LEEDS BECKETT UNIVERSITY
MSC AUDIO ENGINEERING
Film, Music and Performing Arts Department
James Graham Building - Headingley Campus
Church Wood Ave, Leeds LS6 3QS

MIDI Step Sequencer in JUCE

Documentation, Journal and Evaluation

Silvio Gregorini

ID: 77203842

Module: COMP701 - Audio Software Engineering - 35642

Assignment 2 – Journal

Summary

1. INTRODUCTION 1

2. ESSENTIAL MODEL 1

2.1 Structure Model 1

2.1.1 Requirements Definition 1

2.1.2 Requirements Specification 1

2.1.3 FAST Analysis/Mini-Specifications 2

2.1.4 Use Case Model 3

2.1.5 Use Case Diagrams 3

2.1.6 Use Case Specifications 4

2.1.7 Class Diagram 6

2.1.8 Class Narratives 7

2.2 Behavioral Model 9

2.2.1 Sequence Diagrams 9

2.2.2 State Machine Diagram 12

2.2.3 Control Narrative 12

3. IMPLEMENTATION MODEL 14

3.1 Software Architecture 14

3.1.1 UI Specification 14

3.1.2 I/O Specification 14

3.1.3 Architecture Context Diagram 14

3.2 Software Structure 15

3.2.1 MainComponent.cpp 15

3.2.2 StepControl.cpp and NoteControl.cpp 16

3.2.3 TogglePad.h 17

4. JOURNAL 18

4.1 Concept 18

4.2 Design and Documentation 20

4.3 Implementation 22

5. CONCLUSIONS AND EVALUATION 23

6. REFERENCES 24

1. INTRODUCTION

This paper summarise the work undertaken during the Audio Software Engineering module. The first sections (2 and 3) consist of the documentation and the design diagrams that led to the implementation of the software. These are written in accordance with the Unified Modelling Language (UML), integrated with the concepts and methods learnt during the module. The software was coded in C++ through the JUCE Application Programming Interface (API), as it can be seen in the last part of section 3.

Then, a concise journal is presented to detail the work carried out during the module and to discuss problems and outcomes of the process. In the end, in section 5 the whole project is evaluated, focusing on formal testing of the software, debugging.

2. ESSENTIAL MODEL

2.1 Structure Model

2.1.1 Requirements Definition

A multitrack, MIDI step sequencer system is to be built. It will enable the user to create melodic patterns and it will output a set of MIDI commands to trigger software or hardware synthesizers. The pattern can be played back in real-time or stored in the computer.

2.1.2 Requirements Specification

The system must show to the user a grid made of 4 tracks and 16 steps: each track/row indicates a different note or sample, while each step/column indicates the position in the pattern. The user must be able to enter commands to choose a note for each track and to activate a pad in the grid, increase or decrease its velocity. The system must show on a visual display whether a note is active or not through a status indicator. The user must be able to enter commands to change the tempo (showed by the system on the display) up to 400 BPMs. The user must be able to enter commands to play, pause and stop the pattern. The user must be able to enter commands to export the current pattern in the form of a midi file. When the user enters the command to play the pattern, the system must send a set of MIDI signals which will be received by a MIDI instrument plugged in the computer. The software must run as a standalone application on Windows platforms.

2.1.3 FAST Analysis/Mini-Specifications

Objects: system, user, GRID, pad, track, note, step, velocity, VISUAL_DISPLAY, STATUS_INDICATOR, COMMAND, tempo, STORED_PATTERN, MIDI_OUT

Object	Description
system	The step sequencer itself
user	User of the step sequencer, enters COMMAND and views VISUAL_DISPLAY
GRID	Main part of the interface, consists of 4 tracks and 16 steps. Viewed by the user on the VISUAL_DISPLAY
pad	Defined by track and step, the user enters COMMAND to activate it. The system shows whether it's active/inactive on the VISUAL_DISPLAY through a a STATUS_INDICATOR
track	Row of the grid, corresponds to a note
note	Chosen by the user, define the pitch of an entire track
step	Equal time-intervals that slice a track, sets the max duration for a step
velocity	Volume level of each step, controlled by the user through a COMMAND
VISUAL_DISPLAY	Viewed by the user, displays GRID and STATUS_INDICATOR
STATUS_INDICATOR	Indicates on the VISUAL_DISPLAY wheter the pad is active or inactive.
COMMAND	User inputs: PAD_ON|PAD_OFF VALUE_UP|VALUE_DOWN PLAY_PATTERN|PAUSE_PATTERN|STOP_PATTERN EXPORT_PATTERN|IMPORT_PATTERN
tempo	MIDI beat clock of the software. The user changes it and views the BPMs on VISUAL_DISPLAY
STORED_PATTERN	Text file that the system outputs when the user enters a save COMMAND
MIDI_OUT	MIDI signals that the system output to trigger an external device

Table 1 - List of the objects highlighted during the FAST Analysis with a short description.

Operations: enter, show, activate, deactivate, increase, decrease, play, pause, stop, export, send

Operations	Description
enter	Operation through which the user inputs COMMAND in the system
show	The system outputs GRID and STATUS_INDICATOR through the VISUAL_DISPLAY
activate	Operation to ENABLE a pad in the GRID
deactivate	Operation to DISABLE a pad in the GRID
increase	Operation to INCREASE a numeric value (velocity, duration, tempo, note)
decrease	Operation to DECREASE a numeric value (velocity, duration, tempo, note)
play	Operation to PLAY the pattern
pause	Operation to PAUSE the pattern playback
stop	Operation to STOP the pattern playback
export	Operation to EXPORT a text file that contains all the MIDI commandsthat describe the pattern
send	Process performed by the system to OUTPUT a series of MIDI_OUT signals to trigger external devices

Table 2- List of the operations highlighted during the FAST Analysis with a short description.

Constraints: standalone app for Windows platforms

Performance criteria: 4 tracks, 16 steps, tempo up to 400 BPMs

2.1.4 Use Case Model

Actors

- *User* – Person who execute the program, enters commands, see the visual display.
- *Midi Device* – External device connected to the computer which receives midi messages.
- *Storage* – Internal hard drive of the computer or external storage device (external HDD, USB flash drive…)

Use Cases

- *Live Performance*
- *Production*

2.1.5 Use Case Diagrams

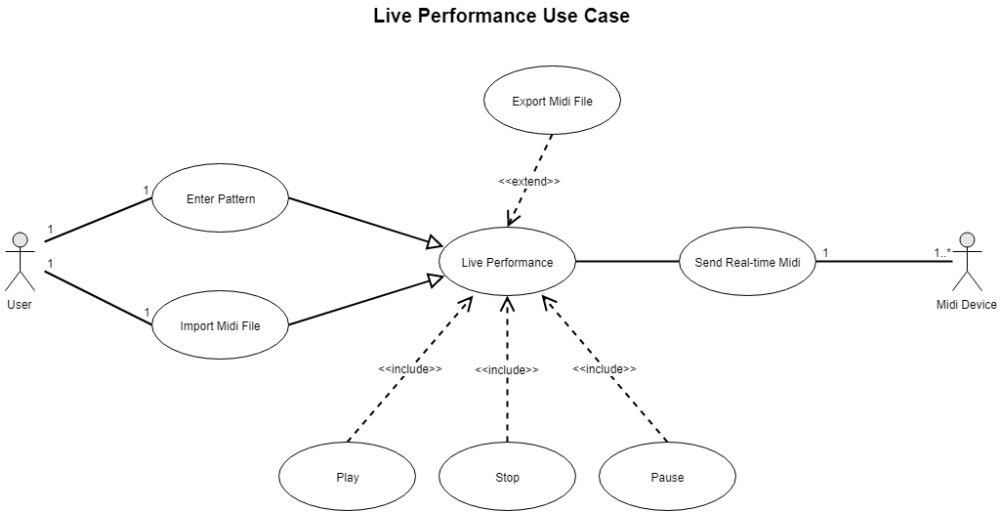

Figure 1 - Diagram showing the "Live Performance" use case.

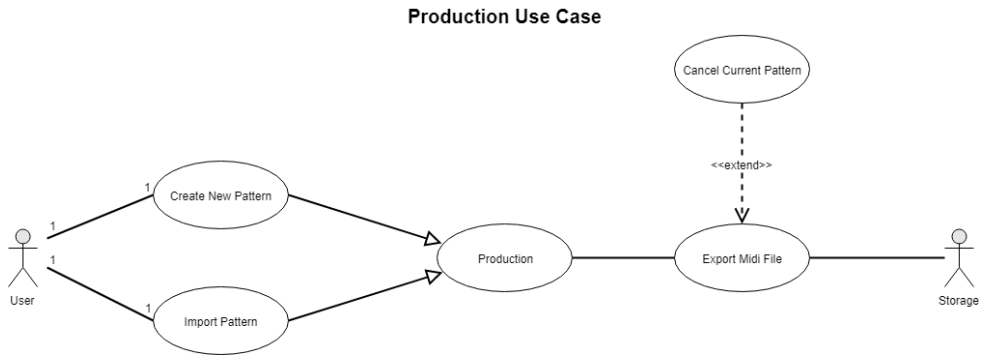

Figure 2 - Diagram showing the "Produciton" use case.

2.1.6 Use Case Specifications

Use Case Name:	Live Performance
Actor(s):	User, Midi Device
Summary Description:	The user runs the software in a live context to send real-time midi messages to external midi devices.
Priority:	Must Have
Status:	Medium Level of Details
Pre-condition:	Midi device connected to the system
Post-condition(s):	Midi device receives midi messages in real-time from the system
Basic Path:	1. User clicks on pads to enable them. 2. Grid highlights selected pads. 3. User drags rotary slider to set velocity. 4. Grid moves rotary slider. 5. StepControl change velocity value according to the rotary slider. 6. User clicks on combo box. 7. Grid opens noteList. 8. User selects one note from the list. 9. Grid shows the selected note. 10. TrackControl sets the note value according to the selection. 11. User drags tempo linear slider. 12. Grid moves linear slider. 13. PlaybackControl sets tempo value according to the linear slider. 14. User clicks on combo box. 15. Grid opens deviceList. 16. User selects one device from the list. 17. Grid shows the selected device. 18. MidiOutputManager opens the selected device. 19. User clicks on play button. 20. Grid highlights play button 21. PlaybackControl starts Timer 22. MidiOutputManager stores pad status, step velocity and track notes for step 1. 23. Grid highlights step 1. 24. MidiOutputManager sends immediately all stored messages. 25. Program repeats actions 22-24 increasing the step number from 1 to 16 and then starting again from 1. 26. If the User clicks on pause button, the loop stops but the step number is stored. 27. If the User clicks on stop button, the loop stops and the step number is reset.
Alternative Paths:	16a. No midi device connected, deviceList empty. 19a. User clicks on save button, use case “Production”. 20a. No midi device selected, program breacks.

Use Case Name:	Production
Actor(s):	User, Storage
Summary Description:	The user runs the software to create a pattern and to export it in the form of a midi file.
Priority:	Must Have
Status:	Medium Level of Details
Pre-condition:	None
Post-condition(s):	Midi file exported and stored
Basic Path:	1. User clicks on pads to enable them. 2. Grid highlights selected pads. 3. User drags rotary slider to set velocity. 4. Grid moves rotary slider. 5. StepControl change velocity value according to the rotary slider. 6. User clicks on combo box. 7. Grid opens noteList. 8. User selects one note from the list. 9. Grid shows the selected note. 10. TrackControl sets the note value according to the selection. 11. User drags tempo linear slider. 12. Grid moves linear slider. 13. PlaybackControl sets tempo value according to the linear slider. 14. User clicks on save button. 15. MidiExportManager opens file browser. 16. User selects path. 17. MidiExportManager sets the path and creates a midi file. 18. MidiExportManager gets notes from track 1-4. 19. MidiExportManager gets pad status for step 1. 20. MidiExportManager gets velocity for step 1. 21. MidiExportManager add messages to a midi buffer. 22. Program repeats actions 17-21 for steps 2 to 16. 23. Midi buffer is written on the midi file.
Alternative Paths:	14a. User plays the pattern. 14b. User pauses the pattern. 14c.User stops the pattern.

2.1.7 Class Diagram

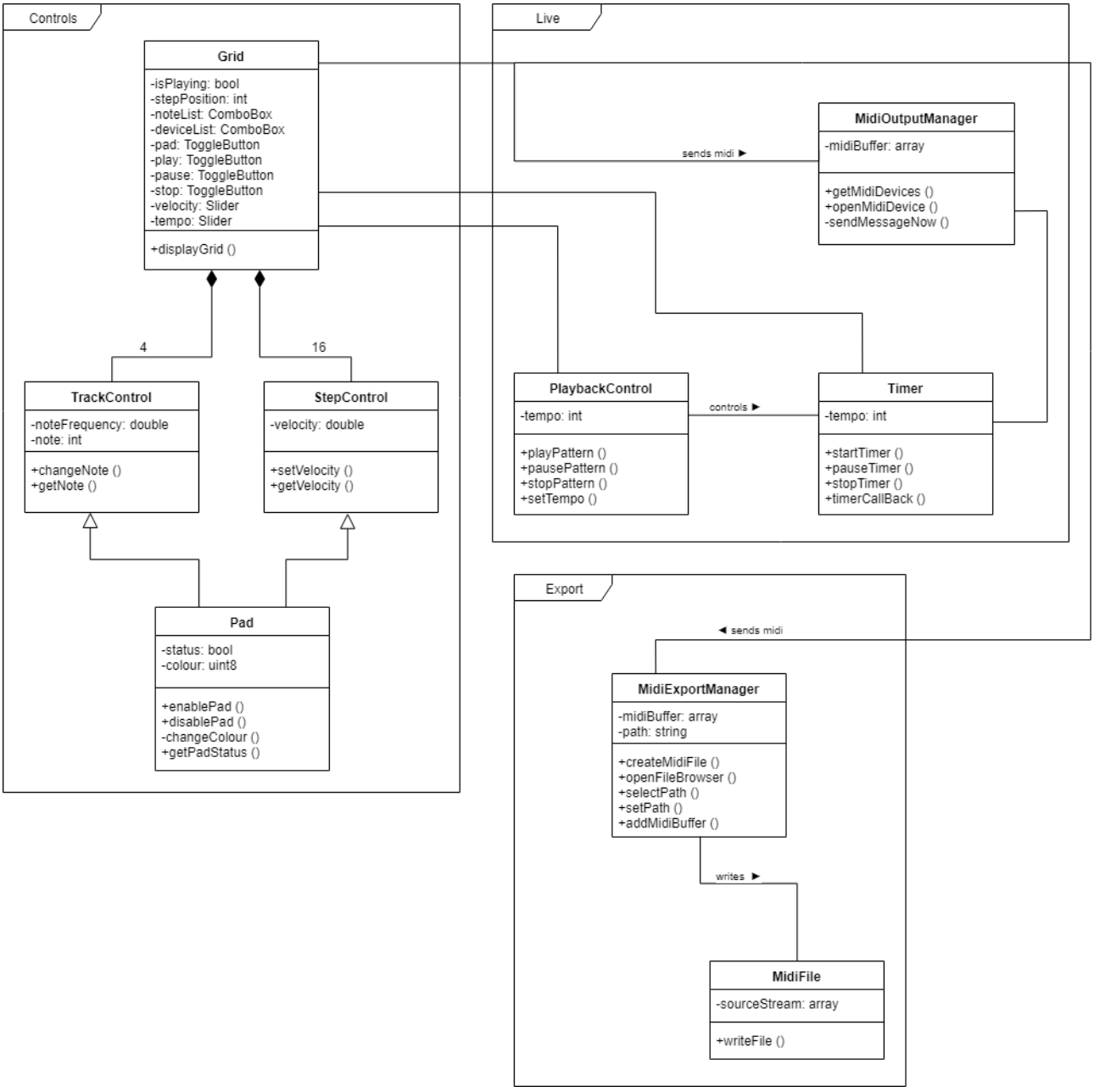

Figure 3 - Class diagram of the software.

2.1.8 Class Narratives

Grid	
isPlaying	True if the program is playing the pattern, otherwise false.
stepPosition	From 1 to 16, indicates the position of the step currently highlighted.
noteList	ComboBox that contains a list of the notes which can be selected by the User, from C0 (24) to B6 (107).
deviceList	ComboBox in which are listed all midi devices connected.
pad	Toggle button that allows the user to interact with the Pad class.
play	Toggle button that allows the User to interact with the PlaybackControl class and play the pattern.
pause	Toggle button that allows the User to interact with the PlaybackControl class and pause the pattern.
stop	Toggle button that allows the User to interact with the PlaybackControl class and stop the pattern.
velocity	Rotary slider that allows the User to set the velocity for each step.
tempo	Linear slider that allows the User to change the tempo of the pattern.
displayGrid ()	Displays the interface through a Visual Display Unit (VDU).

TrackControl	
noteFrequency	Frequency in Hz of the corresponding note.
note	Notes which can be selected by the user through the Grid::noteList
changeNote ()	Changes the note of the track.
getNote ()	Returns the note of the track.
StepControl	
velocity	Velocity of the step, it can be changed by the user through the Grid::velocity slider.
setVelocity ()	Sets the velocity of the step.
getVelocity ()	Returns the velocity of the step.

Pad	
status	Boolean value that indicates whether the pad is enabled (true) or disabled (false).
colour	Uint8 value that indicates the colour of the pad. A pad can change between three colours, depending on its status: disabled pad, enabled pad, mouse over/highlighted pad.
enablePad ()	Sets the Pad::status to true.
disablePad ()	Sets the Pad::status to false.
changeColour ()	Changes the colour of the pad.
getPadStatus ()	Returns the status of the pad.

PlaybackControl	
tempo	Tempo of the pattern in beats per minute (BPM), it can be changed by the user through the Grid::tempo linear slider.
playPattern ()	Plays the pattern.
pausePattern ()	Pauses the pattern.
stopPattern ()	Stops the pattern.
setTempo ()	Sets the tempo of the pattern.

Timer	
tempo	Interval of the timer expressed in milliseconds. It indicates the frequency with which the timer calls the timerCallBack () function.
timerCallBack ()	Called after an interval of the timer, it executes all the instructions it contains.
startTimer ()	Starts the timer.
pauseTimer ()	Pauses the timer.
stopTimer ()	Stops the timer.
MidiOutputManager	
midiBuffer	Array containing a set of midi messages.
getMidiDevices ()	Function that scans the system and returns a list of the midi devices connected.
openMidiDevice ()	Opens one midi device.
sendMidiMessageNow ()	When called, it sends immediately a midi message to the selected midi device.
MidiExportManager	
midiBuffer	Array containing a set of midi messages.
path	String with the path of a file.
createMidiFile ()	Creates a new midi file in the path selected.
openFileBrowser ()	Opens the system file browser.
selectPath ()	Instantiate the MidiExportManager.
setPath ()	Sets a path for the midi file.
addMidiBuffer ()	Adds the selected midi buffer to a midi file.
MidiFile	
sourceStream	Set of midi messages which are going to be written in a midi file.
writeFile ()	Writes the sourceStream to the selected midi file.

Table 3 - Class narratives table, describing each attribute and method of the class diagram.

2.2 Behavioral Model

2.2.1 Sequence Diagrams

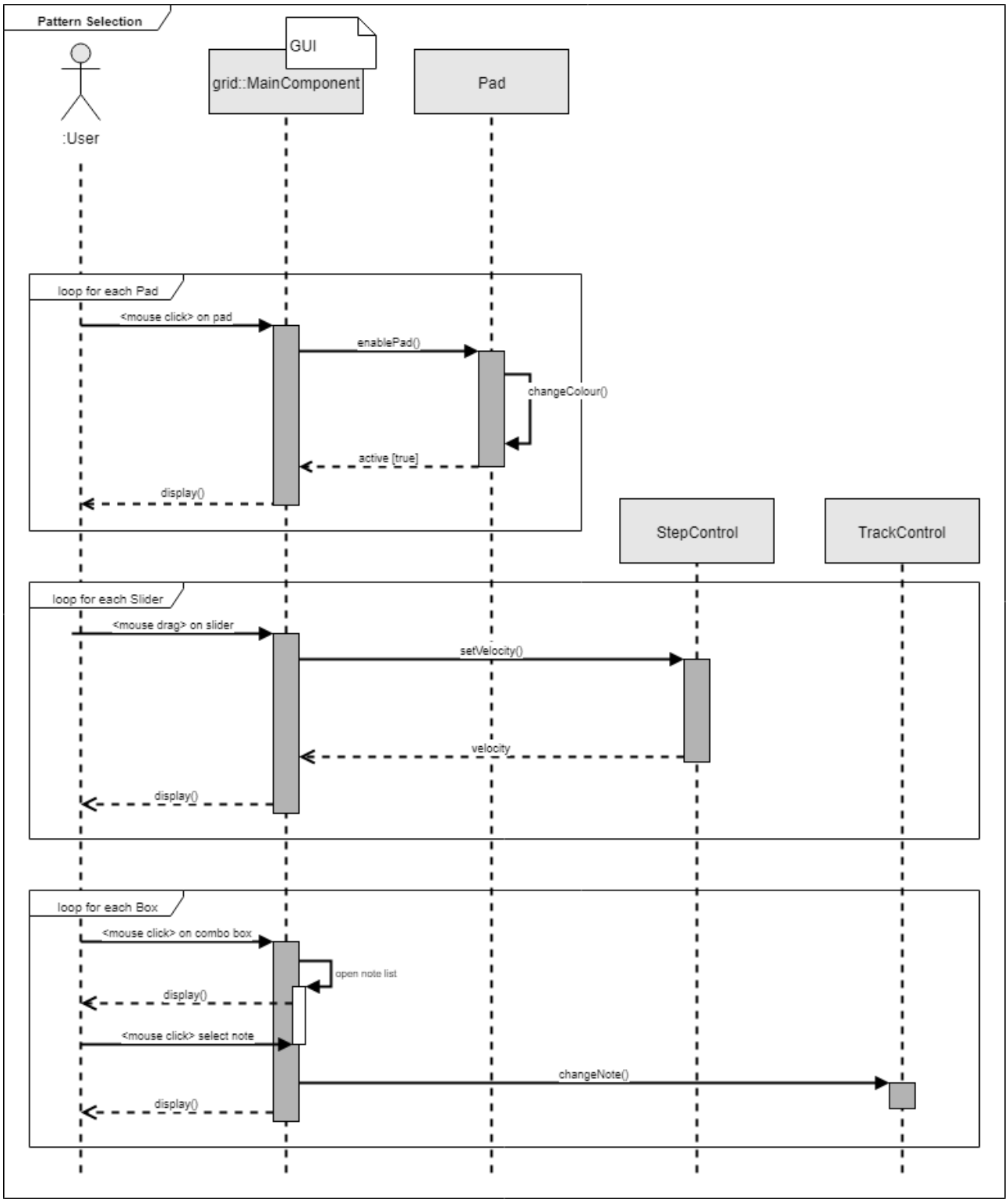

Figure 4 - Sequence Diagram for "Pattern Selection" operation.

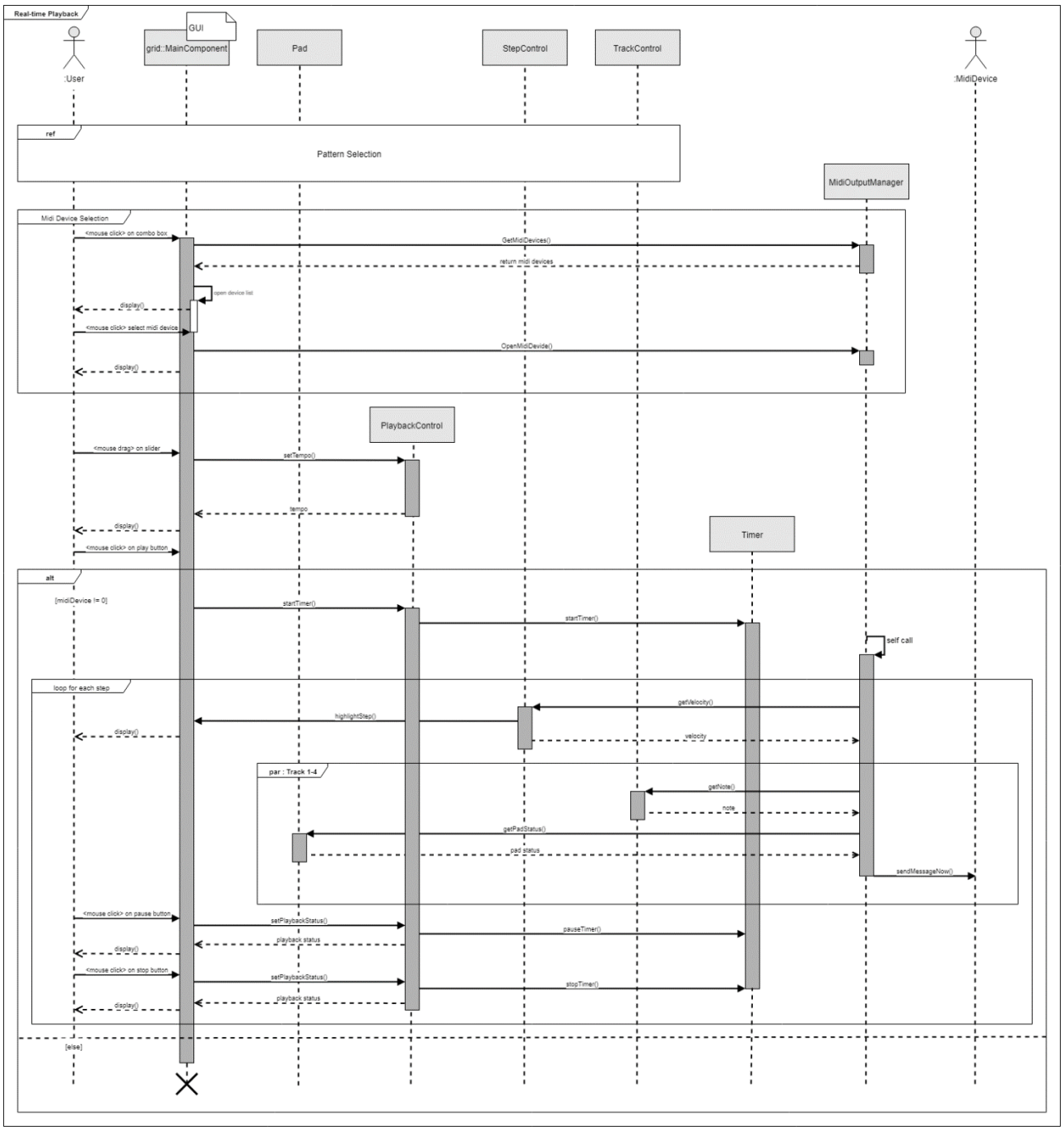

Figure 5 - Sequence Diagram for "Real-time Playback" operation.

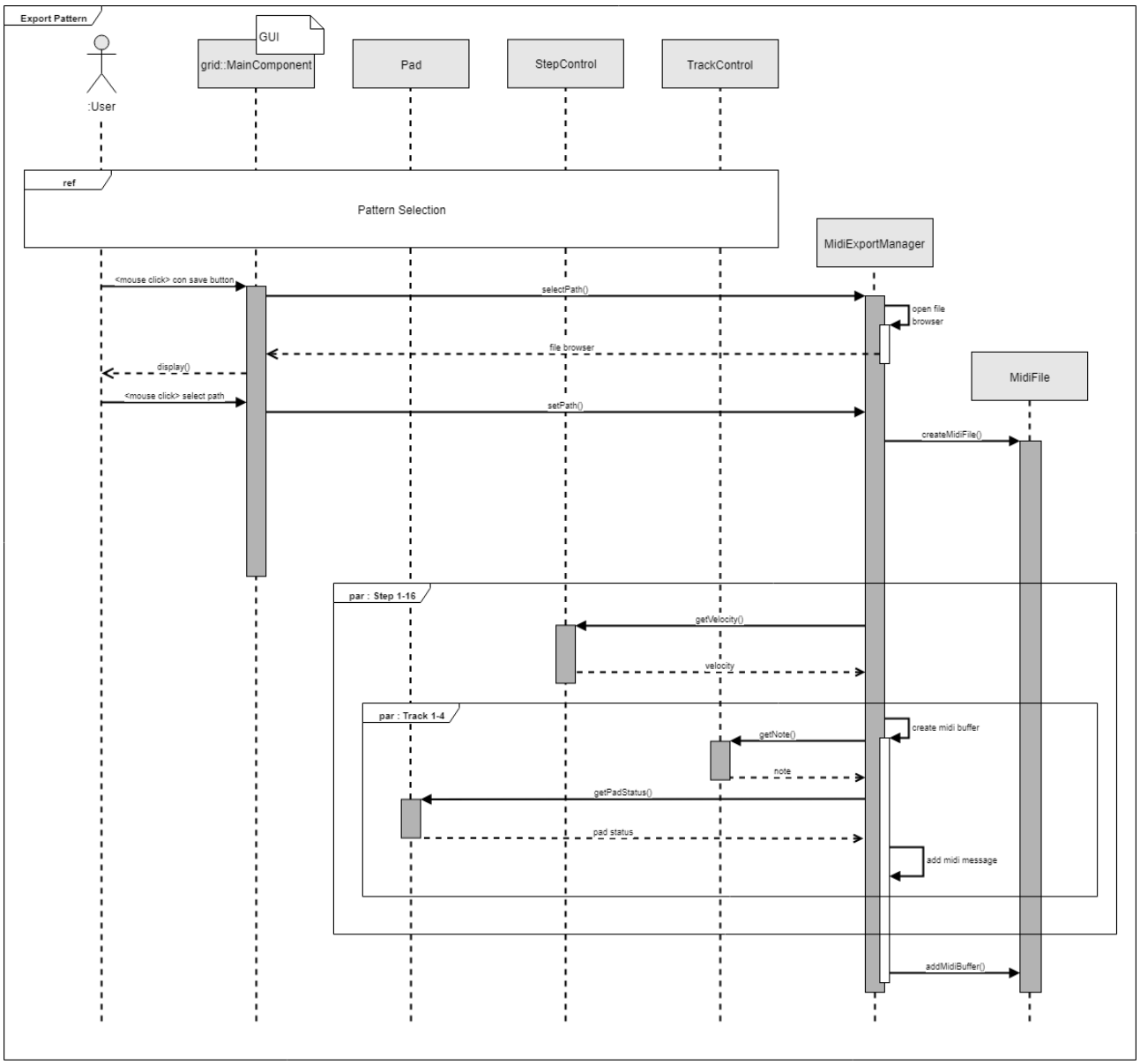

Figure 6 - Sequence Diagram for "Export Pattern" operation.

2.2.2 State Machine Diagram

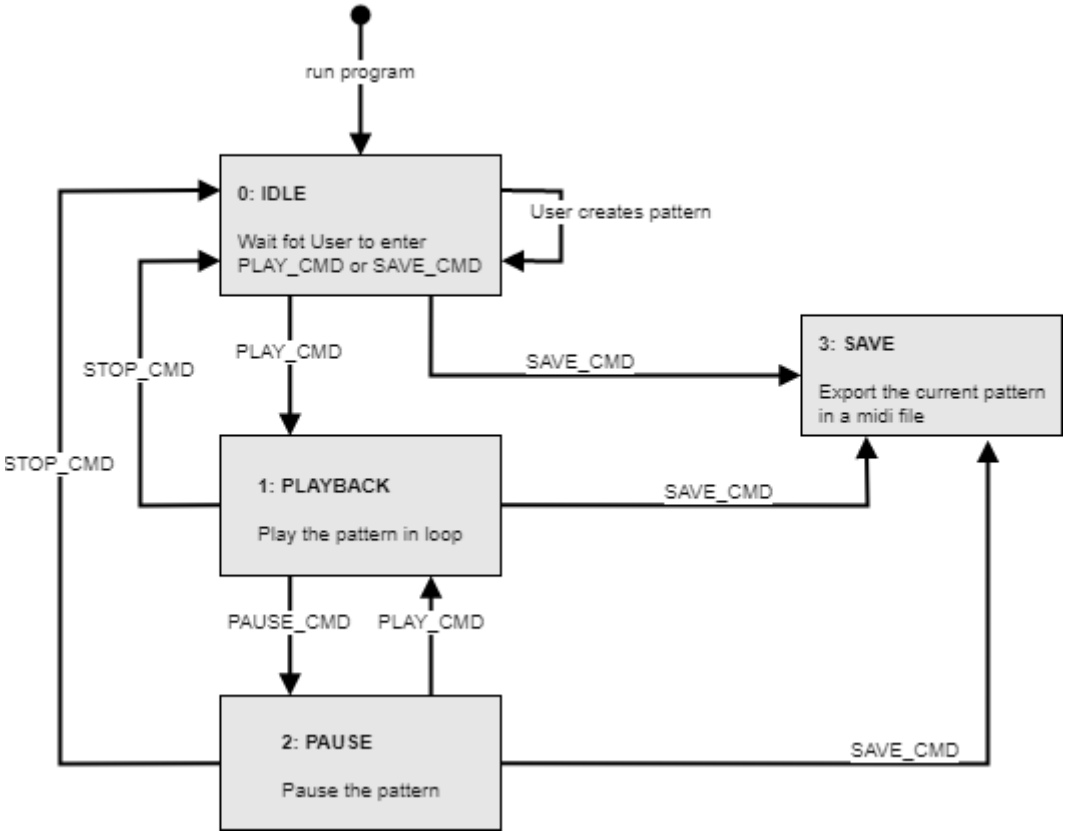

Figure 7 - State Machine Diagram of the software.

2.2.3 Control Narrative

PAD_STATUS

Input: <STATUS>
Output: <STATUS_BOOL>

The PAD_STATUS control takes the input <STATUS> through a toggle button accessed by the User. The default <STATUS_BOOL> is FALSE – i.e. pad inactive –. When the User enters a MOUSE_CLICK on the button, <STATUS_BOOL> value is flipped.

VELOCITY_VALUE

Input: <STATUS>
Output: <CONTROL_VALUE>

The VELOCITY_VALUE control is accessed by the User through rotary sliders placed below each Step. When the User performs a MOUSE_DRAG on the slider, the <CONTROL_VALUE> is passed to the StepControl class.

NOTE_VALUE

Input: <STATUS>

Output: <CONTROL_DATA>

The NOTE_VALUE control is accessed by the User through the noteList object. When the User enters a MOUSE_CLICK the list opens. Then, with another MOUSE_CLICK, the User can select one <CONTROL_DATA>, i.e. a note from C0 (24) to B6 (107) which is passed to the TrackControl class.

TEMPO_VALUE

Input: <STATUS>
Output: <CONTROL_VALUE>

The TEMPO_VALUE control is accessed by the User through a linear horizontal slider. When the User performs a MOUSE_DRAG on the slider, the <CONTROL_VALUE> is passed to the PlaybackControl class.

PLAYBACK_CONTROL

Input: <STATUS>
Output: <PLAYBACK_STATE>

The PLAYBACK_CONTROL is accessed by the Used through a set of three toggle buttons: play, pause and stop. These buttons share the same RadioGroupID, which means that, when the User performs a MOUSE_CLICK and flips the status of one button to TRUE, the others are immediately switched to FALSE. This control passes the <PLAYBACK_STATE> to the PlaybackControl class.

3. IMPLEMENTATION MODEL

3.1 Software Architecture

3.1.1 UI Specification

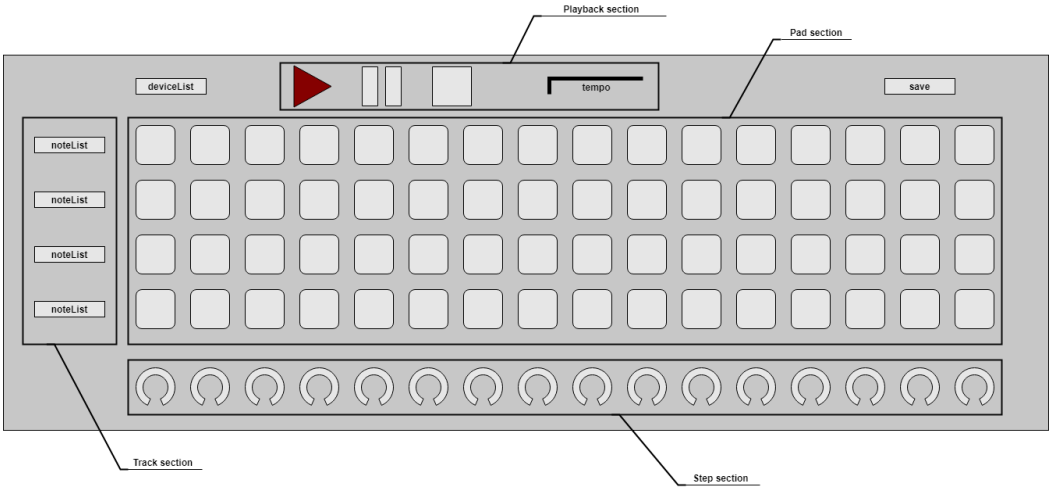

Figure 8 - Design for the Graphical User Interface of the software.

3.1.2 I/O Specification

The User is to input commands in the form of <MOUSE_CLICK> and <MOUSE_DRAG> through the mouse interface module. These are the only inputs the program can take. The software is to output real-time <MIDI_MESSAGES> to an External Midi Device (in “Live Performance” use case) or a <MIDI_BUFFER> to a Midi File (in “Production” use case).

3.1.3 Architecture Context Diagram

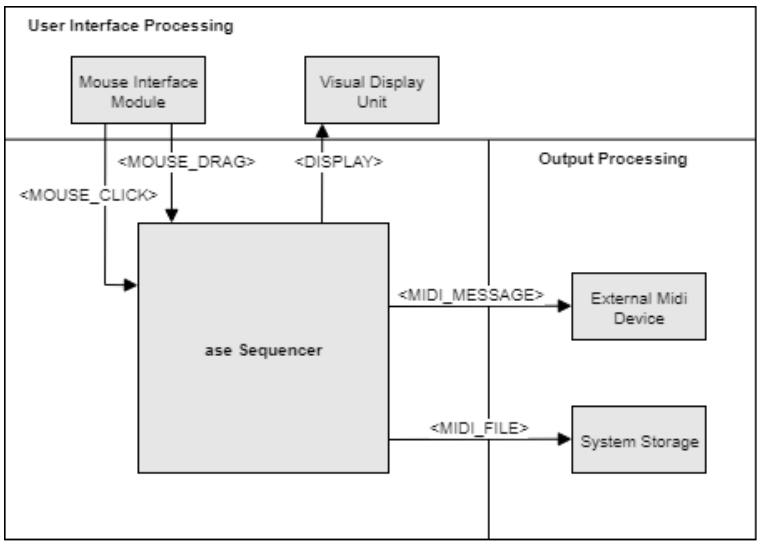

Figure 9 - Architecture Context Diagram of the software.

3.2 Software Structure

3.2.1 MainComponent.cpp

The MainComponent class contains the main cycles of the application: timerCallback to handle the “Live Performance” use case, and saveMidiPattern to handle the “Production” use case.

```
227
228 void MainComponent::timerCallback()
229 {
230
231         if (beatCount > 15) {
232             beatCount = 0;
233
234             midiEventList.clear();
235         }
236
237
238         for (track = 0; track < 4; ++track) {
239
240             pad[beatCount][track].lightUpStep();
241
242             if (pad[beatCount][track].getToggleState() == true) {
243
244                 midiMessage = MidiMessage::noteOn(track + 1, noteControls.getTrackNote(track), stepControls.getVelValue(beatCount) / 127.0f);
245
246                 DBG(midiMessage.getDescription());
247
248
249             }
250
251             else if (pad[beatCount][track].getToggleState() == false) {
252
253                 midiMessage = MidiMessage::noteOff(track + 1, noteControls.getTrackNote(track), stepControls.getVelValue(beatCount) / 127.0f);
254
255                 DBG(midiMessage.getDescription());
256
257
258             }
259
260         midiOutput->sendMessageNow(midiMessage);
261         }
262
263
264
265
266         //GUI Grid
267         if (beatCount == 0) {
268
269             for (int i = 0; i < 4; ++i)
270                 pad[15][i].resetStep();
271         }
272
273         else {
274
275             for (int i = 0; i < 4; ++i)
276                 pad[beatCount - 1][i].resetStep();
277         }
278
279
280         ++beatCount;
281         ++timeCount;
282
283 }
284
```

Figure 10 - Screenshot of the timerCallback method in MainComponent.cpp.

As it can be seen in the code, the timerCallback method is set to perform the scan and the dispatch of just one step at a time. Since this function is called at defined time intervals (chosen by the User through the tempo slider) the cycle marks the time for sending midi messages to the external device. Vice versa, in the saveMidiPattern method there isn’t the need for timed messages, since it wasn’t designed to handle real-time events.

So, in saveMidiPattern (after the FileChooser class is created to allow the User to select a path for the midi file) there are two for cycles which perform the scan and add midi messages to a buffer. These messages are sent to the buffer almost immediately, but they contain a time stamp accordingly to their position in the grid, so that, once written into the midi file, they maintain the playback order.

```
void MainComponent::saveMidiPattern()
{FileChooser chooser ("Select file to save",
        File::getCurrentWorkingDirectory(), "*.mid");

    if (chooser.browseForFileToSave(true))
    {

    double  timerInterval = 60000 / (bpm * 4);

    for (beatCount = 0; beatCount < 16; ++beatCount) {

        for (track = 0; track < 4; ++track) {

            if (pad[beatCount][track].getTogglestate() == true) {

                midiMessage = MidiMessage::noteOn (track + 1, noteControls.getTrackNote(track), stepControls.getVelValue(beatCount) / 127.0f);

                midiMessage.setTimeStamp(timerInterval * (beatCount + 1));
                midiEventList.addEvent(midiMessage);
                midiEventList.updateMatchedPairs();
                DBG(midiMessage.getDescription());
                DBG(midiMessage.getTimeStamp());

            }

            else if (pad[beatCount][track].getToggleState() == false) {

                midiMessage = MidiMessage::noteOff(track + 1, noteControls.getTrackNote(track), stepControls.getVelValue(beatCount) / 127.0f);

                midiMessage.setTimeStamp(timerInterval * (beatCount + 1));
                midiEventList.addEvent(midiMessage);
                midiEventList.updateMatchedPairs();
                DBG(midiMessage.getDescription());
                DBG(midiMessage.getTimeStamp());
            }

        }
    }

    DBG(midiEventList.getNumEvents());


    MidiFile midiFile;
    midiFile.setTicksPerQuarterNote(96);

    FileOutputStream outputStream(chooser.getResult());

    midiFile.addTrack(midiEventList);
    midiFile.writeTo(outputStream, 0);
    outputStream.flush();

    }
}
```

Figure 11 - Screenshot of the timerCallback method in MainComponent.cpp.

3.2.2 StepControl.cpp and NoteControl.cpp

StepControl and NoteControl classes perform similar operation, despite they use rotary sliders and combo boxes, respectively.

```
void StepControls::sliderValueChanged(Slider * slider)
{
    for (int i = 0; i < 16; ++i) {

        if (slider == &velocity[i]) {

            velValue[i] = velocity[i].getValue();



        }
    }


}


float StepControls::getVelValue(int valuePlace)
{
    return velValue[valuePlace];
}
```

Figure 12 - sliderValueChanged method and getVelValue method in StepControl.cpp.

```
442
443 void NoteControls::comboBoxChanged(ComboBox * comboBoxThatHasChanged)
444 {
445     if (comboBoxThatHasChanged == &noteMenu1) {
446
447         trackNote[0] = noteMenu1.getSelectedId();
448     }
449
450     if (comboBoxThatHasChanged == &noteMenu2) {
451
452         trackNote[1] = noteMenu2.getSelectedId();
453     }
454
455     if (comboBoxThatHasChanged == &noteMenu3) {
456
457         trackNote[2] = noteMenu3.getSelectedId();
458     }
459
460     if (comboBoxThatHasChanged == &noteMenu4) {
461
462         trackNote[3] = noteMenu4.getSelectedId();
463     }
464 }
465
466 int NoteControls::getTrackNote(int _track)
467 {
468     return trackNote[_track];
469 }
470
```

Figure 13 - comboBoxChanged method and getTrackNote method in NoteControl.cpp.

3.2.3 TogglePad.h

Although it is not good practice, to respect the timetable it was chosen to create the TogglePad class directly into a header file, without splitting it between .h and .cpp.

```
10
11 #pragma once
12 #include "../JuceLibraryCode/JuceHeader.h"
13
14 class TogglePad : public ToggleButton
15
16 {
17 public:
18
19     void paint(Graphics &g)
20     {
21         Colour fillColour = (getToggleState() == true ? Colour(223, 116, 12) : Colour(20, 31, 51));
22         g.fillAll(fillColour);
23
24         if (getState() == highlight) {
25
26             g.setOpacity(0.5);
27             g.fillAll();
28         }
29
30
31     }
32
33     void lightUpStep()
34     {
35         setState(highlight);
36
37     }
38
39     void resetStep()
40     {
41         setState(ButtonState::buttonNormal);
42     }
43
44 private:
45     ButtonState highlight;
46
47 };
```

Figure 14 - TogglePad class in TogglePad.h

4. JOURNAL

4.1 Concept

20/02/2019 – Beginning of the project, multiple options for the software were considered, and the final choice is the following:

- Drum machine, step sequencer with MIDI functionality and multi-effect processor, split in four modes: LIVE MODE, STEP MODE, TRACK MODE, SEQUENCE MODE

24/02/2019 – Preliminary designs for the software:

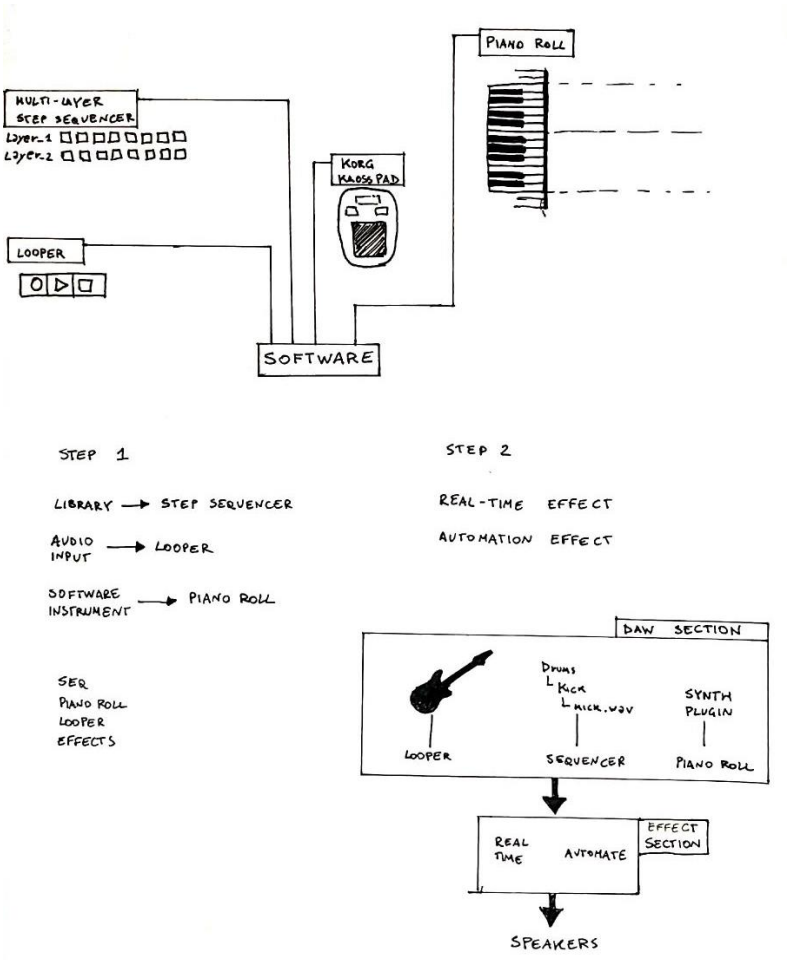

Figure 15 - Preliminary design of the software.

1/03/2019 – First draft of the requirements definition, carrying on the design process:

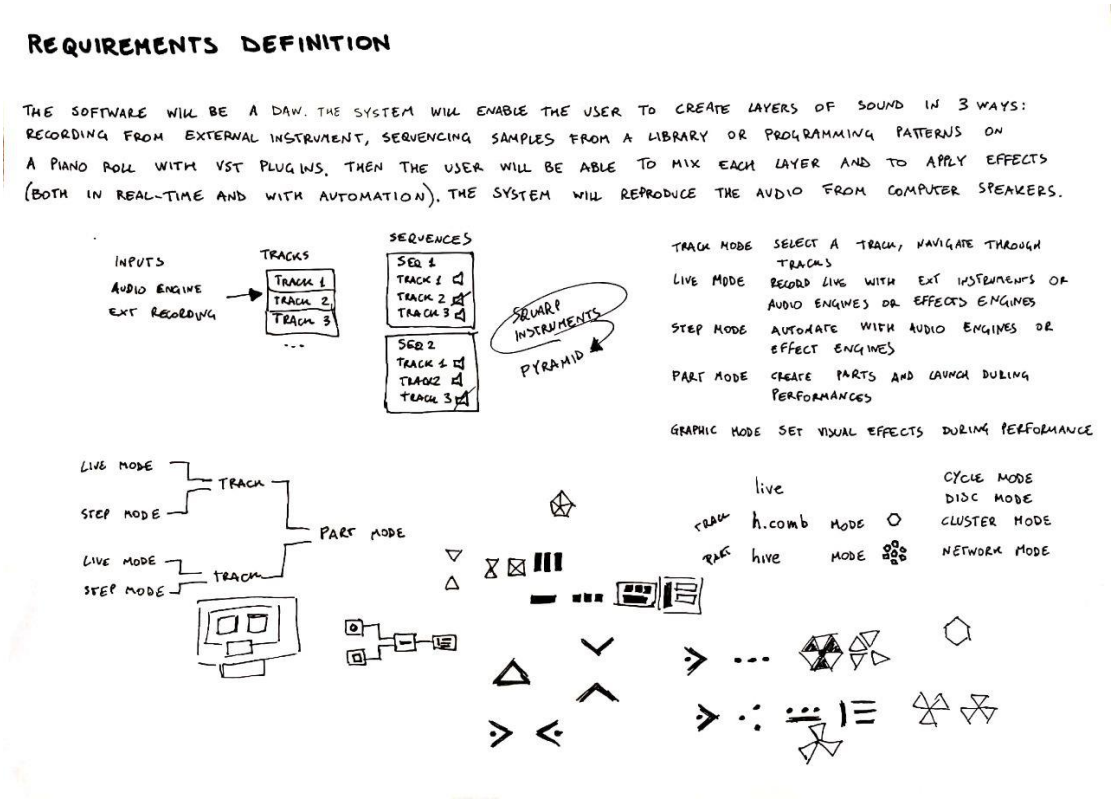

Figure 16 - Requirements definition for the first version of the software.

5/03/2019 – Defining the User Interface and the specification for the LIVE MODE:

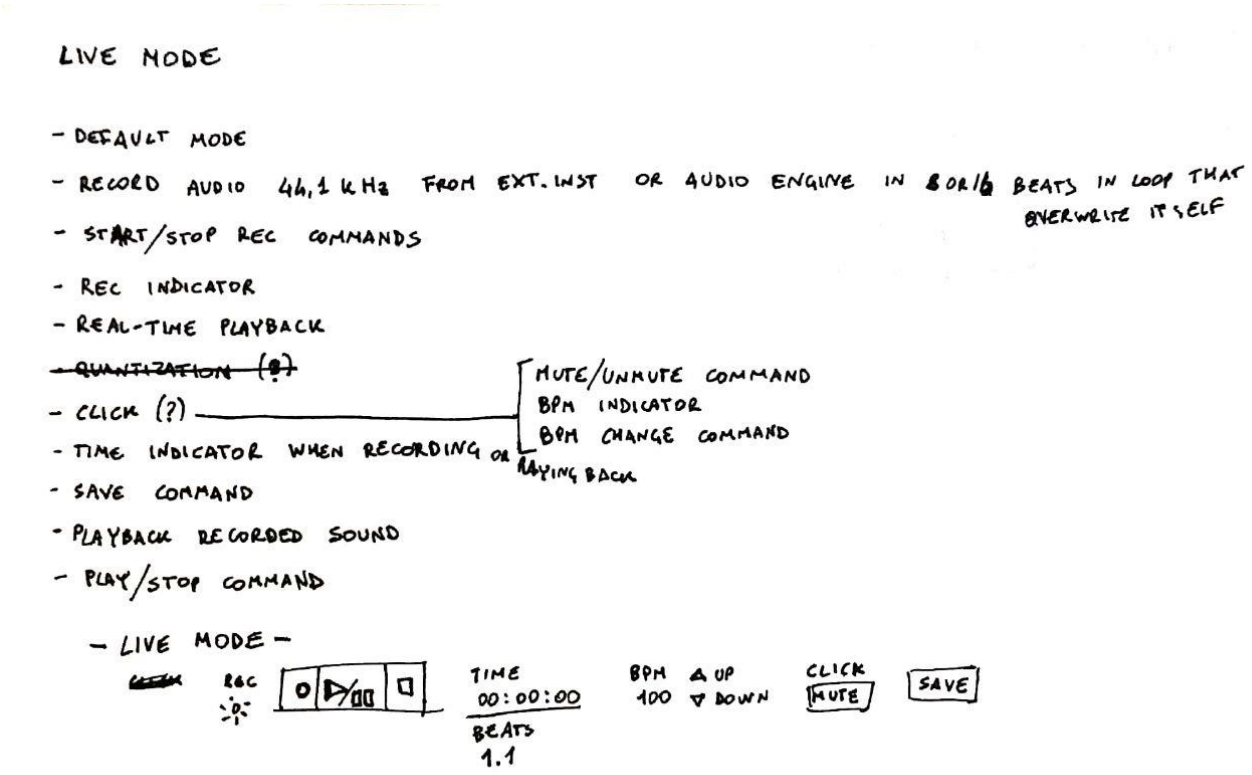

Figure 17 - Details of the LIVE MODE, first version of the software.

8/03/2019 – Since the design process is becoming a lot harder than expected, after talking with the lecturer, the decision is to drastically reduce the software features, because at the moment, with the knowledge possessed, it isn't feasible in the established time period.

13/03/2019 – Completed the requirements definition as seen in section 2.1. The software has been reduced to a step sequencer with midi output functionality.

14/03/2019 – Completed the requirements specification as seen in section 2.1.2. Started to highlight objects, operations and performance criteria for the FAST Analysis.

18/03/2019 – FAST Analysis completed with Table 1 and Table 2.

20/03/2019 – Because of the choice of the programming language (C++), it has been decided to follow an Object-Oriented Programming (OOP) approach. Therefore, the Software Development Summary presented during the lectures, which describes a Process-Oriented Programming (POP) approach, must be integrated with OOP diagrams and methods.

4.2 Design and Documentation

23/03/2019 – With the approval of the lecturer, the software will be designed following the Unified Modelling Language (UML) diagrams and methods, since it's an OOP software design guideline that doesn't different too much from the Software Development Summery presented in class. Completed the use case model in section 2.1.4.

27/03/2019 – Use case diagrams (equivalent to the context diagram in POP, Figure 1 and 2) are completed, and a draft of the use case specifications in 2.1.6 is in progress. To simplify the design process, a commercial example (Figure 18) is being used as a reference, in addition to the JUCE documentation.

Figure 18 - Graphical User Interface of the commercial step sequencer B-Step (Wikipedia, 2018).

5/04/2019 – Use case specifications completed.

7/04/2019 – Drafting the class diagram, some changes were made to the use case specifications, in order to maintain coherence with the documentation and the upcoming diagrams:

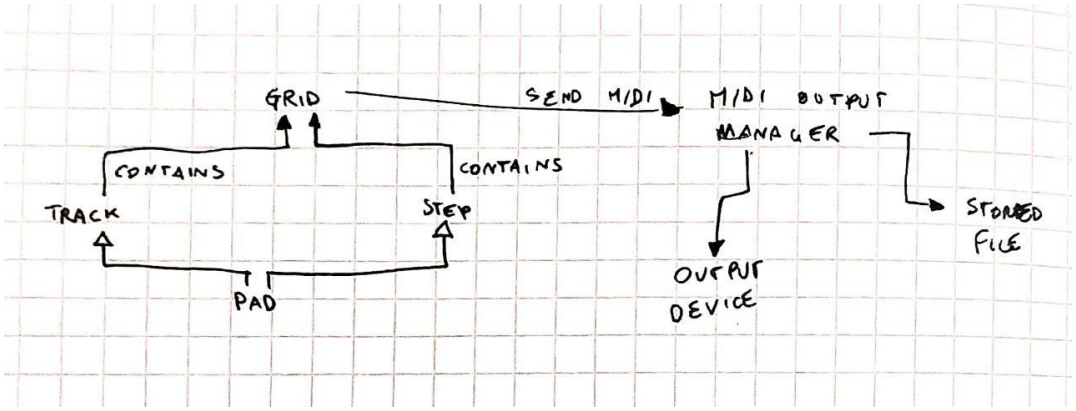

Figure 19 - Draft of the Class Diagram.

9/04/2019 – Class diagram (Figure 3) and Class narrative (equivalent to the Data Dictionary in the POP software design model, Table 3) have been completed.

12/04/2019 – Sequence diagram for "Pattern Selection" (Figure 4) completed, drafting the diagram for "Real-time Playback" operation.

15/04/2019 – Sequence diagram for "Real-time Playback" (Figure 5) completed. The "Pattern Selection" diagram was referenced in order to create a clearer and more understandable diagram, since the real-time playback is the operation that requires the greatest number of steps.

16/04/2019 – Sequence diagram for "Export Pattern" operation (Figure 6) completed, State Machine diagram has been drafted:

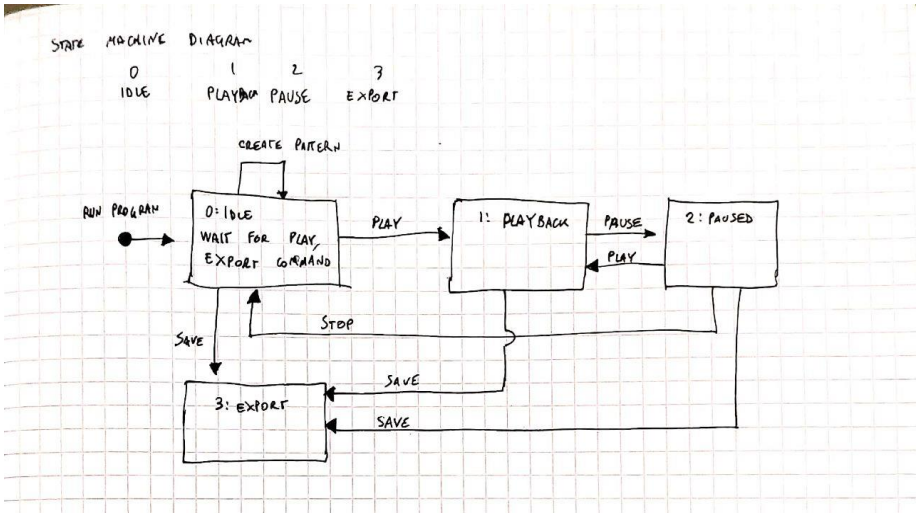

Figure 20 - Draft of the State Machine Diagram.

18/04/2019 – State Machine Diagram completed (Figure 7), drafting the Architecture Context Diagram:

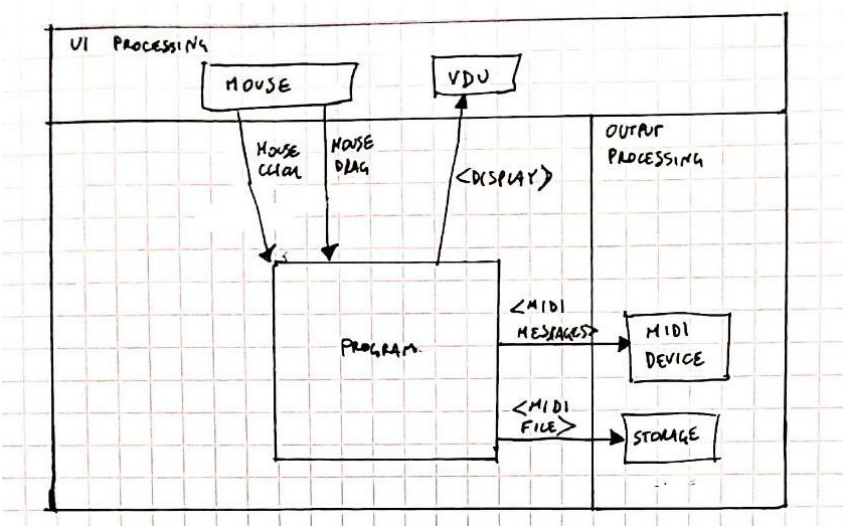

Figure 21 - Draft of the Architecture Context Diagram.

22/04/2019 – Control Narratives (Section 2.2.3), Architecture Context Diagram (Figure 9) and I/O Specifications (Section 3.1.2) completed.

4.3 Implementation

23/04/2019 – Project "aseSequencer" created in JUCE, started to code the Pad class in TogglePad.h and the StepControl class in StepControl.h and StepControl.cpp

26/04/2019 – Pad class completed and linked to GUI elements in MainComponent.cpp, StepControl class almost completed, not yet implemented in the GUI.

27/04/2019 – StepControl class linked to rotary sliders in the MainComponent.cpp, noteMenu combo boxes have been created to deal with the NoteControl class.

28/04/2019 – NoteControl class completed with the list of notes. To simplify the process, the combo box ID of each note represents its midi number.

30/04/2019 - GUI buttons created to control the PlaybackControl class (not yet implemented). Added tempo selection and Timer class to the MainComponent.

1/05/2019 – Working on the timerCallback and saveMidiPattern methods, two different for cycles where created to handle the scanning of the grid: in timerCallback, the cycle follows the beat of the timer (selected

through the tempo slider), in saveMidiPattern, the cycle is immediate, since there's no need to send timed midi messages, the time information is already contained in the message itself.

2/05/2019 – Minor GUI improvements, added the deviceMenu combo box to select the external midi device.

5. CONCLUSIONS AND EVALUATION

After the design process, the implementation of the software went really well and proceeded smoothly. Minor debugging was needed, just to double check the functionality of the application. During the coding process, no bugs where encountered, and the software works exactly as expected. Although the application may benefit from further optimization (the Timer class is not 100% precise, maybe it could be replaced by the JUCE's HighResTimer class, to handle timed midi messages with more precision), the overall evaluation is positive, also considering the notable improvements that were made comparing this programming project with other modules.

6. REFERENCES

Deitel, P. J. and Deitel, H. M. (2017) **C++ how to Program**. 10th Edition edn. Pearson.

Hodge, J. (2017) **The Audio Programmer** [Online]. Available from: <https://www.youtube.com/channel/UCpKb02FsH4WH4X_2xhIoJ1A/about> [Accessed: 28 April 2019].

JUCE (2019c) **JUCE Learn**. Available at: https://juce.com/learn [Accessed: 29 April 2019].

Kirill Fakhroutdinov (2018) **The Unified Modeling Language** [Online]. Available from: <https://www.uml-diagrams.org/> [Accessed 23 March 2019].

Tutotialspoint (2019) **Learn UML** [Online]. Available from: < https://www.tutorialspoint.com/uml/index.htm > [Accessed: 8 March 2019].

Wikipedia (2018) **B-Step Sequencer**, 30 October 2018, 12:06 [Online]. Available from: <https://en.wikipedia.org/wiki/B-Step_Sequencer> [Accessed 27 March 2019].