A modular synthesizer

Programming studio 2 – Final document

Elberkennou Younes

10.5.2023

Younes Elberkennou
1008920

Contents

Younes Elberkennou
1008920

1 Personal information

- Name: Younes Elberkennou
- Student number: 1008920
- Degree program: tietotekniikka
- Year of studies: first
- Date: 10.05.2023

2 General description

2.1 About synthesizers

A synthesizer is a musical instrument which uses electronic circuits to generate sound. This includes computer hardware, where instead of extracting a waveform from oscillating circuits, programs are used to replicate these waveforms. These are referred to as software synthesizers, or “softsynths”.

A synthesizer is most often seen as consisting of seven main components:

- The oscillator(s) generates the basic waveform(s).
- The amplifier controls the volume of the signal.
- The filter filters out certain frequencies of the signal.
- The pitch envelope controls the oscillator’s frequency as a function of time.
- The volume envelope controls the signal’s volume as a function of time.
- The filter envelope controls the filtered frequencies as a function of time.
- The low frequency oscillator provides an oscillating signal that can be used to control other components.

However, a synth can consist of many possible combinations, or routings, of these components, as well as separate components that generate different effects. In general, any component that does some form of transformation on a signal can be used. Here lies the idea of a modular synthesizer: the user can select and create custom wirings between components, thus creating their very own synthesizer.

2.2 The result of this project

During the last months, I have built a program that allows the user to create their own modular, programmable synthesizer. The user is presented with a workspace like that of Unreal Engine’s material editor (1). The user can add and remove nodes that represent individual components of a synthesizer, as well as connect them together using “wires”. Each individual component has parameters, which the user can tweak to their liking. Synths can be saved and loaded, as well as played either via the computer’s keyboard or a separate MIDI controller. Synth components can be chosen from a built-in library.

From the original plan, the features implemented are the following:

All the “bare minimum” requirements, that is:

- Ability to generate a sound signal and direct it to the system’s audio output. Ability to generate a range of simple sounds:
  - Sine
  - Triangle
  - Sawtooth
  - Square
  - Noise

- Ability to model all seven main components of a synthesizer.
- Presents the user with a graph-based editor for building modular synthesizers, the nodes representing components, and connections their wiring.
- Ability to save and load synths and synth configurations.
- Ability to read midi input, both from the computer's keyboard and from a dedicated midi controller.

Features necessary for improved usability:

- Some arithmetic nodes, such as constant values, addition, division and multiplication (amplifier)

Features to flesh out the program:

- Debatably, the ability to modify duty cycle (quite convoluted to do as a user)
- Full capability of FM synthesis
- Ability to make the synth velocity-sensitive
- A delay module

Stretch goals

- None of the stretch goals have been achieved.

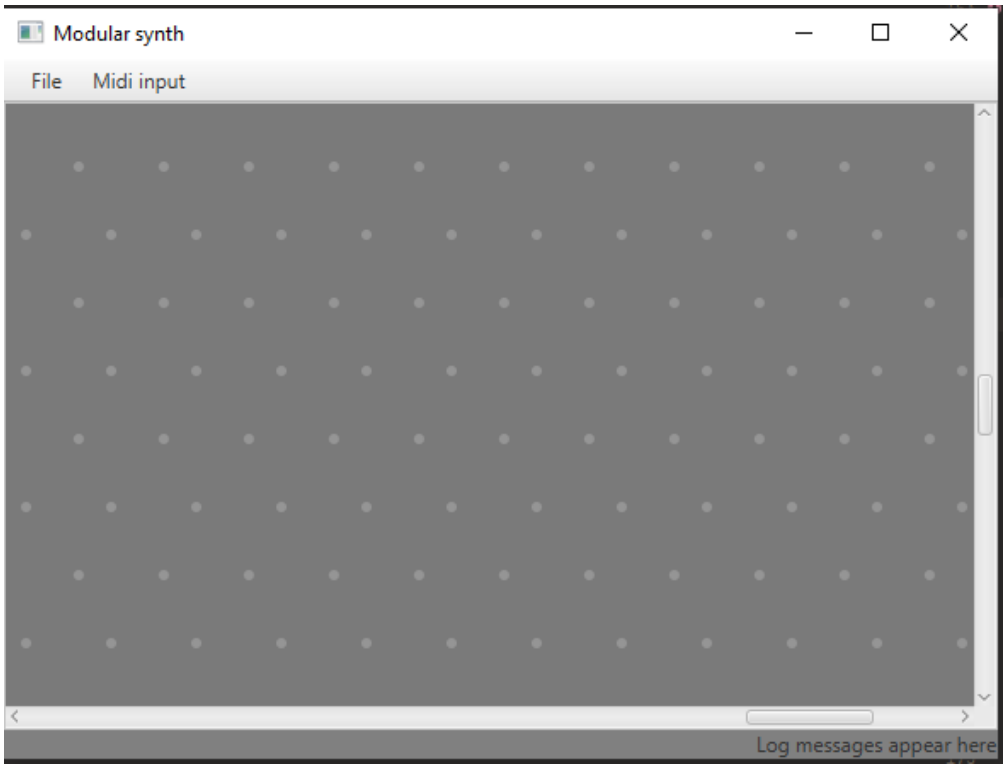
Considering the scope of the project, it has undoubtedly reached the "hard" level.

3 A guide to using the program

3.1 Starting up

The returned Zip file contains a packaged .jar file. This is the definitive build, and the recommended way to use the program.

Upon startup, you will be presented with the following screen:

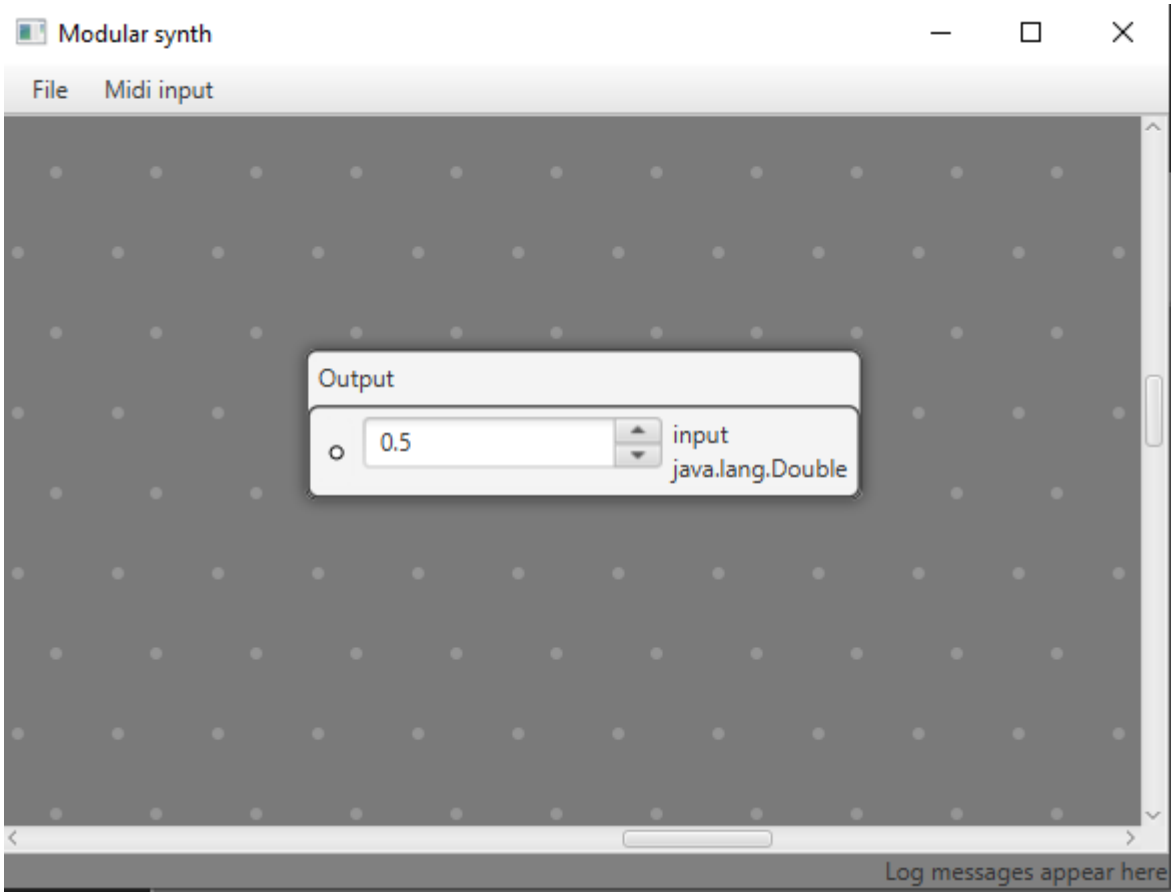


The top bar houses the “File” and “Midi input” menus. The file menu houses buttons for saving, loading and creating a new synth. The midi input menu lets the user select which midi devices the synth will listen to. The window can be resized to your liking, as well as maximized and minimized.

The bottom bar displays the last log message. Here, information about the synth’s internal workings will sometimes pop up.

The crux of the UI is the dotted workspace. This can be panned by clicking and dragging upon it with the mouse, or alternatively using the scroll bars. Zoom can be adjusted by the MIDDLE MOUSE WHEEL, or an equivalent scrolling action (unfortunately, pinching does not work).

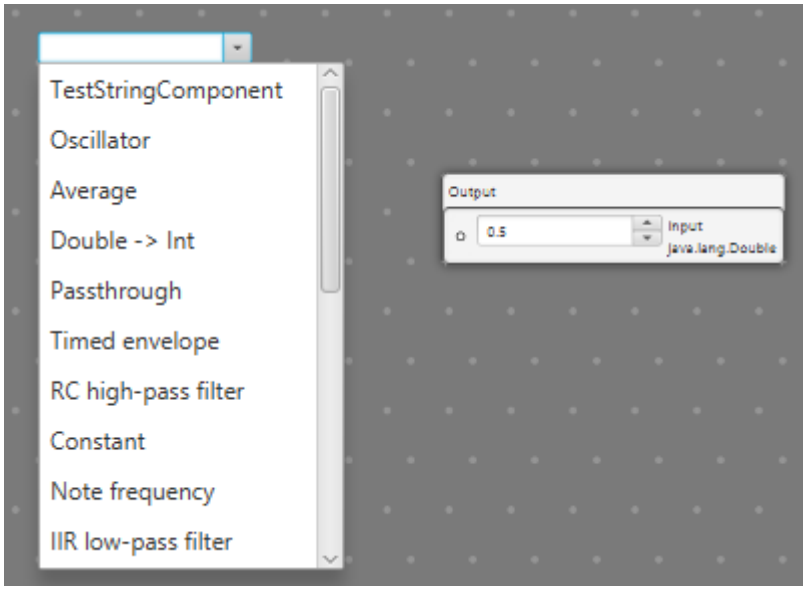
Upon startup, a default “blank canvas” of a synthesizer should be loaded. This holds only a single node, the output component. See if you can find it:



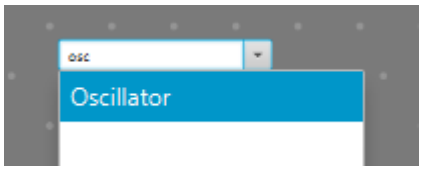
3.2 Editing

3.2.1 Creating nodes

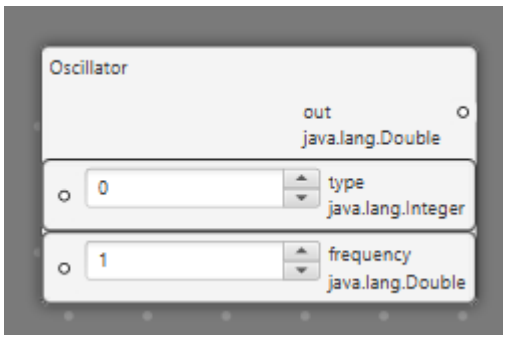
Now we’re ready to start creating a synth. Position the mouse on an empty space and press the SPACEBAR. A search box will pop up:



Here, you can search for synth components either by typing the name into the search box, which will filter the list, or by just finding the wanted component from the list. Try typing in “oscillator”, and select the sole member of the list by left clicking it, or using the arrow keys and pressing ENTER:



This will create a new node on the workspace:



3.2.2 Wiring

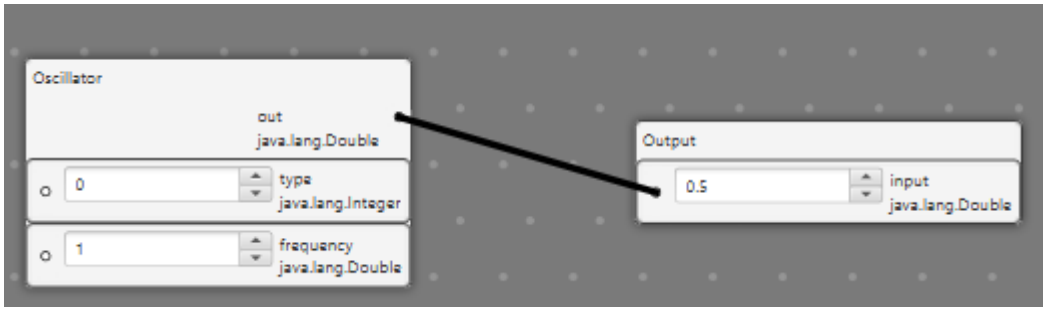
Nodes by themselves don’t do much, just like disconnected electrical components in real life. To bring our synth to life, we need to wire the components together. Our end goal is to generate a time-dependent function, the form of which resembles the waveform of a sound we are after. We then pipe this function into the “output” node, which sends the signal to be rendered by our audio device.

Each component, or node, acts as a sort of function: they generate values based on their inputs. Some of these inputs can be controlled by the user: in the oscillator node, for example, the “type” and “frequency” parameters dictate the sort of signal that will be generated. By editing the values in the input fields, the “default” values of these parameters can be set. By connecting another component to the parameter, the default value is overridden by the connected component’s output signal.

Younes Elberkennou
1008920

Let us start by connecting the oscillator to the output. This is done by the circles on the ends of each box in a node. I call these sockets: 

Left-click and hold on the “out” socket on the oscillator. Drag your mouse pointer over the “input” socket on the output component, and release the mouse. A line should have formed. You may have heard a pop, if successful:



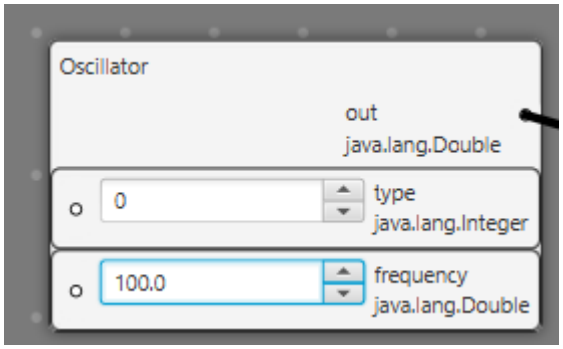
Should you want to disconnect two components, hover on the socket you want to disconnect, and CTRL+LEFT CLICK on it to disconnect it. If you wish to delete a component, click on it with the mouse and press DELETE.

Observe that the sockets each have a data type. The synth won’t allow you to connect sockets of a different type. Conversion nodes, such as “Double -> Int” can be used to do this. A node’s output cannot connect to its own input, either.

3.2.3 First sound

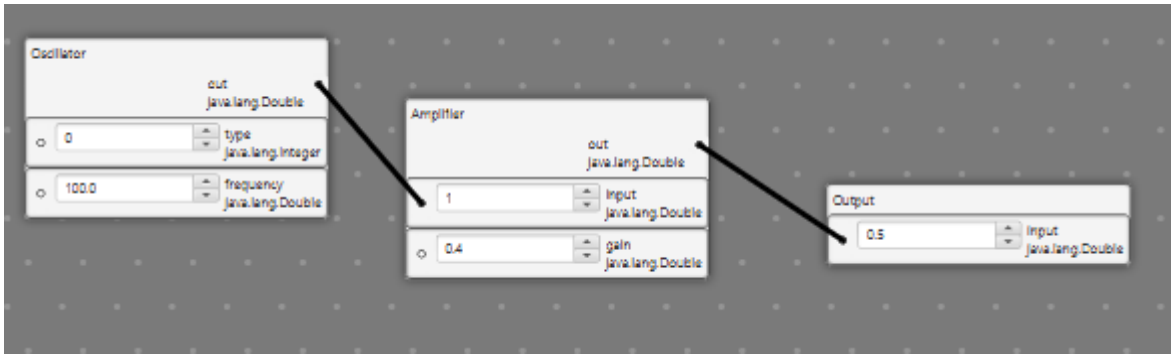
We still don’t have any sound! No audible sound, at the very least. Indeed, if the oscillator is correctly connected, a signal is being sent to your speakers (a sine wave, by default). Only, its frequency is too low to be heard.

To fix this, edit the “frequency” field. Putting something like 100 will produce a low hum.



If invalid values are input into the field, it will turn red:

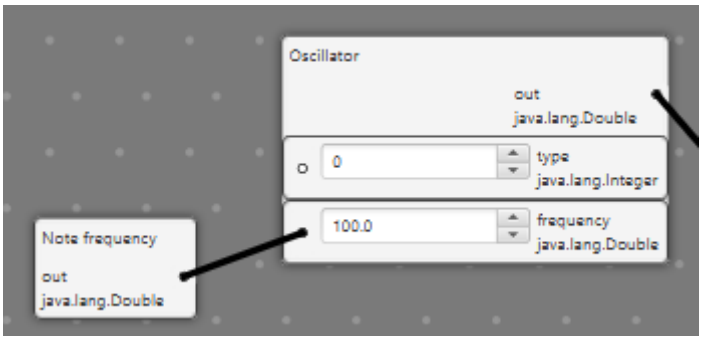
By default, the sound is quite loud. This can be addressed by an “amplifier” node:



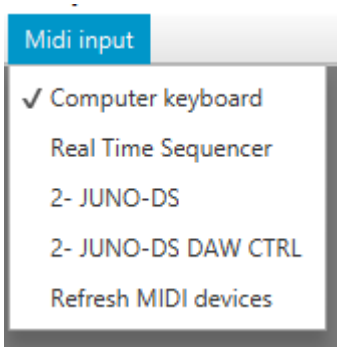
Adjust the value of gain to make the input louder or quieter. Try different values on the oscillator's "type" parameter. This will change the generated waveform (see appendices.

3.2.4 Playing notes

This being a musical synth, we would like to play notes with it. This can be done with the help of the "note frequency" component. Use this to control the oscillator's frequency:



For this node to do anything, it needs user input. This can be provided via the computer's keyboard, or an external MIDI device. Open the "Midi input" menu:

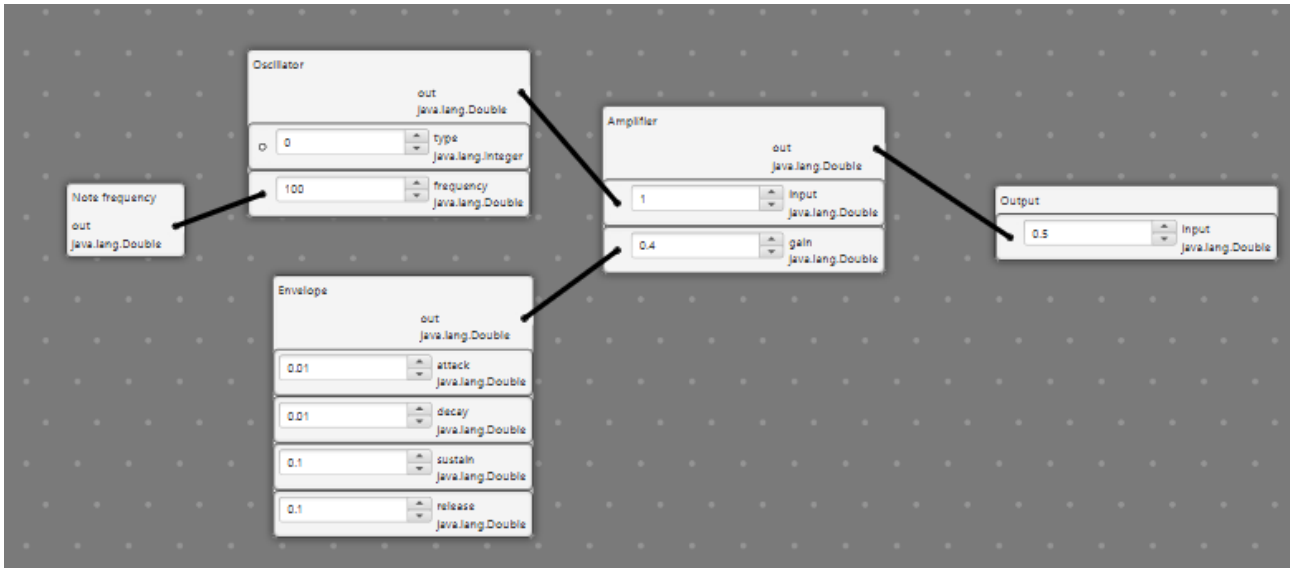


Each item with a checkmark will send midi messages to the "Note frequency" node. You can toggle which devices provide input by clicking on them. For now, make sure "Computer keyboard" is checked.

Bring focus to the workspace and try to press some keys on your keyboard's lower two rows (keys z-k). You should hear varying sounds depending on the key pressed.

The sounds are still continuous and monotone. If we want to be able to play discrete notes, we can use the "envelope" component. This generates a signal between zero and one that increases in the beginning and decreases to zero in the end. Connecting this to the "gain" parameter of the amplifier, we get the desired effect:

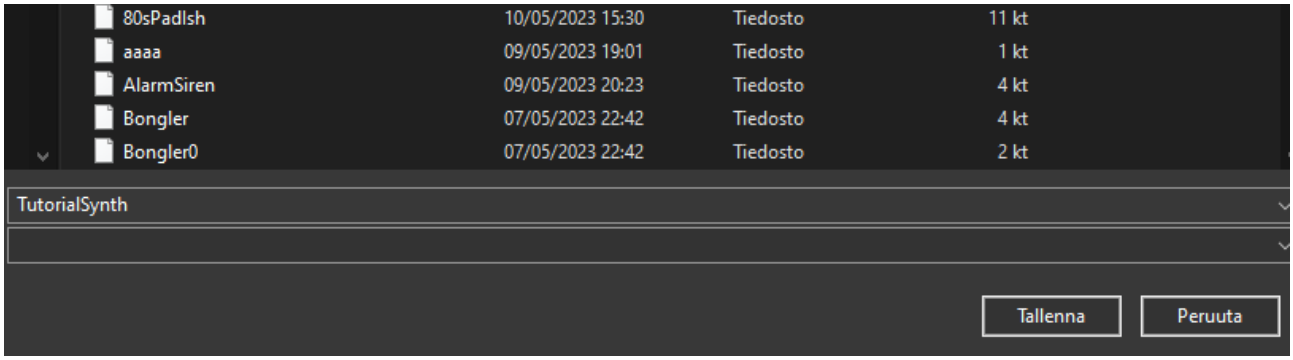
Younes Elberkennou
1008920



Observe that even with such a simple setup, we can generate a vast variety of sound just by varying the Envelope's different parameters.

3.2.5 Saving and loading

Great! We now have a working synthesizer. If we would like to access it later on without having to rebuild it, we can save it. This can be done via File -> Save Synth. This will open a file chooser window, and ask you for a directory as well as a name:



Name your synth and press "save". You can overwrite a synth you have saved earlier with the shortcut CTRL+S.

Loading a synth is just as simple. File -> Load synth will ask you to point to a synth file. Select your saved file, and the workspace will take the exact form it had at the moment of saving. The program comes with a variety of demo synths. Give them a try! (hint: the ones with coherent names are usually the best)

Should you get bored of the same synth, you can create a new one via File -> Create empty. The program will ask you if you want to save your current synth.

3.3 A summary of editor actions

- General

  Drag: pan

  SCROLL: zoom

  SPACE: search for a component

CTRL+S: Save current synth

- Node
  mouse-drag: move

  DELETE: remove

- Socket
  Drag-and-release: connect

  CTRL+LEFT MOUSE: disconnect

3.4 Caveats

Observe that due to the great variety of components, not all of them have been tested with a great variety of values. If you find the synth abruptly stops working, try deleting the last component you edited and re-inserting it.

Some nodes in the library are “second-class”, so to speak. An example of this is the IIR low-pass filter. It has been hastily implemented, with minimal understanding of its functioning principle. I would instead recommend the RC low-pass filter. To get an idea of the nodes I consider of a sounder quality, have a look at the demo files (open them up with the synth. They can be found in the resource folder).

4 Program structure

The general structure of the program does a good job of following the one presented in the general plan. The realized structure is, however, vastly more complicated and intricate than that. Because of this, the following UML diagram only presents a very minimal representation of the actual structure, as attempting to replicate it in any detail would lead to a very noisy diagram. Observe that I’ve included the main class in the UML, as it is ultimately the point of interaction between many of the program’s parts. This could be seen as a flawed design, as main should typically be kept minimal.

Younes Elberkennou
1008920

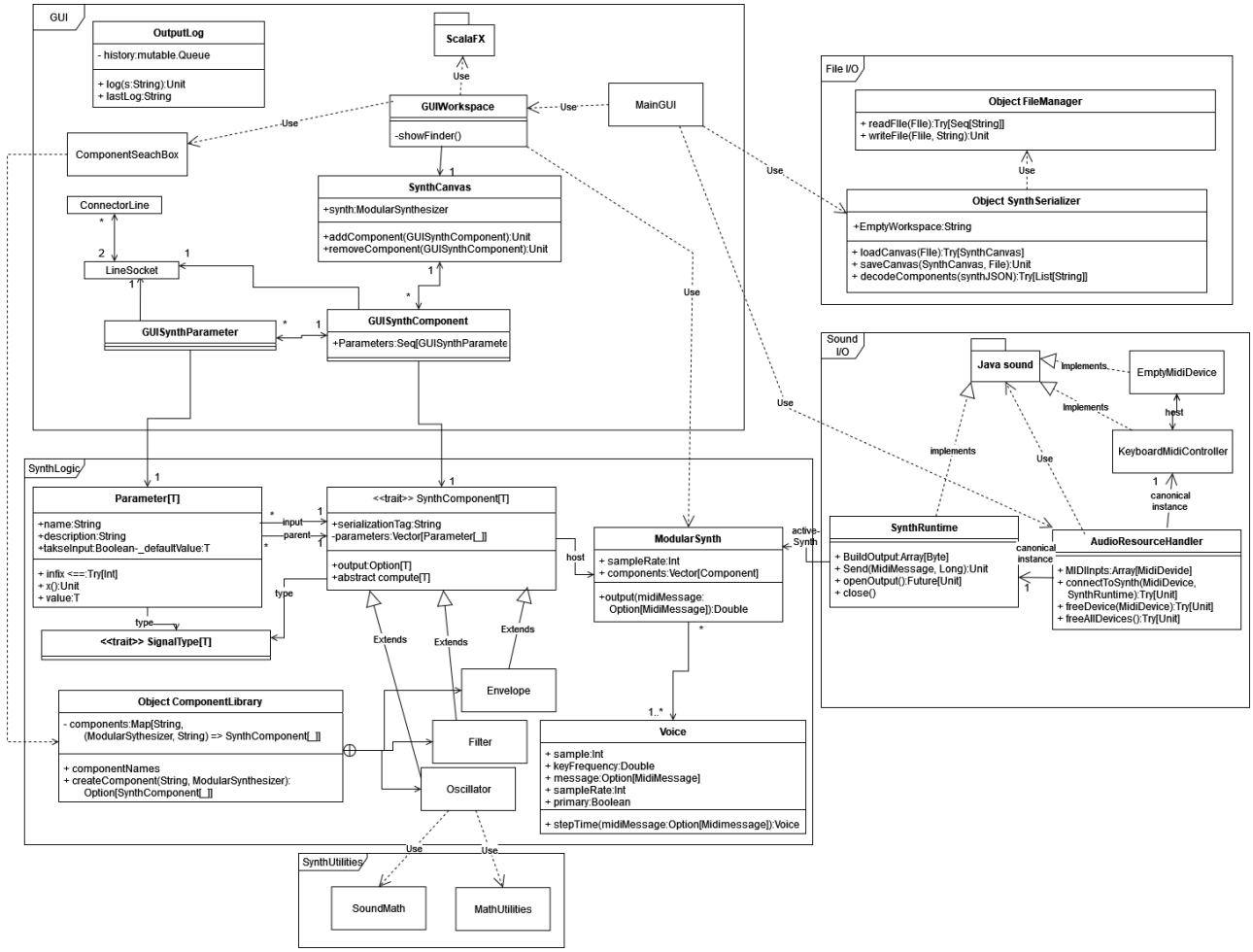


Figure 1-UML diagram of the realised program

Four main packages are found: SynthGUI, SynthFIleIO, SynthLogic and SynthSoundIO. Each of these handles the domain they are named after. The following table presents the main classes in these modules, along with their function:

SoundIO	Interfaces with the Java sound library. The class SynthRuntime essentially works as an I/O adapter between our synthesizers and the Java sound library. Its buildOutput method relays Midi messages and time information to the synthesizer, and returns the waveform generated by the synthesizer(s) based on those inputs. Its send method sends this signal to an audio output. The class AudioResourceHandler is misnamed, as in its current state, it focuses on handling MIDI devices specifically. It allows us to access different MIDI resources. The KeyboardMIDIController class allows us to use the computer keyboard as any other MIDI controller.
SynthLogic	The bread and butter of our program. This module handles all that has to do with waveform generation and audio processing. The most prominent classes are ModularSynth and Component, which form a composition relationship. A modularSynth has one output “component”, which gives SynthRuntime the output resulting from the calculation. Each component essentially represents a transformation of the audio signal. Components hold parameters. These may have different types, as do components. components can plug their output into inputs of the same type. The allowed types are defined by extending the SignalType[A] trait.

	ModularSynth uses the Voice class to relay information about the current frame to the components.
GUI	Handles user interaction with the state of the program. The messiest package of the bunch, all others interact here. The specifics of the classes here aren't all that interesting in the bigger picture, as they are mostly "boilerplate" to mount our program to. What's most important in this package is the role of the custom GUI classes: The GUIWorkspace class can be compared to an easel: it can hold a canvas onto which the user can "paint" their synth. The SynthCanvas is an analogue to a painter's canvas: this is what the user edits. The GUISynthComponent class is our GUI analogue to the SynthComponent class, and the GUIParameter class to the Parameter class. They both refer to their respective analogues and update them based on their changes. The classes Socket and ConnectionWire facilitate the wiring of components and are used to trigger updates in the logical structure of the synth. The OutputLog object is a general utility for displaying logging information to the user. Here, you may see things that don't make sense from a standpoint of sensible design (for example, circular referencing between GUISynthComponent, GUISynthParameter and Socket). For this, I have no justification but my own inexperience and the time constraint. I simply went with the first solution I deemed sensible.
FileIO	Used for saving and loading files. The FIleManager object holds all file read/write operations in a convenient form. These methods return Try objects, as file I/O is typically prone to exceptions beyond our control. The SynthSerializer object holds methods for translating a workspace into and from its serialized form. Not shown in the diagram are the encoder and decoder objects defined for different classes that SynthSerializer makes use of.
SynthUtilities	Generally useful mathematical functions which are not specific to the program.

ComponentLibrary presents a structure which I find particularly interesting: Each synth component is implemented as a class. We create a list of string-lambda tuples (a dictionary). The string serves as a key for the lambda. Each lambda instantiates one of the synth component classes. This allows us to create a specific type of synth component only by supplying a string (key). This structure enables a clean way of integration between different parts of the program: SynthLogic, SynthGui and SynthFileIO.

5 Algorithms

Of the many algorithms our program uses to generate sound, the most important one is the general sound generation algorithm, or how our synth's building blocks are used generate a result. Furthermore, I present the idea behind a nontrivial component, the filter.

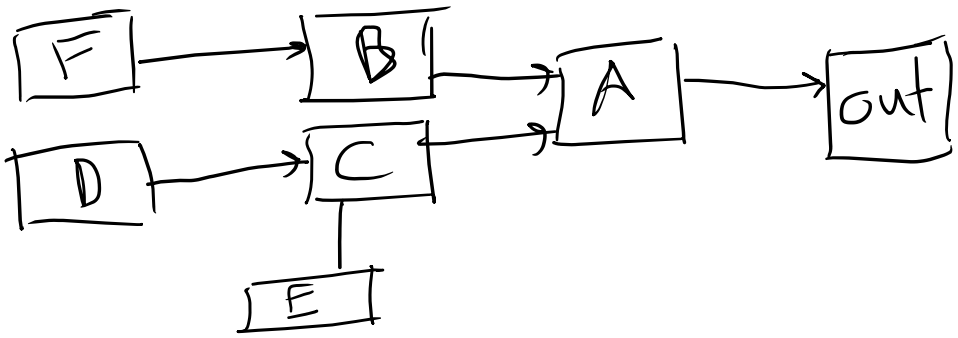
5.1 Sound generation

Our synthesizer consists of components, each of which may have parameters, one or several inputs, and an output. When determining the audio sample at time t, it would make sense to recursively call the output of its input nodes. This would lend itself particularly well to a functional style of programming.

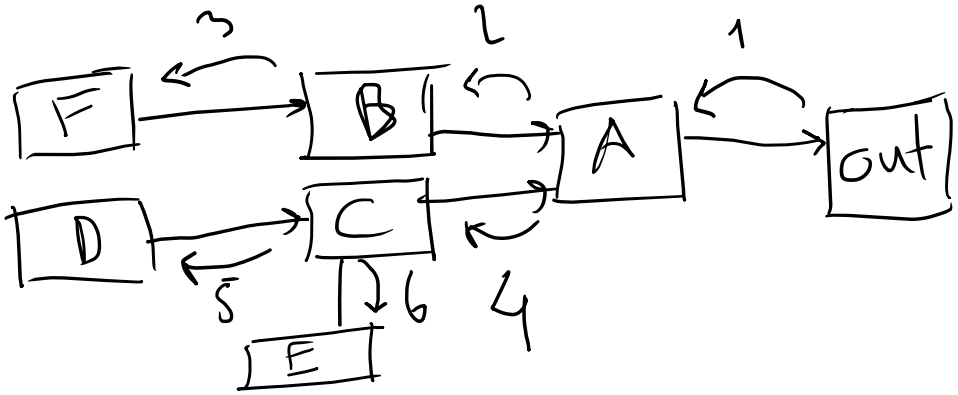
Younes Elberkennou
1008920

In the ModularSynth class, the structure defined by the components forms a tree. This tree's root is the single output node, its nodes are some sorts of transformations, and its leaves give the synth some sort of input, whether it be an audio clip, live audio input, or a generated waveform.

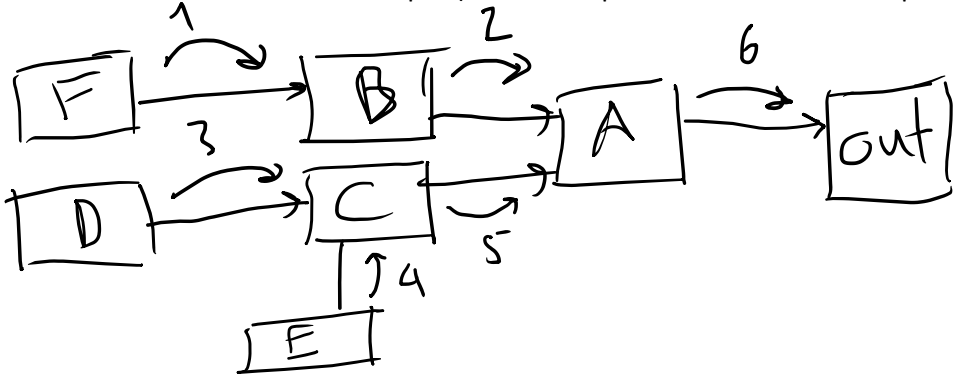
For a given frame in time t, evaluating the position of our output signal is as simple as evaluating the tree. Each node evaluates the outputs of its input nodes with the time parameter t. The evaluation is then passed on, until an input node is found. The following diagram illustrates the process:



It calls the connected modules recursively (numbers represent the call order):



And the information flows back to the output (numbers represent order of call completion):



An alternative would have been for components to write to their output nodes, "pushing" data forward. Along with buffers in each component, this would form a stream of data from the input nodes towards the output nodes. This is very much an imperative way of doing things. While it has nothing innately wrong with it, it would have gone against my original goals of paradigm exploration. The final program did end up with imperative patterns in it.

Feedback loops are possible in my implementation. Infinite recursion is prevented by having each node compute at most once per frame.

5.2 Filtering audio by frequency

Physical high-pass and low-pass filters work with capacitors and inductors. The behaviour of these can be simulated. As we work with sampled audio, it makes most sense to implement this in a time-discrete manner (calculation using timesteps). A simple formula for calculating the output voltage of an RC differentiator, which works as a high-pass filter, for example (2):

$$V_{OUT} = RC\frac{dV_{IN}}{dt}$$

Figure 2- output voltage for an RC differentiator

where R and C represent the resistance and capacitance of the circuit, and dVin is the change in input voltage during the change in time dt. Similarly, an RC integrator works as a low-pass filter (3).

With my background, this method was the easiest for to understand and implement, as it is directly analogous to electrical circuits which I have basic familiarity in. Other, more sophisticated methods may not be all that more complicated algorithmically, but I have chosen to forego them due of a lack of time.

6 Data structures

In most cases, little attention has been paid regarding the specific choice of collections (e.g, List vs, Vector), outside of what the surrounding collections happen to be, as this is not performance-critical to any part of the current program. A strong general pattern, however, has been to favour immutable structures over mutable ones.

In some cases, data types natural to the task have presented themselves: For example, it is natural to use a Map (or dictionary) for the NodeLibrary. A mutable set is natural for many applications in our synth, as we make abundant use of the host pattern. Due to its unexpected behaviour, however, it has in most cases been replaced by buffers where uniqueness is manually enforced.

Expecting it to be widely useful in building a component library, I implemented a queue with a maximum size: MaxSizeQueue. This structure stores an internal queue. Adding a new member, it checks the size of the queue. If this size is exceeded, the first member is dropped. As it stands, its use is limited to the Delay and AvgFilter components. The structure proves quite useful for modelling a “running buffer” (think a tunnel of cars), which are quite central to practical signal processing.

As for the **ModularSyhnth**s themselves, we create the linked structure between the components ourselves using a structure not unlike that of a linked list. In ModularSynth, while components are stored in a buffer, each component may refer to one or several components via its parameters. This creates a directed graph, which enables us to run the sound generation algorithm detailed in 3.3.

7 Files

In the final feature set of the program, the only use of files is in saving and loading synth configurations. I have opted to use a json format, as it fairly easy for a human to read and edit, as well as easy to implement serialization and deserialization for. This has been done using the circe library. Any class that requires

custom serialization has a given encoder and decoder in its companion object. The program does not give the save files any extension.

Refer to **SynthFileExample.txt** for a more detailed explanation of the file structure.

8 Testing

Fairly soon after starting the project, I realized that the original rigorous testing plan would not be compatible with the time limitation and scope of the project. I opted for a more "ad-hoc", manual style of testing, trying out what I thought would be the most common sources of errors as I built the program. I have relied more on choosing patterns that prevent most the most common errors (e.g., lift for off-by-one errors, and Option for NullPointerExceptions, choosing immutability where applicable etc.). This is a naïve approach, one that may still work with this scope, but not at a much greater scale.

The focus has been on systems testing and has mostly been done through the user interface. The most prominent form of testing has been manual testing of all the aspects of the intended use, in the form of building the demo synths. The little amount of unit testing that has been conducted was prompted by bugs found on a system level. All in all, testing conducted against atypical scenarios has been rather lacking. A further complication is the fact that audio can be hardware specific. The program as seems work well, but the lack of unit testing means we cannot consider any part of the program production ready. "If you haven't tested it, assume it to be broken".

One advantage that the nature of the program brings to testing is the ease of error detection: It is fairly to tell when a generated sound is indeed a sawtooth wave as opposed to a something else. The human ear is good at detecting a sound that is "off". Of course, subtle errors are subject to confirmation bias.

The GUI itself offers a nice testing environment: The workspace allows for a fast way to assemble different synth structures, as well as the ability to save, load and retest them using a modified program, enabling a certain level of automation. One danger that reliance on the GUI introduces is the lack of comprehensive testing before a functioning GUI is built. In my case this was quite late in the project. As such, testing has concentrated on the last weeks of the project.

The final build has been at least minimally tested on a variety of hardware has been available to me: three different machines and one MIDI keyboard. The test consisted of the following use case:

1- The user opens the synthesizer program. At this point, the user does not have a controller connected.
2- The user clicks the "load synth" button to load a synthesizer they've been working on.
3- The user supplies the synth configuration file location to the program via the graphical file chooser. The user clicks "open" button or similar on the file chooser window.
4- The user connects a MIDI controller to their computer. They click the "MIDI input" button on the toolbar and select the controller they've just connected.
5- The user plays a note on their MIDI controller.
6- The user stops playing a note.
7- The user tunes a parameter on the synthesizer.
8- The user adds a new component and connects it to the rest of the graph. The GUI node graph is updated, and a new ModularSynth object is constructed based on the updated graph.
9- The user clicks the file/save button and specifies the location to the file chooser window.
10- The user closes the program by pressing the red x.

Younes Elberkennou
1008920

9 Known bugs and missing features

As discussed in part 1, All other basic requirements have been fulfilled. One notable that was heavily present in the original plan (UML), but not the specification, is the command stack which would have allowed for common editor actions such as undo and redo. This has not been implemented.

Known bugs:

- The program has been observed to interfere with the functionality of other audio applications.
- Using the app via a virtual desktop, the zoom functionality does not work.
- When holding a key pressed on a midi controller, and disabling the device in the app, the note continues playing (no note-off message is transmitted). To circumvent this, press another e.g., on the computer keyboard. This could be a hinderance in implementing polyphony.
- Some components, such as LowPassIIR, could acquire an invalid state with the right combination of parameters. This led to output being cut off. Deleting the component solves this.

10 Three best sides, and three weaknesses

To get a feel for what the program can offer, I recommend checking out the demo files (See appendices).

Three best sides:

- A full-on domain-specific language has been implemented! The program is, in essence, both a development environment and an interpreter for this specific type of synthesizer.
- A functioning visual scripting environment for said language. I am fairly pleased with the feature set that the editor offers.
- Great separation of concerns in source code and logic: e.g., save files have separate parts for logic and GUI information. For example, it would not be all that difficult to rip the current GUI implementation out and replace it with another one.

Three weaknesses:

- The DSL has no guaranteed type-safety.
- GUI a bit “stiff”, many things may be unintuitive (some by design, e.g., when connecting two sockets, a continuously following line would have added a fair amount of complexity to the program)
- Bad operability with other audio applications in the background. For example, it has been observed to hog MIDI resources from other applications.

11 Deviations from the plan, realized process and schedule

This table outlines the realized order of implementation, in two weeks’ increments:

Week	Time estimate (h)	Features implemented
1-2	15	A working project setup, a basic farme for the keyboard midi controller. Partial implementation of synth logic
3-4	25	A bare-bones synth runtime capable of generating sound. Basic implementation of the synth runtime, as well as core sound generation algorithm: a working component node tree, that can calculate a value for a sound frame. Basic synth components
5-6	12	Working computer midi keyboard, groundwork for using external controllers

7-8	35	GUI implemented to a great extent, not yet coupled to synth logic. Ability to use external controllers
Last weeks	>50	Finishing up GUI, Coupling between GUI and SynthLogic File reading/writing: Serialization, Loading and saving configurations. Fleshing out the component library, polishing and testing the user interface. Removing bugs. Refactoring code and adding a layer of safety.

In retrospect, parts of the original schedule have proven highly unrealistic (specifically, early weeks and the estimated workload of some of the features, chiefly the GUI). I was, in many ways, already aware of this in the planning stage, hence the copious amounts of "dead time", as well as the inclusion of features that were not part of the basic requirements. Astonishingly, the final development time seems to match the planned time extremely well (approx. 130h), though it is hard to tell how much time has sunk into the project, especially during the last weeks. Even the original planned order of implementation was realized, the difference being that work ended up being significantly "end-heavy".

Perhaps the best decision in the planning stage was to group features by importance. When getting overwhelmed, this has allowed me to quickly get back on track working on features that are necessary.

12 Final evaluation

In general, I am happy with the outcome of this project. I have managed to create a working degree a visual programming environment, a sort of domain-specific language, and even recreated some familiar sounds with it. The core feature set of the program is minimal but works well, and the real-time performance is of no issue, even for nontrivial synthesizers. The need to speed up development towards the end of the project has, however, incurred a fair amount of technical debt. The different choices and solutions to problems encountered during development are likely suboptimal. This was a conscious trade-off, as I could not have completed the project otherwise. The core architecture, however, lends itself well to change. We can consider the planning phase a success. It is, at the very least, the most rugged piece of software I have ever built.

As for points of improvement and further development, the general plan already presents a great variety of useful features. As of now, the user interface is rather minimal. Continuing development, the greatest quality-of-life improvements would relate to quick actions common to all types of editors: copy, paste, cut, undo, redo etc. Implementing the command stack would, as such, be the next logical step.

In practice, the biggest hinderance in creating rich sound has proven to be the lack of polyphony. In principle, implementing this should not be too difficult. In practice, however, technical debt may need to be paid: while implementing the synth structure as well as the different components, the assumption has been made that a single value will be computed for a single sample, to make implementation easier. However, many of the other advanced features described in the project plan would be fairly easy to implement.

Reflecting upon what I should have done differently, I'm not sure it would have been anything. One of the most important goals of this project is the ability to design and deploy a program, as well as gain practical experience in how choices, both in design and implementation, affect the development cycle and the outcome. Mistakes I now made only add upon my abilities as a designer and programmer. The ambitious plan has proved itself a blessing in more ways than one: I have learned much from a wide variety of fields, spanning among others GUI development, signal processing and even type theory. I have gained a new confidence and enthusiasm for developing my own personal projects. I have also, for the first time, witnessed first-hand many of the fallacies one can make in the process of software development. All in all, I would say the course has been wildly successful in its goals. Even the gruelling process of creating three

separate project documents (in far more detail than would be sensible in any project of this scope) has served its purpose: The quandaries to which I have spent time giving answers have, in many ways, helped me find an identity as a developer.

If, however, I would now re-embark on a similar project, the first thing I would keep in mind is to restrict the scope to only that which is necessary. To build the application around one core idea, and maybe let it grow from there. This would allow me to focus on the qualitative aspects of software engineering: documentation and testing, which are commonly described as the mark of a professional.

Another point of improvement, perhaps for the course is to reduce the scopes of the project documents in favour of a prototyping phase. As far as I understand, it is essential step when embarking on any nontrivial software project.

One thing I would have wished is to learn a sensible testing strategy that I could use on my own, when building personal projects. The wide scope and restricted time led to me estimating that thorough testing would not be worth it. The scope here is restricted to the point that no greater trouble followed from this. However, this is not a sustainable philosophy. The one thing I have learned, if anything, is how far I am yet to come as developer if I am to build software for the use of others.

13 Bibliography

1. **Epic Games.** Material Editor Reference. [Online] [Cited: 05 11, 2023.] https://docs.unrealengine.com/4.27/en-US/RenderingAndGraphics/Materials/Editor/.

2. **ElectronicsTutorials.** RC differentiator. [Online] [Cited: 20 02 2023.] https://www.electronics-tutorials.ws/rc/rc-differentiator.html.

3. **RC integrator.** ***Electronics tutorials.*** **[Online] [Viitattu: 12. 05 2023.] https://www.electronics-tutorials.ws/rc/rc-integrator.html.**

14 Appendices

14.1 Extra material

In the project file, the folder “DemoSynths” contains a variety of synth configurations that demonstrate different techniques and sounds. To witness the capabilities of the synth, I recommend the “80sPadIsh”, “ChillyWind” and “TunableKarplus” demos.

The file “SynthFileExample.txt” holds an explanation for the file structure used to save the synths. You can inspect the demo files, too, in order to get an idea of the file structure.

14.2 The Oscillator component – type parameter

The “type” parameter of oscillator dictates the waveform generated:
0: sine
1: square
2. saw
3. noise
4. triangle
Going beyond 5 or under 0 will make the list warp around.