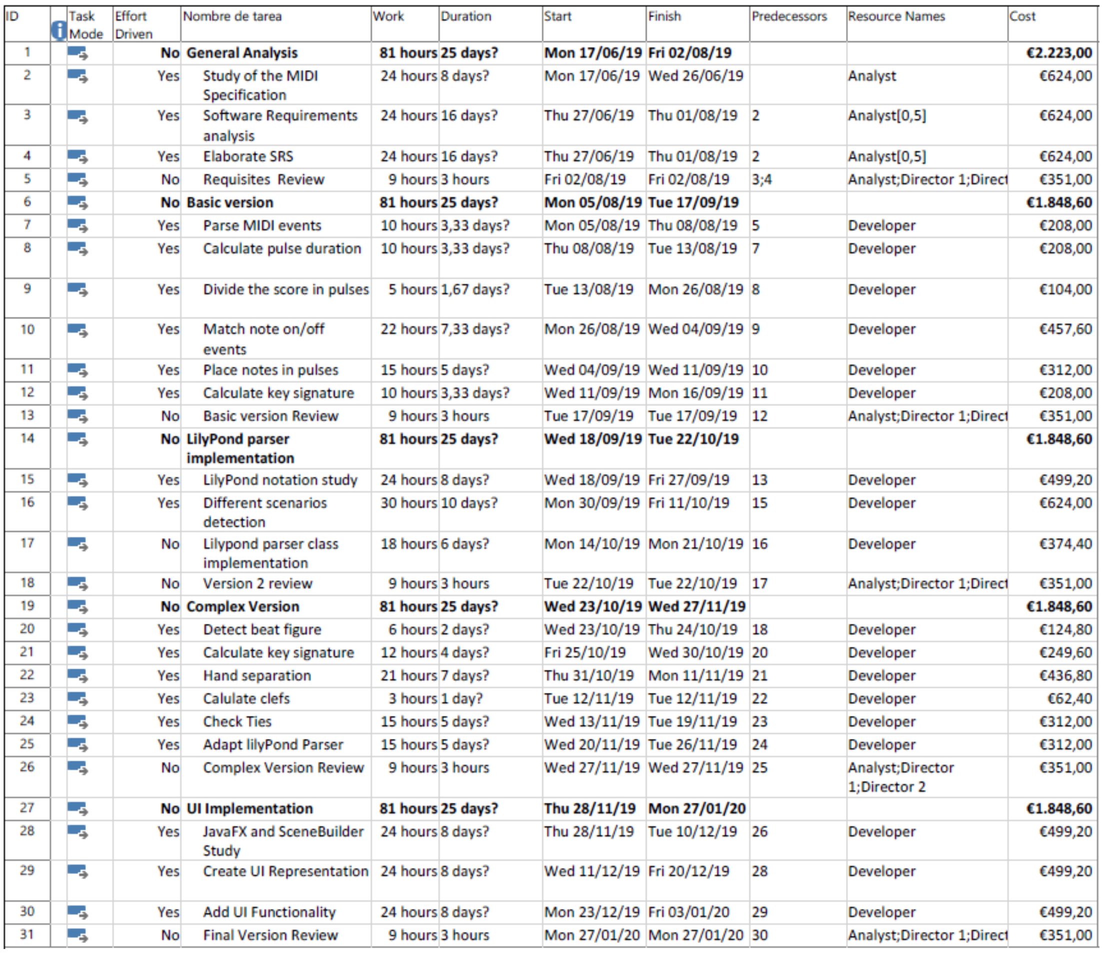
	Nombre	Comienzo	Fin
1	July Exams Period	04/07/2019	17/07/2019
2	Holidays	15/08/2019	25/08/2019
3	All Sants Day	01/11/2019	01/11/2019
4	Spanish Constitution Day	06/12/2019	06/12/2019
5	Christmas Day	25/12/2019	25/12/2019
6	New Year's Day	01/01/2020	01/01/2020
7	January Exams Period	06/01/2020	24/01/2020

Figure 5.1: Exceptions in the project development

The key points of the project development are summarized below. Table 5.2 shows the general information of the planning. Figure 5.2 presents detailed information of every task planned and Figure 5.3 exhibits the Gantt diagram of the project.

	Forecast
Start Date	17/06/19
Finish Date	27/01/20
Duration	125 Days
Work	405 Hours
Cost	9617.40€

Table 5.2: General project planning



ID	Task Mode	Effort Driven	Nombre de tarea	Work	Duration	Start	Finish	Predecessors	Resource Names	Cost
1		**No**	**General Analysis**	**81 hours**	**25 days?**	**Mon 17/06/19**	**Fri 02/08/19**			**€2.223,00**
2		Yes	Study of the MIDI Specification	24 hours	8 days?	Mon 17/06/19	Wed 26/06/19		Analyst	€624,00
3		Yes	Software Requirements analysis	24 hours	16 days?	Thu 27/06/19	Thu 01/08/19	2	Analyst[0,5]	€624,00
4		Yes	Elaborate SRS	24 hours	16 days?	Thu 27/06/19	Thu 01/08/19	2	Analyst[0,5]	€624,00
5		No	Requisites Review	9 hours	3 hours	Fri 02/08/19	Fri 02/08/19	3;4	Analyst;Director 1;Direct	€351,00
6		**No**	**Basic version**	**81 hours**	**25 days?**	**Mon 05/08/19**	**Tue 17/09/19**			**€1.848,60**
7		Yes	Parse MIDI events	10 hours	3,33 days?	Mon 05/08/19	Thu 08/08/19	5	Developer	€208,00
8		Yes	Calculate pulse duration	10 hours	3,33 days?	Thu 08/08/19	Tue 13/08/19	7	Developer	€208,00
9		Yes	Divide the score in pulses	5 hours	1,67 days?	Tue 13/08/19	Mon 26/08/19	8	Developer	€104,00
10		Yes	Match note on/off events	22 hours	7,33 days?	Mon 26/08/19	Wed 04/09/19	9	Developer	€457,60
11		Yes	Place notes in pulses	15 hours	5 days?	Wed 04/09/19	Wed 11/09/19	10	Developer	€312,00
12		Yes	Calculate key signature	10 hours	3,33 days?	Wed 11/09/19	Mon 16/09/19	11	Developer	€208,00
13		No	Basic version Review	9 hours	3 hours	Tue 17/09/19	Tue 17/09/19	12	Analyst;Director 1;Direct	€351,00
14		**No**	**LilyPond parser implementation**	**81 hours**	**25 days?**	**Wed 18/09/19**	**Tue 22/10/19**			**€1.848,60**
15		Yes	LilyPond notation study	24 hours	8 days?	Wed 18/09/19	Fri 27/09/19	13	Developer	€499,20
16		Yes	Different scenarios detection	30 hours	10 days?	Mon 30/09/19	Fri 11/10/19	15	Developer	€624,00
17		No	Lilypond parser class implementation	18 hours	6 days?	Mon 14/10/19	Mon 21/10/19	16	Developer	€374,40
18		No	Version 2 review	9 hours	3 hours	Tue 22/10/19	Tue 22/10/19	17	Analyst;Director 1;Direct	€351,00
19		**No**	**Complex Version**	**81 hours**	**25 days?**	**Wed 23/10/19**	**Wed 27/11/19**			**€1.848,60**
20		Yes	Detect beat figure	6 hours	2 days?	Wed 23/10/19	Thu 24/10/19	18	Developer	€124,80
21		Yes	Calculate key signature	12 hours	4 days?	Fri 25/10/19	Wed 30/10/19	20	Developer	€249,60
22		Yes	Hand separation	21 hours	7 days?	Thu 31/10/19	Mon 11/11/19	21	Developer	€436,80
23		Yes	Calulate clefs	3 hours	1 day?	Tue 12/11/19	Tue 12/11/19	22	Developer	€62,40
24		Yes	Check Ties	15 hours	5 days?	Wed 13/11/19	Tue 19/11/19	23	Developer	€312,00
25		Yes	Adapt lilyPond Parser	15 hours	5 days?	Wed 20/11/19	Tue 26/11/19	24	Developer	€312,00
26		No	Complex Version Review	9 hours	3 hours	Wed 27/11/19	Wed 27/11/19	25	Analyst;Director 1;Director 2	€351,00
27		**No**	**UI Implementation**	**81 hours**	**25 days?**	**Thu 28/11/19**	**Mon 27/01/20**			**€1.848,60**
28		Yes	JavaFX and SceneBuilder Study	24 hours	8 days?	Thu 28/11/19	Tue 10/12/19	26	Developer	€499,20
29		Yes	Create UI Representation	24 hours	8 days?	Wed 11/12/19	Fri 20/12/19	28	Developer	€499,20
30		Yes	Add UI Functionality	24 hours	8 days?	Mon 23/12/19	Fri 03/01/20	29	Developer	€499,20
31		No	Final Version Review	9 hours	3 hours	Mon 27/01/20	Mon 27/01/20	30	Analyst;Director 1;Direct	€351,00

Figure 5.2: Initial planning tasks

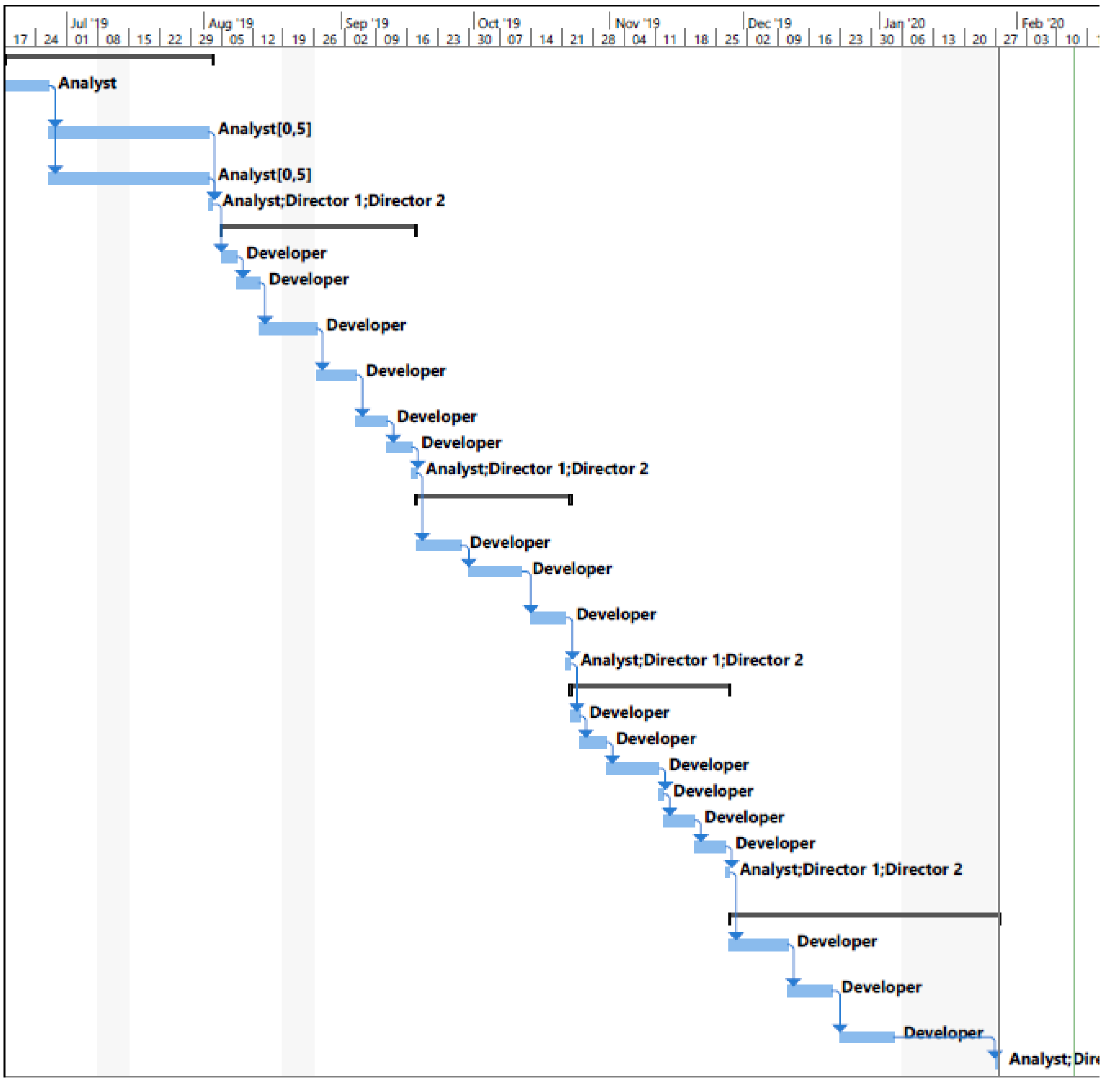


Figure 5.3: Initial planning Gantt diagram

5.2.3 Risk Management

A vital phase when planning a project is the risk management. Risk management consists in identifying, studying and eliminating sources of risk before they begin to threaten the successful completion of a project. The principal risks present in the project development are:

- **Lack of knowledge of the domain.** This lack of knowledge might provoke a low efficient initial implementation which does not meet the established requisites as well as bad decisions through the development project in terms of software and libraries chosen as none of them have been used before by the author. As a response, a great effort will be put in the testing process and analysis of the different versions obtained in each iteration.

- **One-man developing team.** The developing of the project by the author on his own might provoke a bad analysis of the needed requisites as well as poor design decisions that might end up causing a bad quality software. This risk is minimized by the vital helping labour done by the directors of the project that will give the author different points of view and help with the analysis and design.
- **Conexion failure between the keyboard and PC.** The connection between the keyboard and the PC is done using a MIDI cable. This connection may fail because the cable is broken or the interfaces are incompatible or for other handful of reasons. This risk cannot be eliminated but it can be mitigated by using the official MIDI cable of the same brand as the keyboard to ensure no incompatibility issues happening.

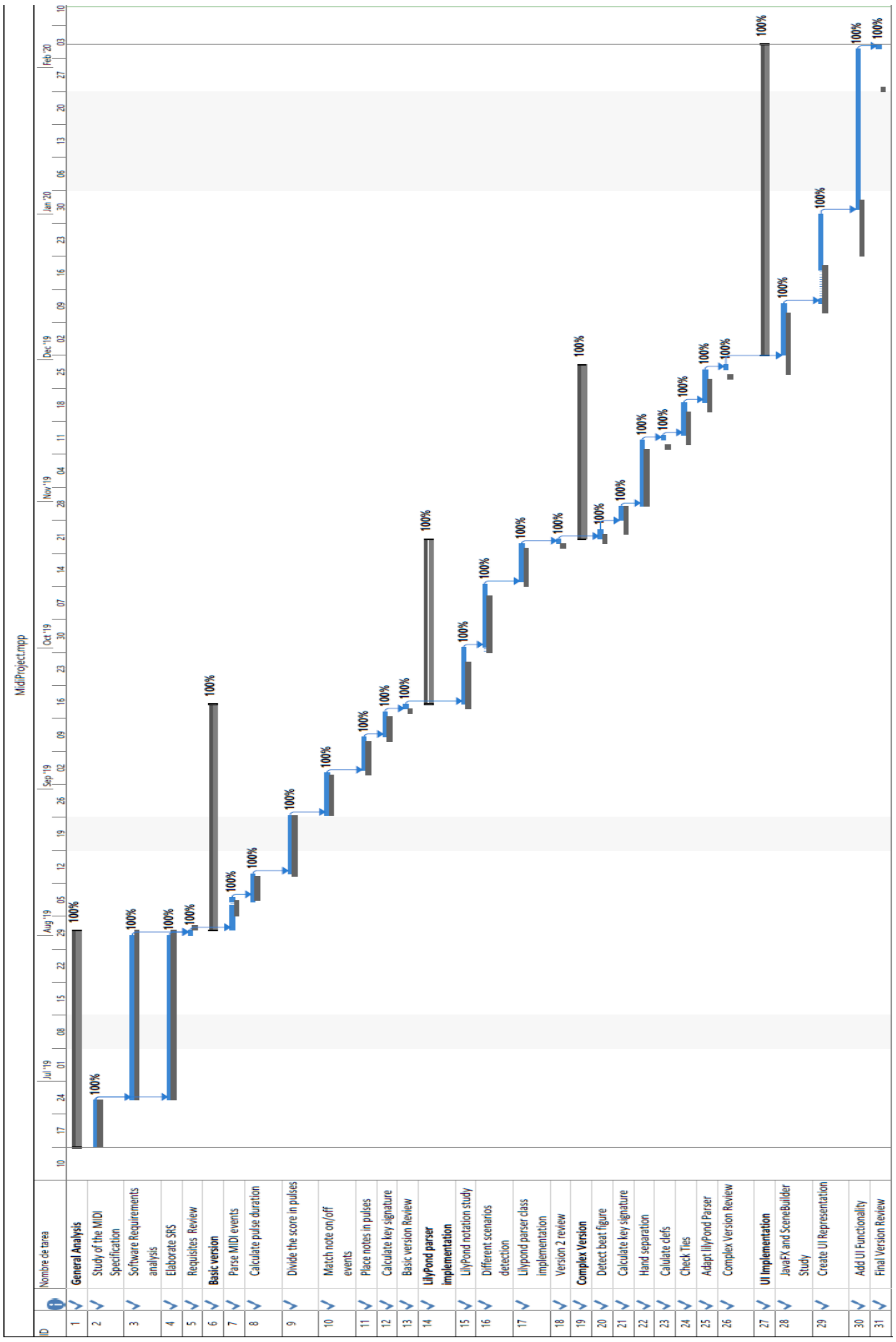
5.2.4 Project follow-up

In general, the development of the project adapted to the planning made. Some tasks needed a pair of additional hours to be done but most of them were compensated by the early finishing of other tasks in the project. The real duration of the project was 6 days (18 hours) longer than the planned one. This variation happened for two reasons. The first reason is that the *Hand Separation* task duration was 2 days (6 hours) longer than expected, as the operation was more complex than predicted. The second reason is that while implementing the task *Create UI representation* the author caught a flu that did not let him work in the project for 4 days.

In Table 5.3 the comparison of the general information between the initial planning and the follow-up can be seen. Besides, the comparison between the initial planning Gantt diagram and the follow-up Gantt diagram is displayed in Figure 5.4.

	Forecast	Follow-up
Start Date	17/06/19	17/06/19
Finish Date	27/01/20	04/02/20
Duration	125 Days	131 Days
Work	405 Hours	410 Hours
Cost	9617.40€	9705.80€

Table 5.3: Comparison between planning and follow-up.



ID	Nombre de tarea
1	**General Analysis**
2	Study of the MIDI Specification
3	Software Requirements analysis
4	Elaborate SRS
5	Requisites Review
6	**Basic version**
7	Parse MIDI events
8	Calculate pulse duration
9	Divide the score in pulses
10	Match note on/off events
11	Place notes in pulses
12	Calculate key signature
13	Basic version Review
14	**LilyPond parser implementation**
15	LilyPond notation study
16	Different scenarios detection
17	Lilypond parser class implementation
18	Version 2 review
19	**Complex Version**
20	Detect beat figure
21	Calculate key signature
22	Hand separation
23	Calulate clefs
24	Check Ties
25	Adapt lilyPond Parser
26	Complex Version Review
27	**UI Implementation**
28	JavaFX and SceneBuilder Study
29	Create UI Representation
30	Add UI Functionality
31	Final Version Review

Figure 5.4: Comparison of initial planning and follow-up Gantt diagrams.

Chapter 6

Analysis

The requirements analysis is critical to the success or failure of a software project. It focuses on the tasks that have to be implemented in order to achieve the project objectives. This analysis was made in the early stages of the project.

6.1 Requirements analysis

An initial analysis is done to determine a series of requirements the software has to meet in order to produce high quality music scores.

Actors

The software is a desktop application that will have one actor, which is the user of the application. There is no distinction between users, everyone has access to the totality of the functionality of the program.

6.1.1 Functional requirements

The functional requirements include every feature that the system needs to have. The functional requirements of the system are:

- **Process pre-recorded MIDI file.** As stated in sections above, MIDI files may contain more or less information about the song depending on the capabilities of the software used to record it. The system has to be able to create a score with the least amount of information so every MIDI file can be processed.
- **Process MIDI piano recordings.** The piano is the instrument that creates the more complex scenarios so then the extension to other instruments will be easy.

- **Error management.** The system has to be implemented to give response to a user error such as trying to run the program without a MIDI file, with a file with another extension, etc.
- **Create scores similar to real scores.** The score processed by the system has to be as similar as possible to the real score of the song (if the song recorder has a music score). While there are some elements that are impossible to detect (it will be explained in development), the more similar the created and the real scores are the greater quality of the product.
- **UI Implementation**. The system should have a UI to interact with the user and display the music score created.
- **Music player.** The application needs to have a music player so the user can hear the music transcripted in the score.

6.1.2 Non functional requirements

The non functional requirements include the features that system has to implement in order to have good quality. These requirements represent a set of standards used to judge the specific operation of a system. The non functional requirements of the application are:

- **Ease to use.** The system is implemented to be simple to use and understand.
- **Adaptable UI**. The score displayed has to be adaptable to different size windows.
- **Process time.** Some of the process time is out of the control of the system as it depends on LilyPond score generation. The system has to be as fast as possible to compensate for the possibility of a slow score generation.

6.1.3 Prototypes

The prototype for the UI of the application can be seen in Figure 6.1. As observed, it was desgined with the aim of providing a simple and intuitive interface to allow users to easily load a MIDI file, execute the program and use the player.

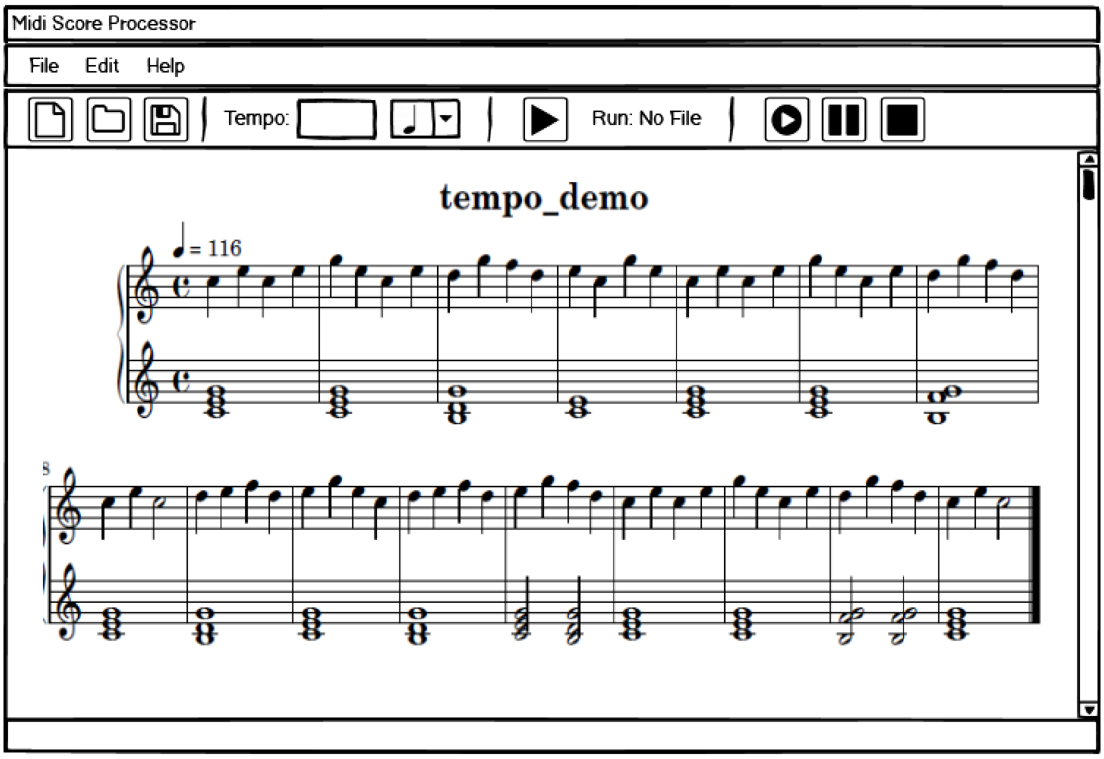


Figure 6.1: UI Prototype

6.2 Architecture

The architecture of the system will be discussed in this section. The proposed architecture is displayed in Figure 6.2

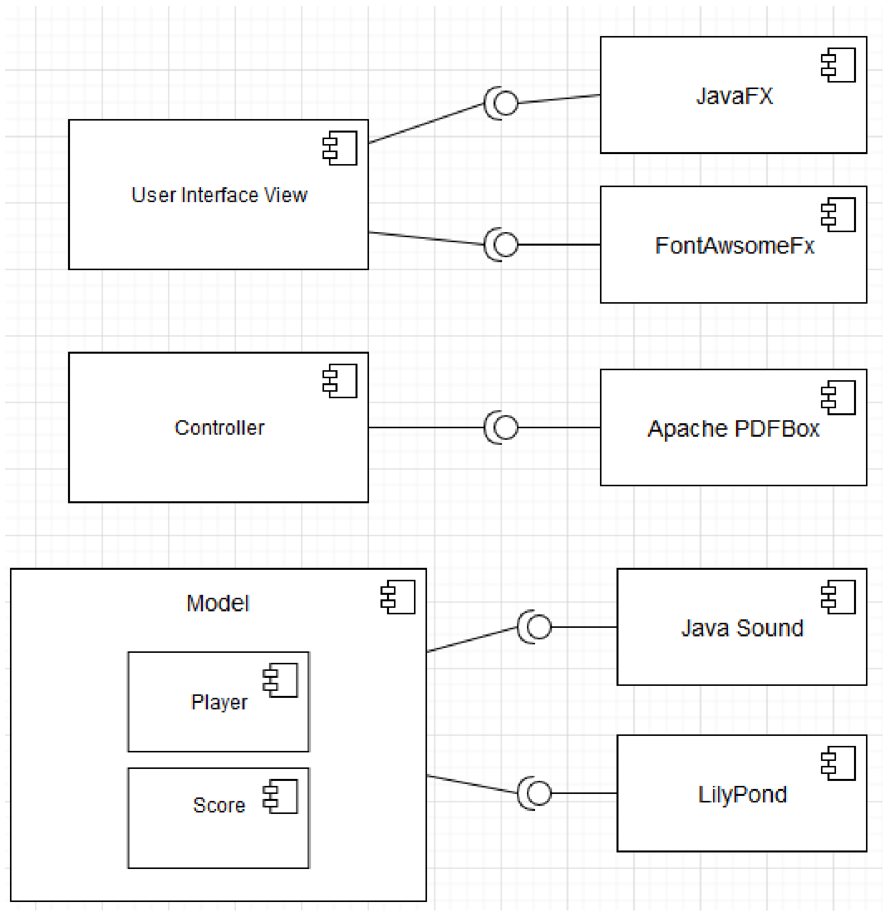


Figure 6.2: Application Architecture

The architecture proposed follows the structural pattern MVC to separate the representation of the UI from the logic of the system. The three components of this architecture are:

- **View**. The view is the representation of the UI presented to the user when using the application.
- **Controller**. The controller is the component in charge of linking the user and the system. The user will use the controller to manipulate the model.
- **Model**. The model represents the knowledge and data of the application. It updates the views when modified.

In addition, the pattern subsystem interface is used to divide the model system into the subsystems score and player.

Chapter 7

Development

This chapter contains every step done throughout the development phase of the project. The design decisions and the corresponding operations done at each task defined in the requirements phase is going to be explained in detail.

7.1 Iteration 1

The first iteration of the process was of great importance in order to clarify what information is needed to be able to create accurate scores.

One piece of information which is essential to process MIDI files is the tempo of the composition. When processing the events of a MIDI file, the different notes of the piece can be detected. Then, the event times of note_on and note_off for each note can be subtracted in order to get its duration. The problem arises when assigning the musical note figure to each note detected. There is no reference whatsoever of the expected duration of a quarter note, half note or eight note.

When trying to solve this issue, some software such as MuseScore gets the duration of the first note and assign the obtained value to the beat figure. It has already been demonstrated that the first note of a composition may be many different note figures so this assumption is wrong and will lead to unsatisfactory results.

At this point, the tempo of the score comes into play. If the tempo in BPM is known, then the duration of the beat of the song can be calculated. Once this information is known, the expected duration of each possible note figure possible is easy to measure. Finally, a note figure will be assigned to each note in the score by comparing their duration values.

As explained in previous sections, some recording programmes cannot change the default tempo of the MIDI file so every one will have the default value for tempo regardless of the real tempo of the composition. Every MIDI file contains the information of the number of BPM, in order to solve this problem and process every MIDI file the following operations need to be done.

First, the system will ask the user to introduce the tempo of the score in BPM before executing the program. The tempo of the MIDI file is represented by (M)BPM, the MIDI file TPB variable is represented as (M)TPB and the tempo introduced by the user in BPM is represented by (R)BPM. The operations needed to obtain the duration of the beat are:

1. Perform a calculation of the microseconds per beat using the tempo of the MIDI file.

$$\frac{60}{\text{(M) BPM}} \cdot 10^6 \; (ms/beat) \tag{7.1}$$

2. With the obtained value, calculate the duration of 1 tick in the MIDI file in microseconds.

$$\frac{ms/beat}{\text{(M) TPB}} -> (ms/tick) \tag{7.2}$$

3. Calculate the microseconds per beat value of the real tempo introduced by the user.

$$\frac{60}{\text{(R) BPM}} \cdot 10^6 \; (R)(ms/beat) \tag{7.3}$$

4. With the values obtained in the two last operations, calculate the number of ticks per beat for the real tempo.

$$\frac{\text{(R) ms/beat}}{ms/tick} \; (tick/beat). \tag{7.4}$$

Then, the duration of the beat is known and the problem is solved.

7.2 Iteration 2

In this iteration a basic version of the software was developed.

7.2.1 Pulse

Pulse is a class of the system. In this iteration of the development, the pulse will have as attributes an array with every note in the pulse, the pulse start and end times in ticks, its duration and the velocity of the pulse, which is the sum of the velocity of every note in the pulse that will be used in next iterations.

The duration of the pulse is the same as the duration of the beat. Then, by doing the operations explained above, its duration value can be obtained.

When the MIDI file is parsed, a sequencer object that contains all the information in the file is obtained. The sequencer implements a function that retrieves the length in ticks of the sequence (composition). Therefore dividing the duration of the sequence by the duration of a pulse, the number of pulses of the compositions is attained. Finally, the resultant number of pulses are created. These pulses will form a sequential array in which the end time of one pulse is the start time of the next one.

7.2.2 Notes

Note is another class of the system. Notes are the basic element of the system as every operation will be done around their information. Notes attributes are note pitch, note figure, octave, start and end times, duration and keyboard key.

As explained in the Fundamentals and main concepts chapter, the note_on and note_off events are independent. Hence, it is necessary to perform some operations to relate both of them. In addition, these events contain the key and velocity of the note. Again, some operations are required to obtain the desired information.

One assumption may be that every note_on event will be related with the next note_off event that is received while parsing the MIDI file. In figure 7.1 it can be seen that the previous assumption is wrong.

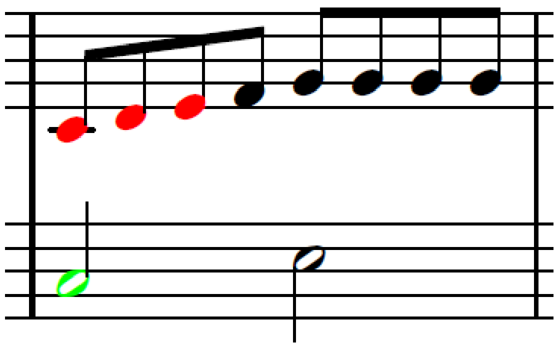


Figure 7.1: Note events example

There are three eight notes coloured in red and a half note coloured in green. In this example, the first red note and the green one would receive the note_on event at about the same time. However, when the note_off event of the green note occurs, every red note has finished so their corresponding events have also been already processed.

In order to relate both events, an identifier has to be found for each note that separates them from each other. That identifier is the key contained in the MIDI events. While the same key of the keyboard may be played more than once throughout the course of the composition, once a note_on event appears, that key is being held down until the note_off event occurs.

So as to match these two events, for every note_on event, a new note will be created with its correspondent key and stored temporally in an array of unfinished notes. Once the note_off event arrives, a check through the array elements is done until a matching key is found. Then the new information is added to the note and is eliminated from the array where it was stored.

With the processing done until this point, each note only contains information about keyboard key, start and end time, duration and velocity. Some additional operations will be done to obtain the remaining attributes.

The note pitch and octave can be derived from the key value. As the scale used is the chromatic scale, the number of possible pitches are 12. An electronic keyboard is composed by a set of chromatic scales of consecutive octaves that are joined together. The order of the notes in the chromatic scale are: C - C♯/D♭ - D - D♯/E♭ - E - F - F♯/G♭ - G - G♯/A♭ - A - A♯/B♭ - B. These note positions are important when calculating the specific pitch of a note as they are going to match with the position of each keys in a scale set. They are stored in an array and the operation to obtain a specific note pitch is:

$$\text{NotePos} = \text{key} \;\%\; 12 \tag{7.5}$$

$$\text{Pitch} = \text{ScaleArr.get(NotePos)} \tag{7.6}$$

Where % represents the modulus operation, and ScaleArr represents an array that contains the possible pitches ordered based in the chromatic scale. The octave is obtained in a similar way. The octave count starts in 1 and each set of 12 pitches correspond to the same octave. Therefore, the operation to obtain the octave is:

$$Octave = \frac{\text{key}}{12} + 1. \tag{7.7}$$

Finally, the note figure is the last attribute left. In order to obtain its value, a detection of the beat figure of the composition is needed. At this stage of the project no time signature is processed. It will be supposed that every composition has a simple time signature so the beat figure will be a quarter note. Thus, both the duration of each note and the duration of a quarter note are known.

Then, the calculation of every note figure is done. In Figure 7.1 an example representing the duration of each note in ticks can be seen where the quarter note is the beat figure of the score and its duration in ticks is set to 100:

Note Figure	Duration in ticks
𝅝	400
𝅗𝅥.	300
𝅗𝅥	200
♩.	150
♩	100
♪.	75
♪	50
𝅘𝅥𝅯.	37.5
𝅘𝅥𝅯	25
𝅘𝅥𝅰	12.5
𝅘𝅥𝅱	6.25

Table 7.1: Note Figure Example

.

While both thirty-second and sixty-fourth figure notes could also be accompanied by a dot, their duration values are so low that they are hardly ever used. After the calculation of the duration for the known beat figures, the next step is to compare them with the duration of each parsed note in the score. The assigned note figure will be the one whose duration is closer to the duration value of the note being processed.

If the duration value of the note is greater than the duration of a whole note, then a recursive process of decomposition will be performed. This process consists in adding the highest possible note figures and subtracting their value to the duration of the note being processed until its total duration is satisfied. This procedure can be illustrated with the next example, where the the measurement of the duration will be done in beats. If the quarter note is the beat figure (duration of 1 beat), then the whole note duration is 4 beats. If a note with a 6 beat duration arrives, the decomposition process would first subtract a whole note tied to the next note figure and so on until the remaining duration reached 0. In this specific case, the process will end with a whole note tied to a half note.

After the explanation of how MIDI events work and how note figures are calculated, it has become clear the huge dependency of the system on the interpreter of the music composition. It highly depends on the accuracy playing of each note as one minimum error may lead to assigning the incorrect note figure, which might decompensate the whole score. The higher the score tempo is, the easier it is for the interpreter to deviate from the exact time and produce an error on the final product.

Once every note attribute is satisfactory obtained, the next step is to place each one of them in their correspondent pulse. To accomplish this task a comparison between the starting time of a note with the starting and end time of each pulse in the score is done. In such manner, notes are going to be placed where their starting time fits between the starting and ending time of a pulse. The note figure is not taken into account when placing notes inside pulses. Therefore, it will not matter if the note figure is greater in duration than the beat figure of the score. This might produce that some pulses may end up empty.

One example is if two half notes are played one after the other, the duration of the whole section would be 4 beats or pulses. Using the method explained, the obtained result would be a half note figure inside pulses 1 and 3, having pulses 2 and 4 empty. The reason why it does not matter is because, when writing the score, the pulses are processed sequentially. Therefore, the writing process would write a half note in pulse 1, nothing in pulse 2, another half note in pulse 3 and nothing in pulse 4, which is the expected and desirable representation.

To conclude, as a result of using the length of the sequencer to divide the score in pulses, a number of empty pulses at the last positions are going to be created. These pulses does not exist in reality as the composition is already finished. The reason why this happens is that even though the events related with the composition are finished, the sequencer sends some meta-messages that represents the end of the MIDI file so in that period of those pulses are created. Again, this is not a problem when writing the score so they are ignored.

7.2.3 Key signature

As explained in the Fundamentals and main concepts chapter, the key signature is a set of sharp (♯) or flat (♭) elements located in certain pitches indicating that every note with one of these pitches is altered by that element.

As explained above, By processing MIDI events the pitch of each note is known and, therefore, the number of natural and accidental notes for each pitch can be stored. Once all notes appearances have been stored, a criteria to decide if a certain pitch has to be present in the key signature needs to be defined. The selected criteria is that if the average of altered notes in comparison with the natural notes of a certain pitch is greater than 50% then the correspondent element has to appear in the key signature at the corresponding position.

One issue that arises is how to distinguish if an altered note is altered by a ♯ or a ♭ element. The order of appearance of the elements in the different key signatures can as a solution to this problem. Recall that the order followed by key signatures are:

- ♯ **Key Signatures:** F♯ C♯ G♯ D♯ A♯ E♯ B♯
- ♭ **Key Signatures:** B♭ E♭ A♭ D♭ G♭ C♭ F♭

The process followed to calculate the key signature is:

- First, the sharp key signatures are processed. The procedure starts checking if the average number of F♯ in comparison with natural F notes is greater than 50%. If this happens, then the key signature of the score will have at least one ♯. Afterwards, the same process is followed in order until one of the pitches does not meet the criteria. The key signature will contain the ♯ elements of all the previous pitches that met the criteria.
- In case the criteria was not met in the first iteration of the process (i.e. the average of F♯ was lower than 50%) then the key signature does not have any ♯ element. Then the same process is done for the ♭ elements. If the average number of B♭ in comparison with natural B is greater than 50% then the key signature will have at least one ♭. The process is repeated in order until one of them does not meet the criteria. The key signature will contain the ♭ elements of all the previous pitches that met the criteria.
- If neither the first ♯ element nor the first ♭ element met the criteria, then the key signature has no elements. Which means that excepting accidental elements that can appear at some point in the composition, every note in the scale is natural.

All the tasks planned for Iteration 2 have been done. To test that the functionalities implemented works correctly some simple scores will be processed, comparing the accuracy of the information obtained with the real composition. One example of a music piece used for testing is displayed in Figure 7.2 and the obtained results in Figure 7.3

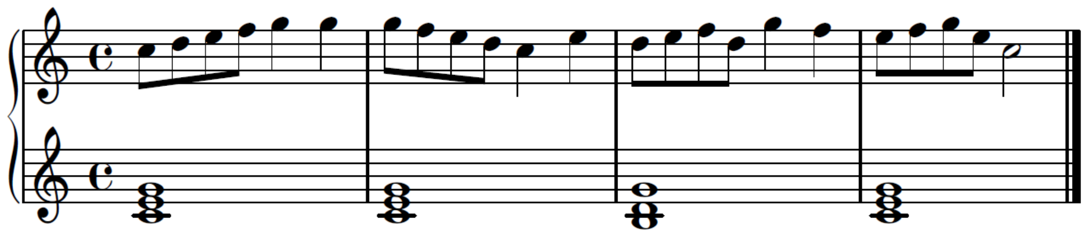
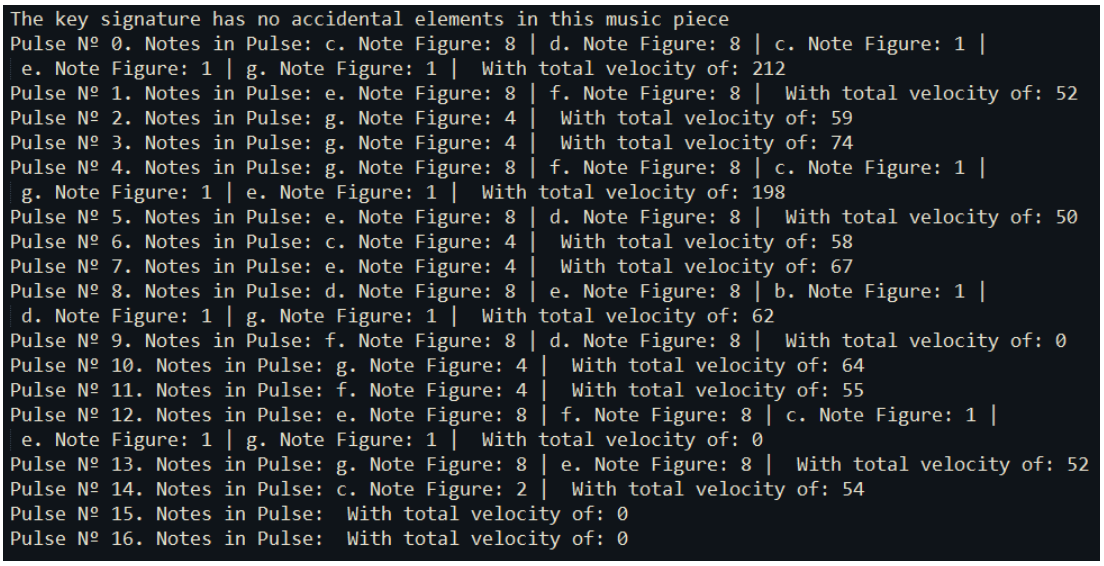


Figure 7.2: Iteration 2 testing score



```
The key signature has no accidental elements in this music piece
Pulse Nº 0. Notes in Pulse: c. Note Figure: 8 | d. Note Figure: 8 | c. Note Figure: 1 |
 e. Note Figure: 1 | g. Note Figure: 1 |  With total velocity of: 212
Pulse Nº 1. Notes in Pulse: e. Note Figure: 8 | f. Note Figure: 8 |  With total velocity of: 52
Pulse Nº 2. Notes in Pulse: g. Note Figure: 4 |  With total velocity of: 59
Pulse Nº 3. Notes in Pulse: g. Note Figure: 4 |  With total velocity of: 74
Pulse Nº 4. Notes in Pulse: g. Note Figure: 8 | f. Note Figure: 8 | c. Note Figure: 1 |
 g. Note Figure: 1 | e. Note Figure: 1 |  With total velocity of: 198
Pulse Nº 5. Notes in Pulse: e. Note Figure: 8 | d. Note Figure: 8 |  With total velocity of: 50
Pulse Nº 6. Notes in Pulse: c. Note Figure: 4 |  With total velocity of: 58
Pulse Nº 7. Notes in Pulse: e. Note Figure: 4 |  With total velocity of: 67
Pulse Nº 8. Notes in Pulse: d. Note Figure: 8 | e. Note Figure: 8 | b. Note Figure: 1 |
 d. Note Figure: 1 | g. Note Figure: 1 |  With total velocity of: 62
Pulse Nº 9. Notes in Pulse: f. Note Figure: 8 | d. Note Figure: 8 |  With total velocity of: 0
Pulse Nº 10. Notes in Pulse: g. Note Figure: 4 |  With total velocity of: 64
Pulse Nº 11. Notes in Pulse: f. Note Figure: 4 |  With total velocity of: 55
Pulse Nº 12. Notes in Pulse: e. Note Figure: 8 | f. Note Figure: 8 | c. Note Figure: 1 |
 e. Note Figure: 1 | g. Note Figure: 1 |  With total velocity of: 0
Pulse Nº 13. Notes in Pulse: g. Note Figure: 8 | e. Note Figure: 8 |  With total velocity of: 52
Pulse Nº 14. Notes in Pulse: c. Note Figure: 2 |  With total velocity of: 54
Pulse Nº 15. Notes in Pulse:  With total velocity of: 0
Pulse Nº 16. Notes in Pulse:  With total velocity of: 0
```

Figure 7.3: Results Obtained after iteration 2

.

As observed, the test has produced satisfactory results, since the system is able to correctly differentiate between whole, quarter and eight notes. Besides, every singe note is positioned in the correct pulse. Finally, the key signature was properly detected. This same results were obtained for other tests performed.

7.3 Iteration 3

In this iteration an adapter class is implemented to collect all the information obtained from the operations implemented in iteration 2 and modify it in a manner that can be used by LilyPond to generate a PDF file with the rendered score.

In order to perform this operation, one vital task is the recognition of every possible scenario in regards to the structures formed by the notes of the composition to ensure that the gathered information can be modeled using LilyPond notation. Each pulse has an array of notes ordered by start time. Notes will be processed sequentially, processing every pulse of the score in order. There are several case scenario that may happen when processing each pulse:

One note

The simplest scenario is that where one pulse contains a single note. The note obtained is written with the necessary information. In particular, LilyPond notation to describe a note in the score is the note pitch follows by a number of apostrophes (') or commas (,) that represent the octave of the note and finally its note figure. For example, a whole c note located in the fourth space of a G-clef staff (Octave 4), the representation would be: *c"1.*

Two or more notes

A pulse may contain two or more notes. The first attribute to analyze in this case is the note figure for each note stored in the pulse. The possible scenarios are:

- **Every Note in the pulse has the same duration**. If every note in the pulse has the same duration, then the start time attribute for each note hast to be checked. If every one of them starts at the same time, then all of the notes in the pulse are forming a chord as it can be seen in Figure 7.4.

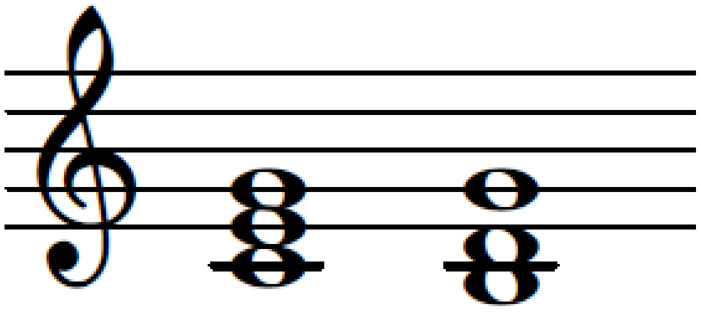


Figure 7.4: Notes with same duration and start time

The LilyPond representation for this scenario is: *<c' e' g' >1 <b d' g' >1*

When the starting times of every note in the pulse are not equal then several scenarios may happen. In these scenarios is vital to be able to detect which notes start at the same time forming chords and which notes starting time is separate from others, meaning they do not assemble any chord and have to be written as sigle notes. Some examples are:

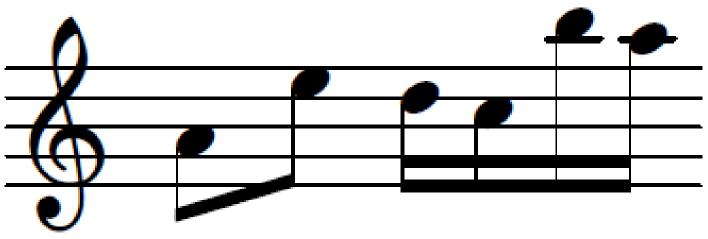


Figure 7.5: Notes with same duration and different start time 1

In Figure 7.5 two different pulses are represented. When processing both of them, the duration of each note is the same but their start times are different. The second processing of the start times obtains that notes appear in a sequential way one after another. Thus, the notes have to be written down as single notes. The LilyPond representation is: *a'8 e"8 d"16 c"16 b"16 a"16.*

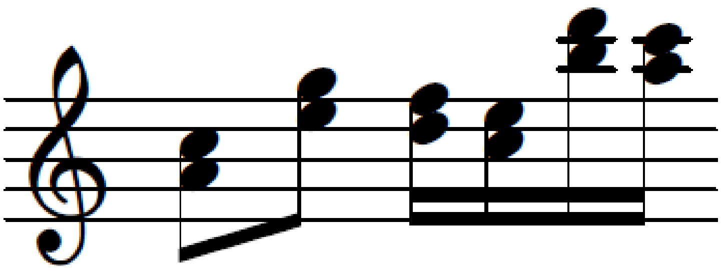


Figure 7.6: Notes with same duration and different start time 2

In Figure 7.6 another case scenario is represented. The first processing will also obtain that every note has the same note figure. Then, the second processing has to be able to detect those notes that starts at the same time and separate them in different sets to be able to write them properly in form of chords in the file. The representation of this example is: *<a' c">8 <e" g">8 <d" f">16 <c" e">16 <b" d"'>16 <a" c"'>16.* While the chords showed in the example are compounded by two notes, there is no limitation in the number of notes forming one.

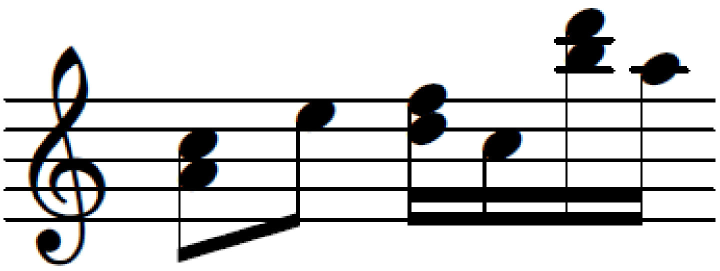


Figure 7.7: Notes with same duration and different start time 3

Another case scenario can be seen in Figure 7.7. In this example, a mix between notes assembling chords an single notes is represented. The processing is similar to the ones explained above. A organization of notes in sets depending on their starting time is done. Afterwards these sets are checked in sequential order. If the number of notes in the set is greater than one, they are forming a chords. Otherwise, they are represented as single notes. The LilyPond representation is: *<a' c">8 e"8 <d" f">16 c"16 <b" d"'>16 a"16.*

- **Notes in the pulse have different duration.** These scenarios are the reason why the number of different combinations to model in music is unattainable. Formerly, every time two notes shared the same starting times it was known that they would assemble a chord. However, in these scenarios, two notes with the same starting times might or might not have the same duration. If they do not have the same duration, then they cannot be written down as the same voice as LilyPond only allows to have a block of notes at the same time. For example if a quarter note and eight note share the same starting time they cannot be written in a LilyPond file as *c4 e8* or *e8 c4* as they will be represented one after the other. Neither they can be represented as chords, as they cannot be formed by different figure notes.

  The procedure needed to write both notes correctly is to separate them in different voices. Therefore, it would be needed to detect the number of different voices that appears in the score and process them separately. The issue is how to detect which note has to be placed in which voice. One example representing a possible scenario is exhibited in Figure 7.8

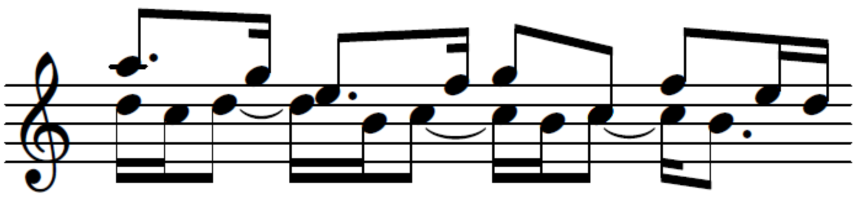
  

  Figure 7.8: Notes with different duration

  The representation of this scenario is: *« { a"8. g"16 e"8. f"16 g"8 c"8 f"8 e"16 d"16 } //{ d"16 c" d"8 d"16 b' c"8 c"16 b' c"8 c"16 b'8. } ».*

  This example is presented to clarify the complexity of these scenarios. First, the operations needed to determine the voices are very complex. Secondly, there are hundreds of possible combinations of different notes that makes impossible to be able to detect every one of them. The project will detect some of the most common an easier ones which are when one of the voices is composed by a note or group of notes where their note figure is equal or greater to the beat figure and equal between them.

7.4 Iteration 4

In this iteration the advanced version of the software is developed.

7.4.1 Beat Figure of Composition

In this iteration of the project, the calculation of the time signature is done. So as to be able to detect if the time signature of the composition is simple or compound, the beat figure has to be known. After analysing the different options, the conclusion was that there was no reference whatsoever to determine if the figure beat was a quarter note or a dotted quarter note. With the tempo of the music piece the number of TPB is acknowledged, but there cannot be assigned to any figure note. Therefore, the resolution of the analysis was that the beat figure of the score had to be asked to the user.

As a consequence, the structure of the system had to be extended. In iteration two the number of TPB was always assigned to be a quarter note and afterwards the duration of the rest of the note figures was calculated. Thus, the system was extended to slightly perform the specific operations according to the beat figure.

7.4.2 Time Signature

Calculating the time signature with the information provided by MIDI is complex. The only useful information for time signature detection is the velocity of the score notes. As these notes are placed in their corresponding pulses, the total velocity of the pulse can be calculated.

Musical theory explains that each time signature follows a pattern of strong and weak pulses. The patterns followed by the different time signatures is represented in Figures 7.9, 7.10 and 7.10:

- **2/4 and 6/8:**

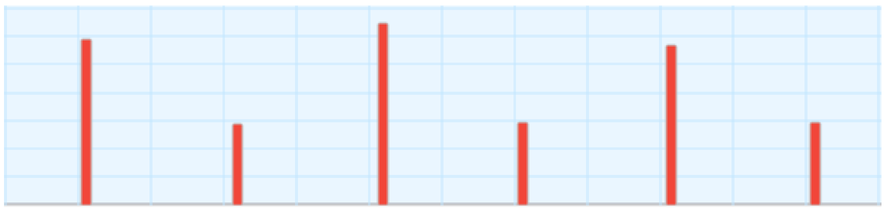


Figure 7.9: Pattern followed by binary time signatures

- **3/4 and 9/8:**

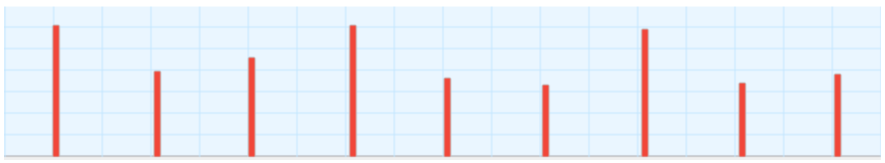


Figure 7.10: Pattern followed by ternary time signatures

- **4/4 and 12/8:**

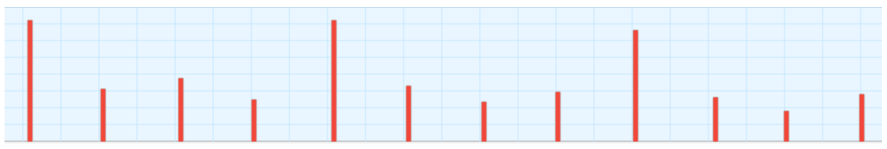


Figure 7.11: Pattern followed by quaternary time signatures

Therefore, performing a convolution through the pulses of the score comparing their total velocity will clarify to which pattern the composition resembles the most. It has to be noted that the total velocity of a pulse will not be the calculation of the velocities of every note. The pattern above exposed represents the velocity of the starting point for the pulse as this is the defining moment for each pulse. Thus, only the velocity of notes whose start time matches the pulse start time will be considered. Along these lines, if a pulse is contains two sequential eight notes one after another, only the velocity of the first eight note (the one that starts at the beginning of the pulse) will count.

Three kernels will be used in the convolution process to determine the time signature. The three kernels are:

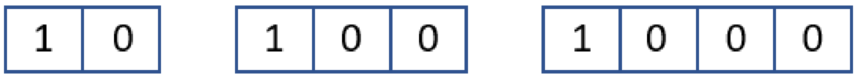


Figure 7.12: Convolution kernels

Each kernel will iterate through the array of pulses moving in each iteration as many positions as its size value. For example, the first element of the binary kernel would iterate through positions 0, 2, 4, 6 ... The ternary kernel through positions 0, 3, 6, 9 ... And the quaternary kernel through 0, 4, 8, 12 etc.

The convolution operation will consist in the sum of every product between the kernel values and the velocity values of the respective pulses divided by the number of iterations or operations performed.

The kernel that obtained the higher value is the one that most resembles the score velocities and specifies its time signature. The time signature obtained in respect with higher kernel value and the beat figure is shown in Table 7.2

Higher value kernel	♩ Beat Figure	♩. Beat Figure
Binary	2/4	6/8
Ternary	3/4	9/8
Quaternary	4/4	12/8

Table 7.2: Time signatures in respect to kernels and beat figure

After the analysis on the operations performed to calculate the time signature, again, the conclusion is that the success of these operations depend highly on the interpretation of the composition done by the user. While theoretically the pattern followed in such compositions is one of the above explained, some times it does not happen inn reality because the interpreter does not play the music nuances properly or the composition velocity structure does not match any of the pattern above explained in particular.

This pattern becomes more precise in large compositions as there are more samples of its structure. In short compositions, if this patterns are not followed as expected the final solution may be wrong. One example can be seen in Figure 7.13 where in both the second and fourth operations of the quaternary convolution process the outcome of the product is 0.

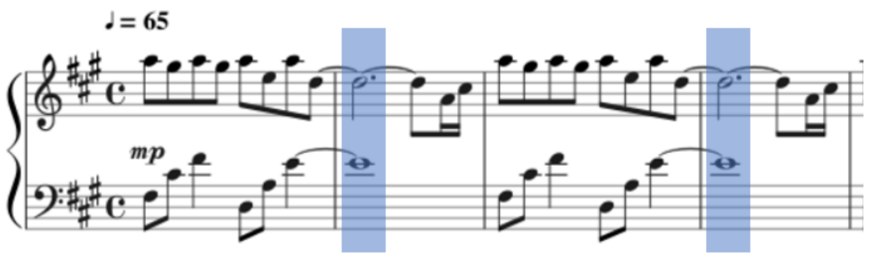


Figure 7.13: Convolution example

7.4.3 Hands separation

The hand separation process comprises various checking operations to decide whether a note should be played by the left or right hand. Until this point, every pulse contains notes that are put together. For this process, two variables representing the octave notes of each hand will be used. When the process starts, the notes inside the pulses of the first bar of the score are checked to set an initial value to the left and right hand octaves. Once their value is set, every pulse of the composition is checked.

There are two main scenarios that can occur in a music composition. The first main scenario is when the octave variables values for each hand are separated by more than 1 octave. In this first scenario, notes that have either the same or higher octave than the right hand expected octave or they have lower or same octave value than the left hand expected octave are placed directly in their corresponding locations.

Also, it may happen that notes appear in the octave between the right and left hand expected octaves. In these cases, some calculations are done in order to detect where to locate them. The operation done is to calculate the absolute difference between the current note key and the key of the rest of the notes in the pulse. Hence, the maximum differences with the highest key and lowest key of the pulse are calculated. After this operations different operations can occur:

- There is only one note in the pulse. There is no possible way to determine if the note should be played by the left or right hand. The decision of the author was to suppose that the played note would be in the right hand.
- There are lower key notes within the pulse but not higher key notes. If there are lower key notes but there are not higher that means that the current note has to be assigned to the right hand.
- There are higher key notes within the pulse but not lower key notes. This scenario means that there are higher key notes being played in the right hand so the current note is assigned to the left had.
- There are both higher and lower key notes within the pulse. In this scenario, the differences calculated in the previous step are compared between them. If the difference is lower with the higher note than with the lower note, then the more probable scenario is that the current note has to be played by the right hand. Otherwise, the more probable scenario is that the current note has to be played by the left hand.

  It can also happen that the value obtained is equal for both differences. Then again, there is no possible way of determining which is the correct decision. As the left hand usually play chords and arpeggios which derives higher jumps between notes, the author will suppose that these notes are going to be played by the left hand.

The second main scenario occurs when the expected right hand and left hand octaves are contiguous values. In this case, notes with higher octave that the right hand octave expected as well as notes with lower octave that the left hand octave will be placed directly in their corresponding locations. The rest of the notes will have to processed calculating the absolute difference as above explained.

7.4.4 Clefs

Once the notes are separated in both hands, the calculus of the clef for each hand is easy. First, the right hand clef for piano is always the G-Clef so it is already assigned. For the right hand clef, the process comprised checking the number of notes located in each octave. Afterwards, if the majority of notes were in octave 3 or lower, the F-Clef was selected, otherwise, the G-Clef is selected.

7.4.5 Ties

The ties that will be calculated are the union ties. The expression ties provides information on how the interpreter has to play specific notes that are not useful for the project development.

The union ties will be calculated in order to maintain the expected length of each bar of the score. This operation is vital as an error in the length of a specific bar may cause the rest of the score to be wrong.

This operation is going to be applied to every note of the music piece, the procedure to follow is:

- Get the starting time of the note.
- Calculate the maximum note figure that fits within the remaining duration of the bar (end time of the last pulse of the bar minus note start time).
- Compare the maximum note figure calculated with the note figure of current note. If the maximum note figure is greater or equal than the current note figure, then the iteration ends and the process is started again with the next note.
- If the maximum note figure is less than the current note figure, this implies that a tie has been found in that bar, and therefore, a decomposition process of the current note figure is done.
  - The maximum note figure is subtracted from the current note figure to calculate the note figure of the remaining duration.
  - Place the maximum note figure in the pulse where the previous note was placed. Place the note figure created with the remaining duration in the first pulse of the next bar and join them with a tie. A representation of this process can be seen in Figure 7.14.

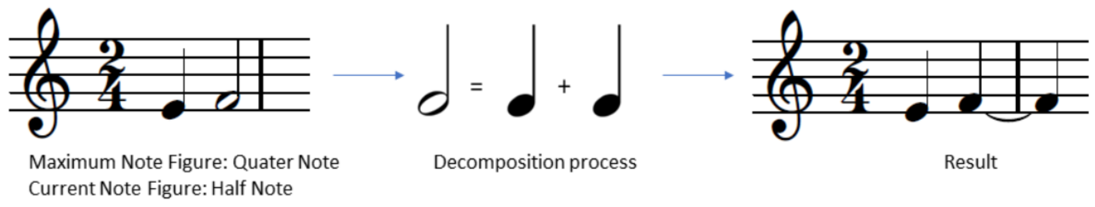

Maximum Note Figure: Quater Note
Current Note Figure: Half Note
Decomposition process
=
+
Result

Figure 7.14: Example of the required operation for a tie.

7.4.6 LilyPond adapter extension

The extension to this class was easily made because the upper and lower staffs written separately one after the other. Then, the same process that was implemented in the previous iteration is done twice. One for the left-hand notes and the other for the right hand notes.

7.4.7 Testing

All the functionality of the application is done at this point. Then, some test are done to check if the implementation is done correctly. The musical pieces used for this testing were taken from the book *Practical Method for Beginners on the Pianoforte* from Carl Czerny [34]. The ones selected can be seen in Figures 7.15 and 7.16. This two test will clarify that the time signature, hand separation, clefs and the adaptation to LilyPond notation is performed correctly.

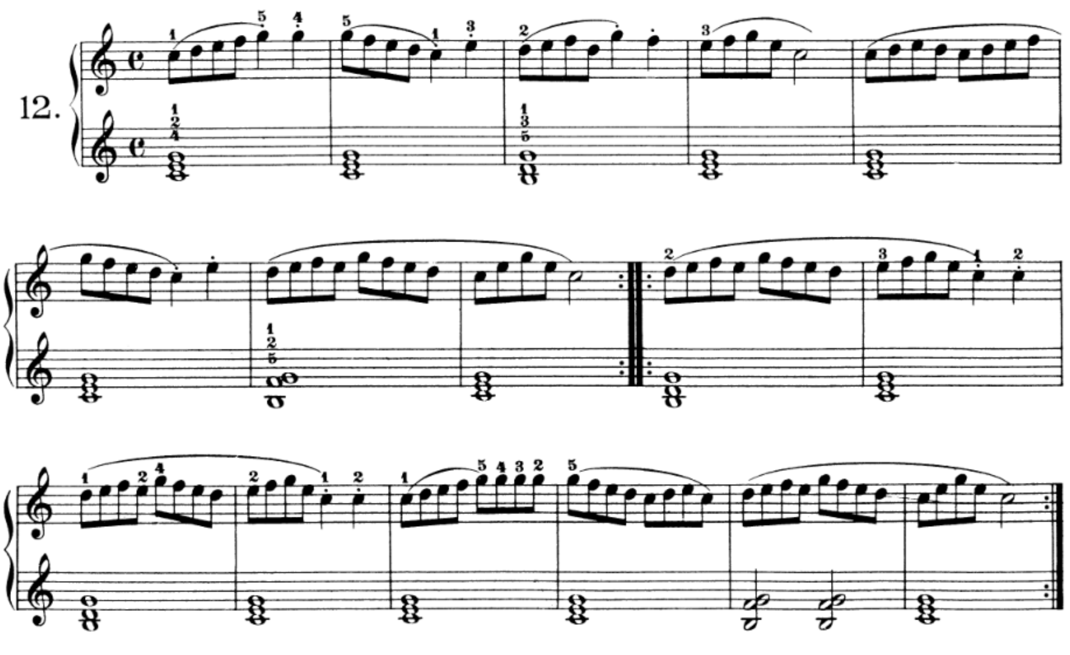
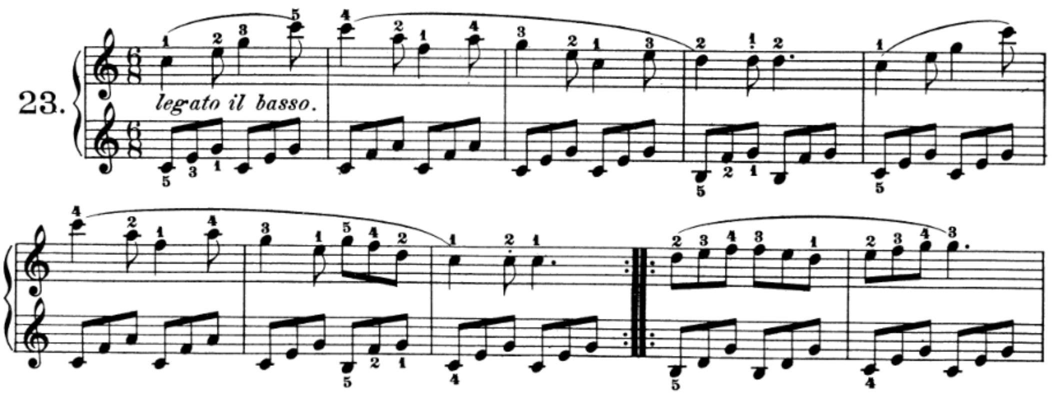


Figure 7.15: Test 1



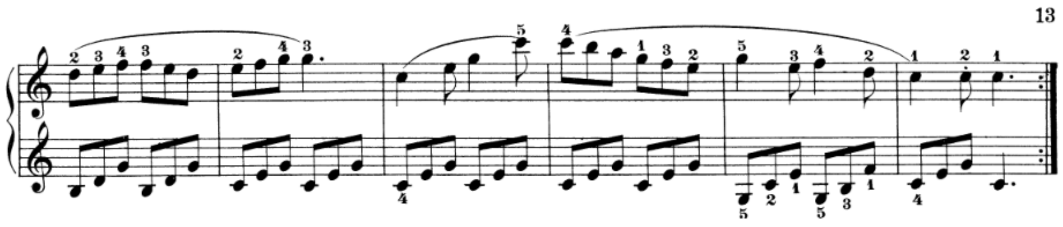


Figure 7.16: Test 2

The result obtained are displayed in Figures 7.17 and 7.18.

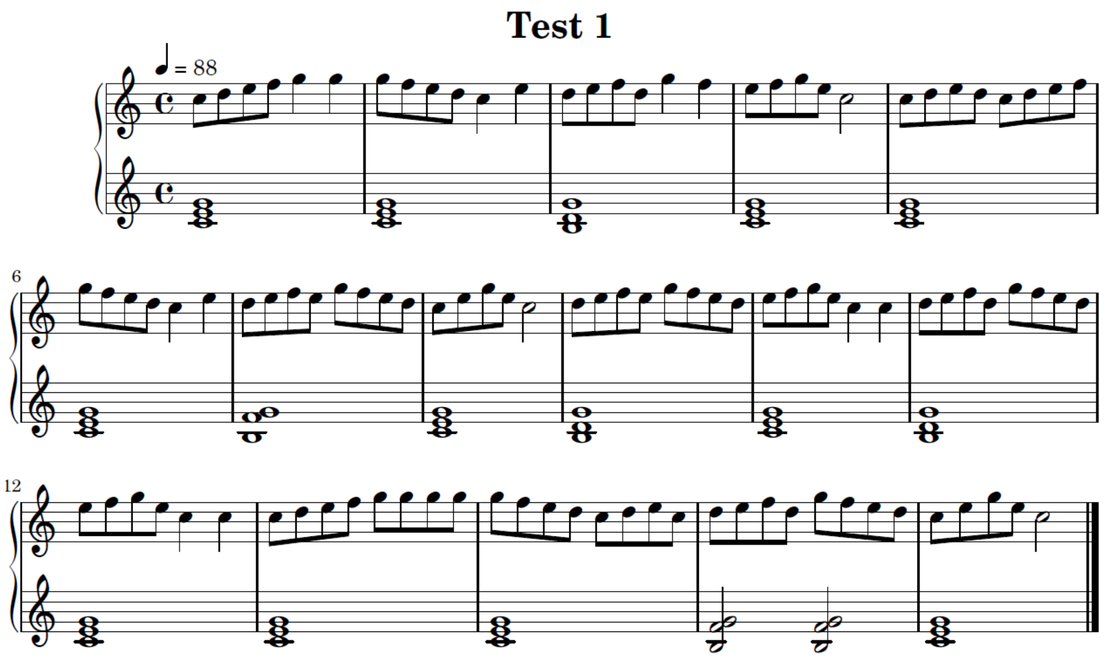


Figure 7.17: Test 1 Result

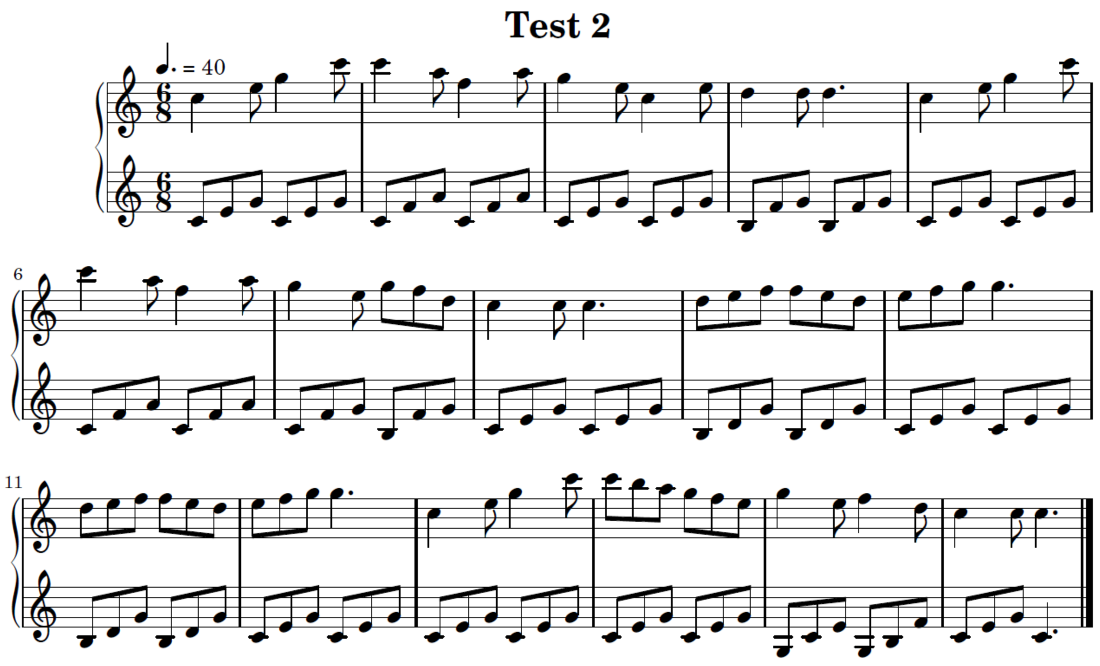


Figure 7.18: Test 2 Result

It can be observed that the original scores contain repeating bar lines while the ones created by the system do not. This is because those bar lines are not supported by the program and instead of repeating those parts, the composition was played just one from beginning to end.

It can be observed that the result obtained are quite satisfactory. The system is able to produce really good scores despite of the complexity of music pieces and large amount of different combinations existent for each one of the functionalities implemented in this iteration.

7.5 Iteration 5

In this iteration the UI was created. The UI was implemented to be simple and resemble as much as possible to the prototype. The UI functionalities are:

- Load a file to process.
- Restart UI screen.
- Execute program.
- Save score as PDF..
- Start player.

- Pause player.
- Stop player.

The UI representation is:

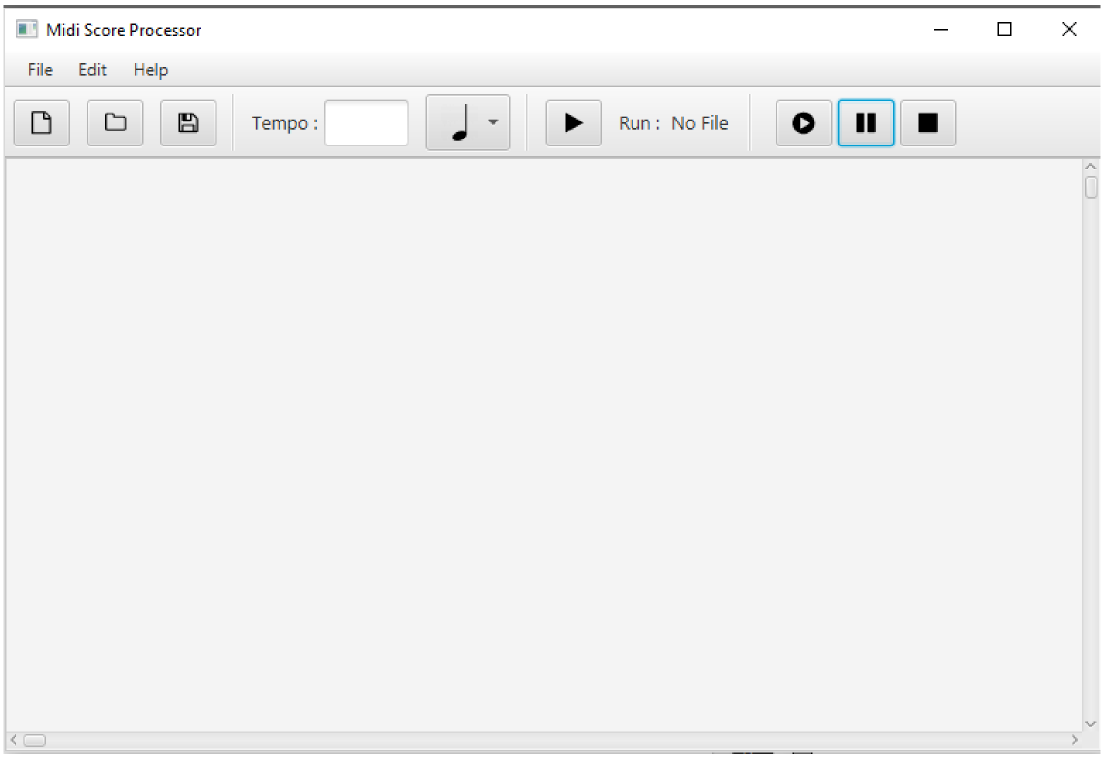
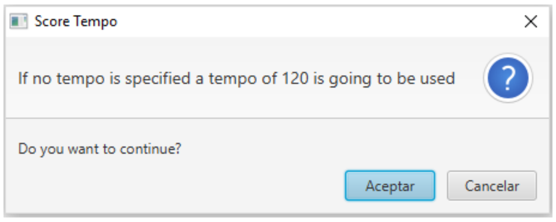


Figure 7.19: UI representation

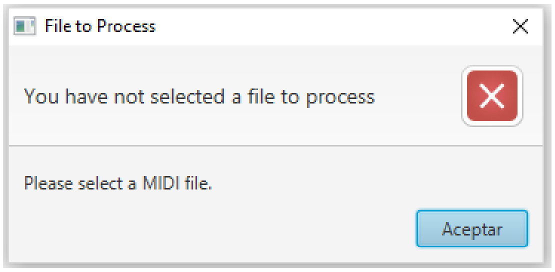
Besides, some alerts have been created in order to respond to user errors. In Figure 7.19 an attention alert is displayed reporting the user that if no tempo is specified the system will process the file with a tempo of 120 BPM. In Figure 7.20 an error alert is displayed reporting the user that no MIDI file was selected. Finally, in Figure 7.21 an error alert is displayed reporting the user the the file selected is not a valid file.



If no tempo is specified a tempo of 120 is going to be used

Do you want to continue?

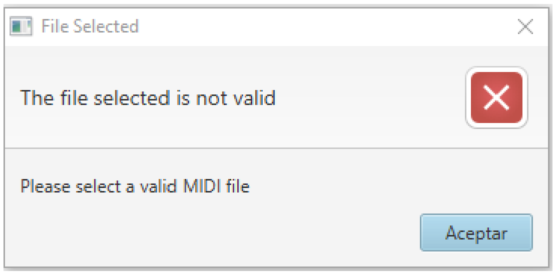
Figure 7.20: UI attention alert


File to Process

You have not selected a file to process

Please select a MIDI file.

Figure 7.21: UI error alert 1


File Selected

The file selected is not valid

Please select a valid MIDI file

Figure 7.22: UI error alert 2

Chapter 8

Conclusions

In this chapter an analysis, evaluation and conclusions of the work carried out in the development of this project will be done.

The final product created after the developing process meets the objectives of the project. The system is flexible in terms of the ability to process MIDI files with little information and, therefore, expanding the number of potential users that use open software applications which record MIDI files with less features than the commercial ones. Besides, the system is able to calculate many features of a score programatically, asking the user only the minimal necessary information in order to have a reference from which the system can start working with.

The results obtained in the test performed are quite satisfactory. The software was able to create scores that look very similar to the real ones despite the difficulty to address the calculus of certain critical parameters with little information to produce a correct score and the huge casuistry of musical compositions.

If compared with the other available software (Midi sheet music and MuseScore), the perspective of the results obtained is even better. First, the software created is able to process way more MIDI files just for the fact that it can detect compound time signatures while the others do not. Moreover, the hands separation is better making the score look much more clear and professional.

On the other hand, the results obtained by this software depends largely on the performance of the user when recording the MIDI file. MIDI treats music notes as mathematical numbers, so the interpreter of the music piece has to be extremely accurate when playing each one of them, as playing a specific note a little bit shorter or longer may cause the final length approximates more to an undesired figure note. This same problem happens when calculating the time signature, if the specified patterns are not being played accurately, then the operation might end up having a wrong result. accurately.

Furthermore, the developed software can be used to represent from basic to intermediate music pieces, but will probably fail when trying to process more advanced compositions.

There are two main reasons why this happens. The first one is the complexity of music. As analyzed through the different iterations of the project, the number of possible combinations that can appear in a composition at one moment is immensely large, which makes the task of modelling them nearly impossible. The second one is the low amount of information provided by MIDI.

MIDI was designed since its inception as a protocol used for sound applications. Since its launch in 1984, MIDI remained in the same version until this year. Throughout this time, many functionalities have been added to its specification but mostly everyone of them oriented to sound applications which have made big progress in recent years. Thus, the quantity of information for music notation was non-existent. One reason may be again the complexity of representing music in a score.

One example is the event that MIDI sends when the keyboard pedal is used. Sound applications do not need that information, they just hear a pitch starting at a specific time and ending at another. Then, they can perform the desired operations to modify the sound. However, this event for music notation cannot be considered as events provided by MIDI are: pedal pressed at a specific time in ticks, pedal undressed at another specific time. In many cases, the pedal is pressed to melodically unify some notes that appear sequentially so some of the different pitches are mixed. Yet, some times might be used to maintain a sound without the need of pressing the key or both of them. Therefore, there is no possible way of using those events, which leads the system to only use the information of note_on and note_off events. This is another reason why more advanced and complex composition cannot be processed successfully by the system as in these compositions the interpreter uses the pedal in many occasions to be able to free the hands and position them for the following notes. This cannot be done if the record is expected to be processed by the software, the hands must maintain the keys pressed until the end.

As it can be observed, the dependency on the performance of the user when recording MIDI files is in big part as a consequence of the MIDI protocol limitations. However, this accuracy needed to obtain a satisfactory result is useful in terms of the learning process. This will make users play the different compositions using a metronome, learning to play songs better and with more precision, which was one of the motivations of this project.

To summarize, the objectives of the project have been satisfactorily accomplished. The final product is an useful software that can process different types of compositions. During the development of this project it has been acquired knowledge on music and how is processed by computers, new tools and technologies and deeper understanding of analysis and design techniques. To conclude, one of the most useful skills the author has learnt during the development of this project is the capacity to solve different problems effectively.

Chapter 9

Future work

During the development of the project a wide range of different topics have been addressed. Some possible future lines of work are:

- Add visual elements and functionalities to the UI. The UI implemented is simple, easy to use and satisfies the objectives of the project. However, some future enhancements could be made to improve user experience such as coloring the notes of the score as the music piece is being reproduced by the player. Another possible addition would be a keyboard element whose keys are colored following the reproduction of the composition as well.

- Introduce machine learning techniques into the system to try detecting elements that cannot be detected otherwise. One example may be detecting repetition bars by recognizing sections that are repeated in the composition one after another. Another example might be detecting the musical phrasing of the composition to represent some expression ties.

- Study and analyze the specification for MIDI 2.0. The MIDI 2.0 standard was presented on 17 January 2020 at the Winter NAMM Show in California. A study of the new functionalities implemented in this version can be done to determine if they would could be used for improving some features of the system or extending its capabilities.

Appendices

List of Acronyms

MIDI	Musical Instrument Digital Interface.
UI	User Interface
IFF	Interchange File Format
ASCII	American Standard Code for Information Interchange
PNG	Portable Network Graphics
PDF	Portable Document Format
WYSIWYG	What You See Is What You Get
OO	Object Oriented
API	Application Programming Interface
SVG	Scalable Vector Graphics
WYSIWYM	What You See Is What You Mean
BPM	Beats Per Minute
MVC	Model View Controller
XML	Extensible Markup Language
CSS	Cascading Style Sheets
IDE	Integrated Development Environment
SRS	Software Requirements Specification
TPB	Ticks Per Beat

Bibliography

[1] G. Galindo, “The Importance of Music in Our Society.” [Online]. Available: http://www.borrowednotes.org/the-importance-of-music-in-our-society/

[2] F. W. Nietzsche, *Twilight of the idols, or, How to philosophize with the hammer.* Indianapolis, Ind: Hackett Pub, 1997.

[3] ““Music gives soul to the universe and wings to the mind”.” [Online]. Available: https://www.wvi.org/article/%E2%80%9Cmusic-gives-soul-universe-and-wings-mind%E2%80%9D

[4] A. Ros-Morente, S. Oriola-Requena, J. Gustems-Carnicer, and G. Filella Guiu, “Beyond music: Emotional skills and its development in young adults in choirs and bands,” *International Journal of Music Education*, vol. 37, no. 4, pp. 536–546, Nov. 2019, wOS:000494687700004.

[5] Ifpi, “Global music report 2019,” 2019. [Online]. Available: https://www.cudisco.org/pdf/GLOBAL-MUSIC%20REPORT-2019.pdf

[6] M. M. Association, *The Complete MIDI 1.0 Detailed Specification.* The MIDI Manufacturers Association, 1996.

[7] “Musical Terms | Music Glossary: Terminology.” [Online]. Available: https://www.naxos.com/education/glossary.asp?char=S-U#

[8] A. Klapuri and M. Davy, *Signal Processing Methods for Music Transcription.* Springer Science & Business Media, Feb. 2007.

[9] “Why does the chromatic scale have both sharps and flats? - Quora.” [Online]. Available: https://www.quora.com/Why-does-the-chromatic-scale-have-both-sharps-and-flats

[10] D. M. Randel, *The Harvard Dictionary of Music: Fourth Edition*, edición: 4 new edition ed. Cambridge, Mass: Harvard University Press, Dec. 2003.

[11] M. Music, "An explanation of clefs: treble, bass, alto, tenor," Nov. 2013. [Online]. Available: https://makingmusicmag.com/explanation-clefs-treble-bass-alto-tenor/

[12] "A Complete Guide to Time Signatures in Music," Mar. 2019. [Online]. Available: https://www.musicnotes.com/now/tips/a-complete-guide-to-time-signatures-in-music/

[13] "Music key signatures explained." [Online]. Available: https://www.piano-keyboard-guide.com/key-signatures.html

[14] "basicmusictheory.com: E major key signature." [Online]. Available: https://www.basicmusictheory.com/e-major-key-signature

[15] E. Selfridge-Field and C. P. M. a. S. S. E. Selfridge-Field, *Beyond MIDI: The Handbook of Musical Codes.* MIT Press, 1997, google-Books-ID: Xm3J9DG9EFcC.

[16] M. Vaidyanathan, "Midi Sheet Music." [Online]. Available: http://midisheetmusic.com/features.html

[17] "musescore/MuseScore." [Online]. Available: https://github.com/musescore/MuseScore

[18] J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley, and D. Smith, "The Java® Language Specification," p. 774.

[19] "Java Sound Programmer Guide." [Online]. Available: https://docs.oracle.com/javase/7/docs/technotes/guides/sound/programmer_guide/contents.html

[20] "Google Code Archive - Long-term storage for Google Code Project Hosting." [Online]. Available: https://code.google.com/archive/p/abc4j/

[21] "Writing Music in Java: Two Approaches." [Online]. Available: https://objectcomputing.com/resources/publications/sett/january-2008-writing-music-in-java-two-approaches

[22] "Guía del usuario de JFugue | Acorde (Música) | Java (lenguaje de programación)." [Online]. Available: https://es.scribd.com/document/110800461/Guia-del-usuario-de-JFugue

[23] "LilyPond - Debian Wiki." [Online]. Available: https://wiki.debian.org/LilyPond

[24] "LilyPond Contributor's Guide: 10.1 Overview of LilyPond architecture." [Online]. Available: http://lilypond.org/doc/v2.18/Documentation/contributor/overview-of-lilypond-architecture

[25] “Ensayo sobre grabado musical automatizado: LilyPond — Ensayo sobre grabado musical automatizado.” [Online]. Available: http://lilypond.org/doc/v2.18/Documentation/essay/

[26] “MidiEditor.” [Online]. Available: https://www.midieditor.org/

[27] “1 JavaFX Overview (Release 8).” [Online]. Available: https://docs.oracle.com/javase/8/javafx/get-started-tutorial/jfx-overview.htm

[28] Oracle, “JavaFX Scene Builder Information.” [Online]. Available: https://www.oracle.com/technetwork/java/javase/downloads/javafxscenebuilder-info-2157684.html

[29] “Apache PDFBox | A Java PDF Library.” [Online]. Available: https://pdfbox.apache.org/

[30] “Features - IntelliJ IDEA.” [Online]. Available: https://www.jetbrains.com/idea/features/

[31] V. Farcic, “Software Development Models: Iterative and Incremental Development,” Jan. 2014. [Online]. Available: https://technologyconversations.com/2014/01/21/software-development-models-iterative-and-incremental-development/

[32] ——, “Software Development Models: Iterative and Incremental Development,” Jan. 2014. [Online]. Available: https://technologyconversations.com/2014/01/21/software-development-models-iterative-and-incremental-development/

[33] “Guia Salarial Sector TI Galicia 2015-2016 | Plataformas de computación | Desarrollo de software.” [Online]. Available: https://es.scribd.com/document/288511179/Guia-Salarial-Sector-TI-Galicia-2015-2016

[34] “Practical Exercises for Beginners, Op.599 (Czerny, Carl) - IMSLP.” [Online]. Available: https://imslp.org/wiki/Practical_Exercises_for_Beginners%2C_Op.599_(Czerny%2C_Carl)