Assignment 2

Team number: 30
Team members:

Name	Student Nr.	Email
Sammy Fehr	2835488	s.f.fehr@student.vu.nl
Brian Vera	2835621	b.v.vera@student.vu.nl
Shota Goginashvili	2703632	s.goginashvili@student.vu.nl
Jose Gutierrez	2835470	j.l.g.gutierrez@student.vu.nl

Summary of changes from Assignment 1

Author(s): Brian

After receiving feedback from our TA Jacob we have found areas of improvement in our features. (1) Feature 1 Clarification and CRUD redistribution. (2) Feature 2 clarification. (3) Feature 3 implementation specification.

- Feature 1 will have **CRUD** removed and developed into **a separate feature** as proposed by Jacob giving us an even 4 features or 1 for each member.
  - The member responsible for this new feature 4 will be **Sammy**.
- Feature 2:
  - Changes for assignment 2:
    - User is able to choose **how long a note is played** for via a **slider**
    - User is able to control the tempo (frequency) of a beat/ instrument in terms of beats per minute using a slider
    - User is able to control "**velocity**" via a **slider adjusting volume of specific note**
- Feature 3:
  - This feature will be a **GUI**. More specifically, it will be a very simple version of an MPC. Our version will have 16 buttons for instruments, and a small "screen" for a beat sequencer, with sliders for Tempo, Velocity and other features as described in Feature 2.

Package diagram

Author(s): Sammy, Brian, Shota and Jose

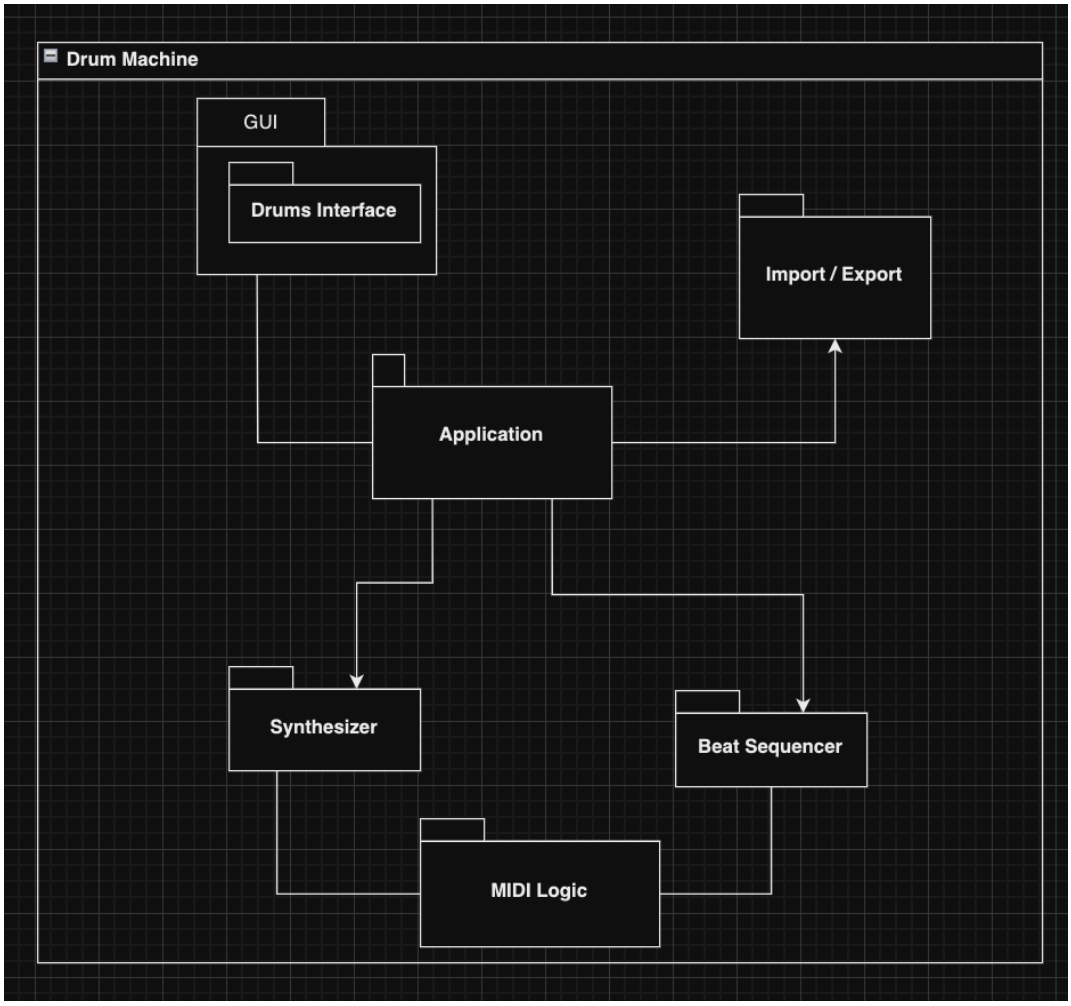

Figure 1. Package Diagram

Our package diagram, much like our drum machine, is quite simple - Our application package has access to everything in order to keep the program going. The MIDI Logic contains the building blocks of the MIDI system, broken down later in the class diagram. The beat sequencer deals with creating, recording and loading sequences as well as playing them, while the synthesizer plays the sounds. The application also has access to import/export methods to satisfy our first feature. Finally, the GUI takes care of the user-facing part of the app, and has a dedicated package for the “Drums”, i.e. the buttons responsible for playing the associated sound.

Class diagram

Author(s): Brian and Shota

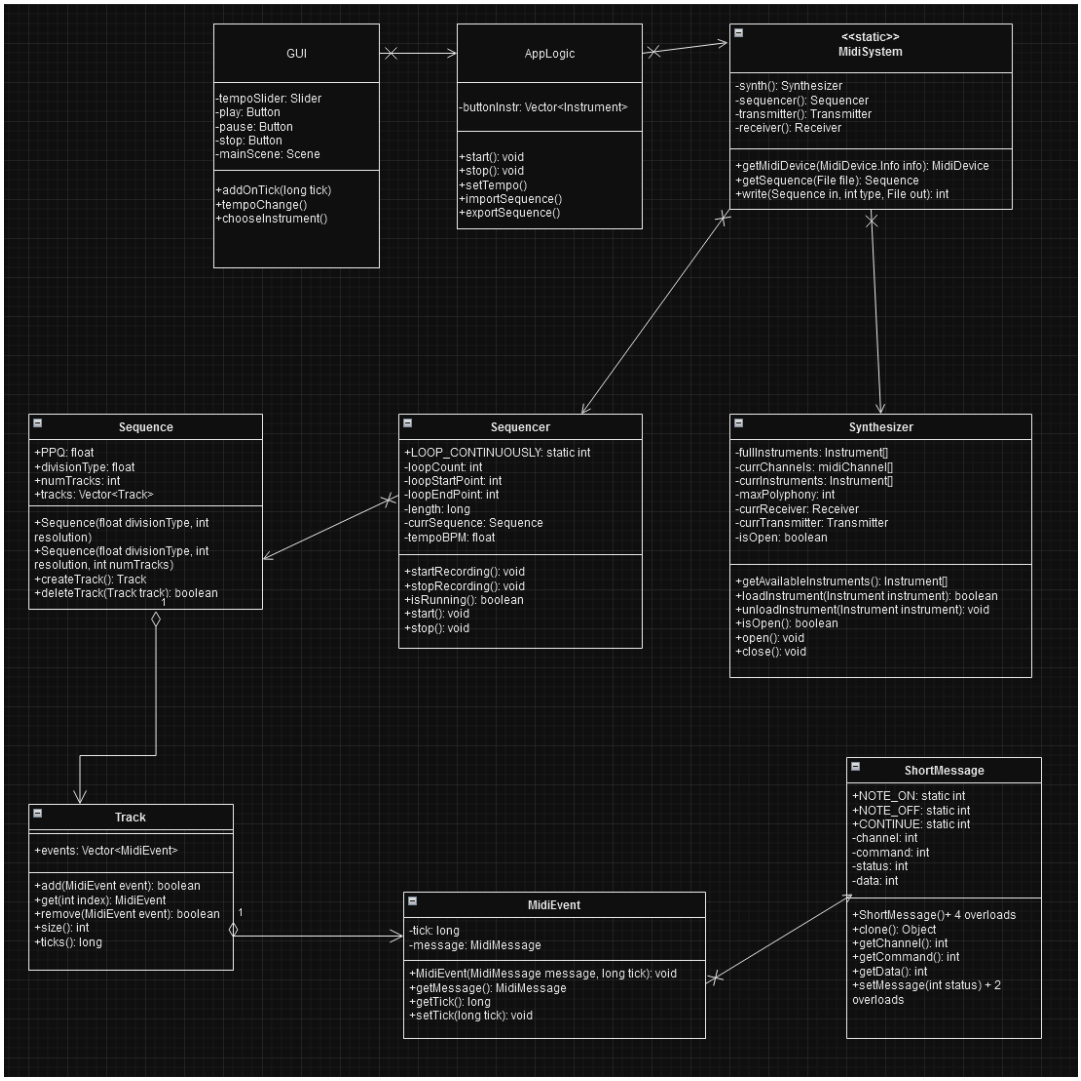

Figure 2. Class Diagram

Most of the classes in the above diagram come from the MIDI package in Java, which simplifies the inner workings of organizing and playing music from existing files or creating our own.

ShortMessage

This class is the building block of MIDI tracks. A ShortMessage is a MIDI message containing a status byte (NOTE_ON, NOTE_OFF, etc) and two data bytes. It has several constructors for setting a message. We conceive this class as the “note”,

containing data about the sound. The class has a binary association with the MidiEvent class, with each MidiEvent holding one MidiMessage, having access to all of the attributes and operations of the message.

MidiEvent

The MidiEvent class is very simple - it simply contains a MidiMessage (in our case a ShortMessage) and a corresponding time-stamp expressed in “ticks”. The objects of this class represent the MIDI event information stored in a MIDI file or Sequence. MidiEvents are aggregated into a track.

Track

Our Track class represents a MIDI track, which in turn is an independent stream of time-stamped MIDI data, being the middle level in the hierarchy of data played by a Sequencer: Sequencers play a sequence, which contains tracks, which contain MIDI events. Unlike other classes, since a track is controlled by and stored in a sequence, they’re only created by Sequences and therefore do not have a constructor. This class has methods to add and remove MidiEvent objects, as well as information such as size about itself. Tracks are aggregated into a Sequence.

Sequence

Our Sequence class contains musical information to be played by a Sequencer object. In our case, it simply contains tracks and timing information. It has two constructors, one without tracks and one with, and deals with creating or deleting tracks. It has an association with Sequencer to allow for usage.

Sequencer

A Sequencer deals with playing back a MIDI sequence. It contains the current sequence, tempo and details about start and end points in the loop. It also has several operations to start and stop a sequence, record a new one or change the tempo. In essence, this is our sequence player. It has a binary association with our static MidiSystem class to access its functions.

Synthesizer

While the sequencer deals with playing sequences, the synthesizer generates sound. Essentially, whenever one of the channels of the Synthesizer receives a noteOn message, it generates sound. The synthesizer will not be implemented by us, but rather imported from the MIDI package, as originally it is an interface with several implementations. We have the class to simply show its functionality. It contains information about channels, instruments, and the maximum number of notes that can be played at the same time (maxPolyphony). It also deals with the inner workings of the MIDI package, using transmitters and receivers. The synthesizer class has a binary association with the MidiSystem class

MidiSystem

The MidiSystem is a representation of the system containing all the aforementioned structures. It contains a synthesizer, sequencer, transmitter and receiver. It has a function to write a sequence to a file, and thanks to the binary association, has access to all of the operations and fields of the Sequencer and Synthesizer classes. It has a binary association with our AppLogic class.

AppLogic

Our AppLogic class controls the entire application - with its associations, it deals with the operation of the entire MidiSystem, the export/import of Sequences and the flow of music. We have omitted many functions in the class diagram as they will simply be a way to “get” a function from classes such as the Synthesizer. It contains data about which instrument is loaded into which button in the GUI, which is binary associated with the AppLogic class.

GUI

Our GUI class uses the JavaFX library to implement a very simplistic graphical interface akin to a MPC. It has fields representing the buttons and sliders controlling the music, and the main Scene aka the main window. We only show a few operations as examples - addOnTick() deals with adding a “beat” when a user clicks on a “tick'' in the beat sequencer. The operation tempoChange() deals with changing the tempo, and so on. This class, similarly to AppLogic, will mostly call other classes' methods.

State machine diagrams

Author: Jose

Sequencer Class Finite State Machine

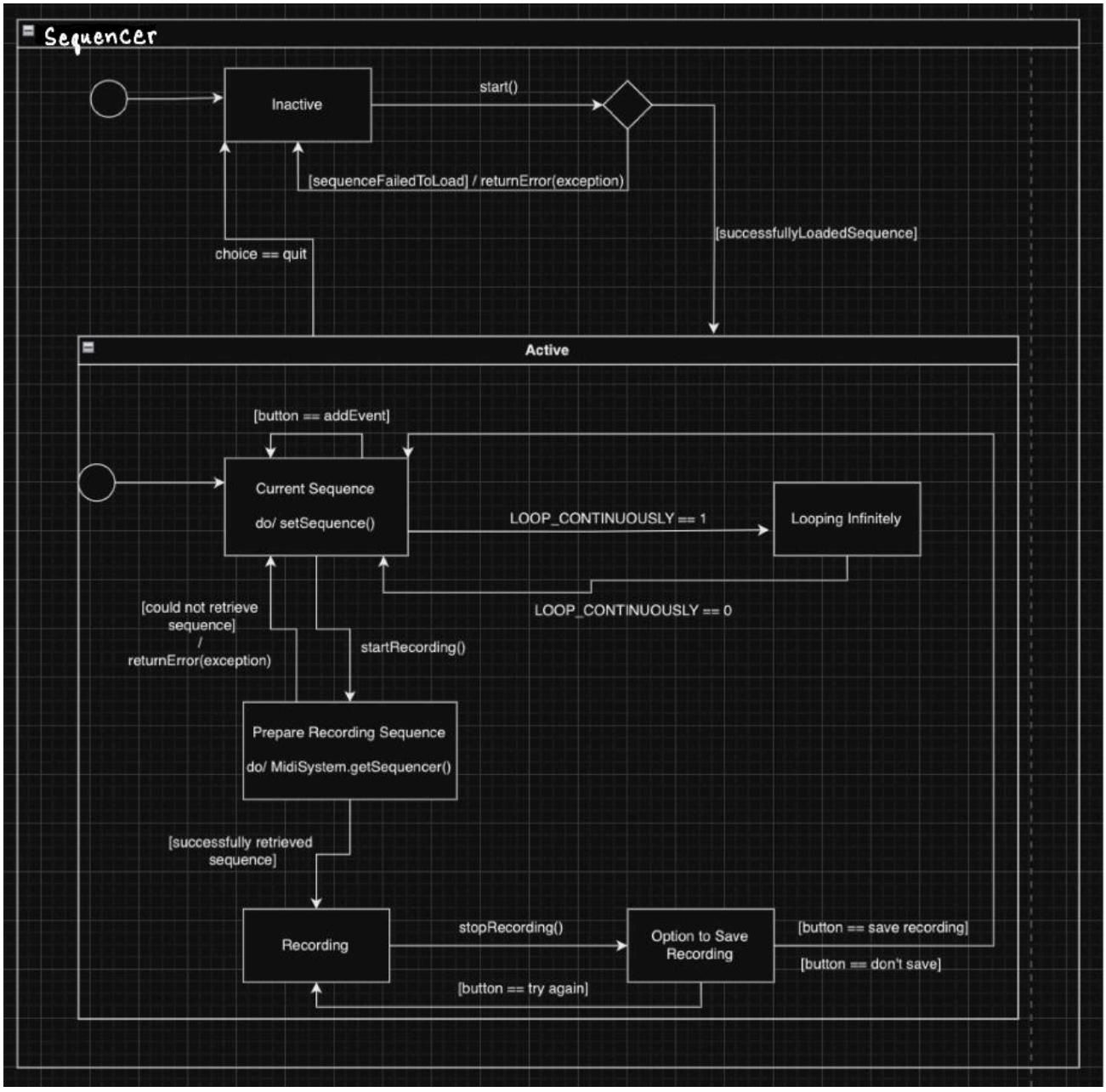

Figure 3. Sequencer Class State Machine Diagram

When the program is launched and before any sound can be played by the Synthesizer, a default **Sequence** object for the user must be created in the **Sequencer** class. After the default object is created, the class can go into an active state where multiple operations can be performed. The initial state of active is the **Current Sequence** state where the class is sitting idle, waiting for the user to either add an event to the sequence (a single note/sound), loop the currency displayed sequence endlessly, or begin recording a sequence to be saved. Anytime the class returns to the **Current Sequence** state, **setSequence()** is called to ensure that our sequence is up to date no matter what edits have been made. If a user decides to begin looping a beat, they are stuck in that loop until they decide to end the loop and return to the **Current Sequence** state. If a user decides to begin recording a beat, the sequence class must first ensure that a new **Sequence** object is able to be created using **MidiSystem.getSequencer()**. If the **Sequence** object is successfully created, the class moves onto the **Recording** state, else the class throws an error message and returns to the **Current Sequence** state. While in the recording state, the user is limited to only stopping the recording. After the recording has stopped, the user can try a new recording, putting it back into the **Recording** state, or the user can return to the **Current Sequence** state after saving/ choosing not to save the recording.

Synthesizer Class Finite State Machine

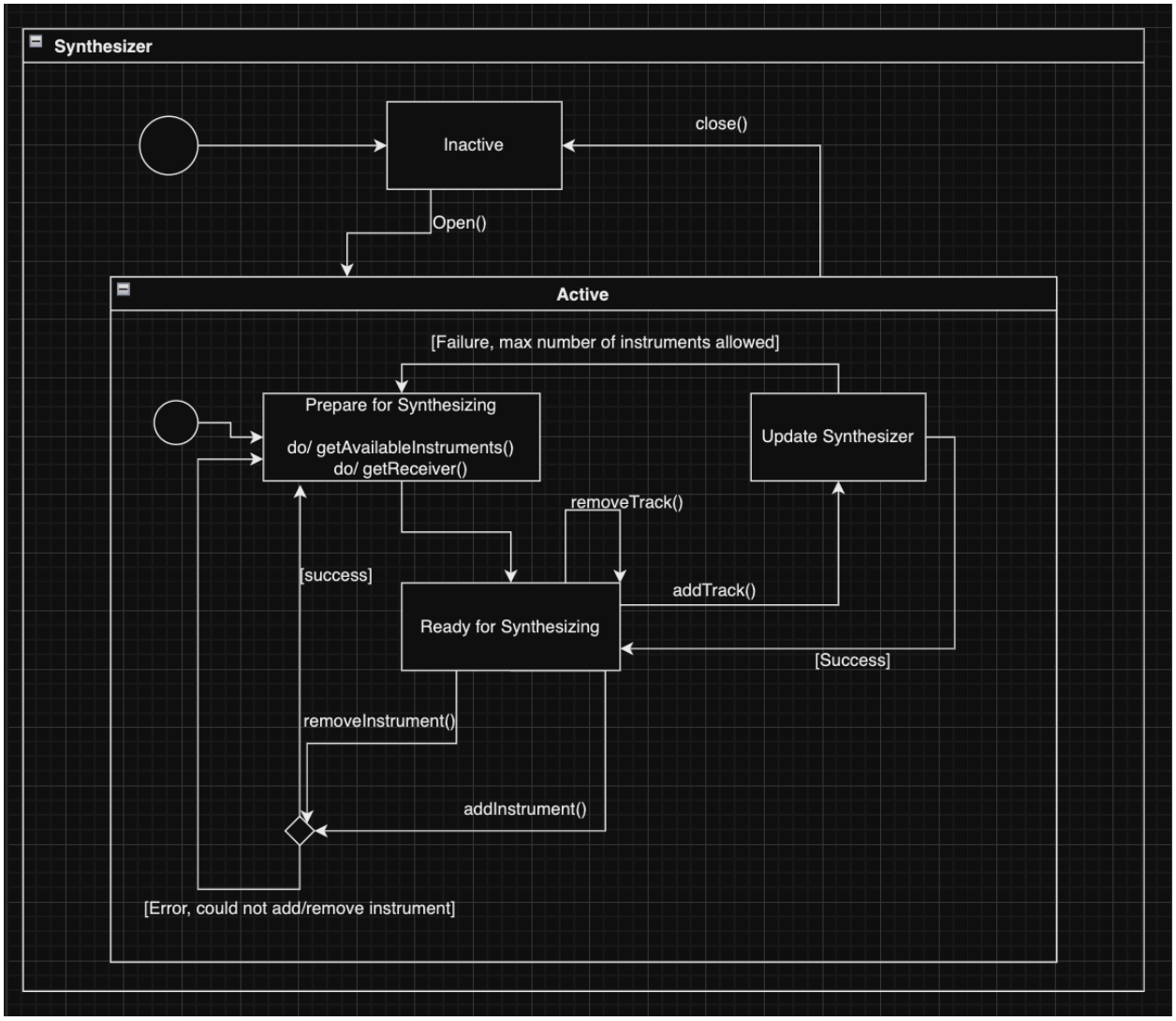

Figure 4. Synthesizer Class State Machine Diagram

The Synthesizer class acts as the conductor of our instruments and allows the separate tracks (individual instrument patterns) to be played at once, creating a beat with many instruments and components. Initially, the **Synthesizer** object must be declared open, allowing a wide variety of operations to be performed. Immediately after entering the active state, the **Synthesizer** must enter the **Prepare for Synthesizing** state and prepare to compile the tracks by calling **getAvailableInstruments()** and **getReceiver()**. Immediately after, the object then enters the **Ready for Synthesizing** state where the object's tracks list can be edited and instruments can be both added and removed. If a user decides to add or remove an

instrument, the synthesizer is updated, returning a success or error message, and returns to the **Prepare for Synthesizing** state so that the synthesizer's contents stay up to date. From the **Ready for Synthesizing** state, the user can also decide to delete or add tracks to the synthesizer. If a track is removed, the object simply updates the current tracklist and returns to the **Ready for Synthesizing** state. If an attempt to add a track is made, the object then enters the **Update Synthesizer** state. If the tracklist is not full, the object returns to the **Ready for Synthesizing state,** else the object fails to add a track and returns to the **Prepare for Synthesizing** state.

Sequence diagrams

Author(s): Brian and Sammy

User interface with Application Sequence Diagram

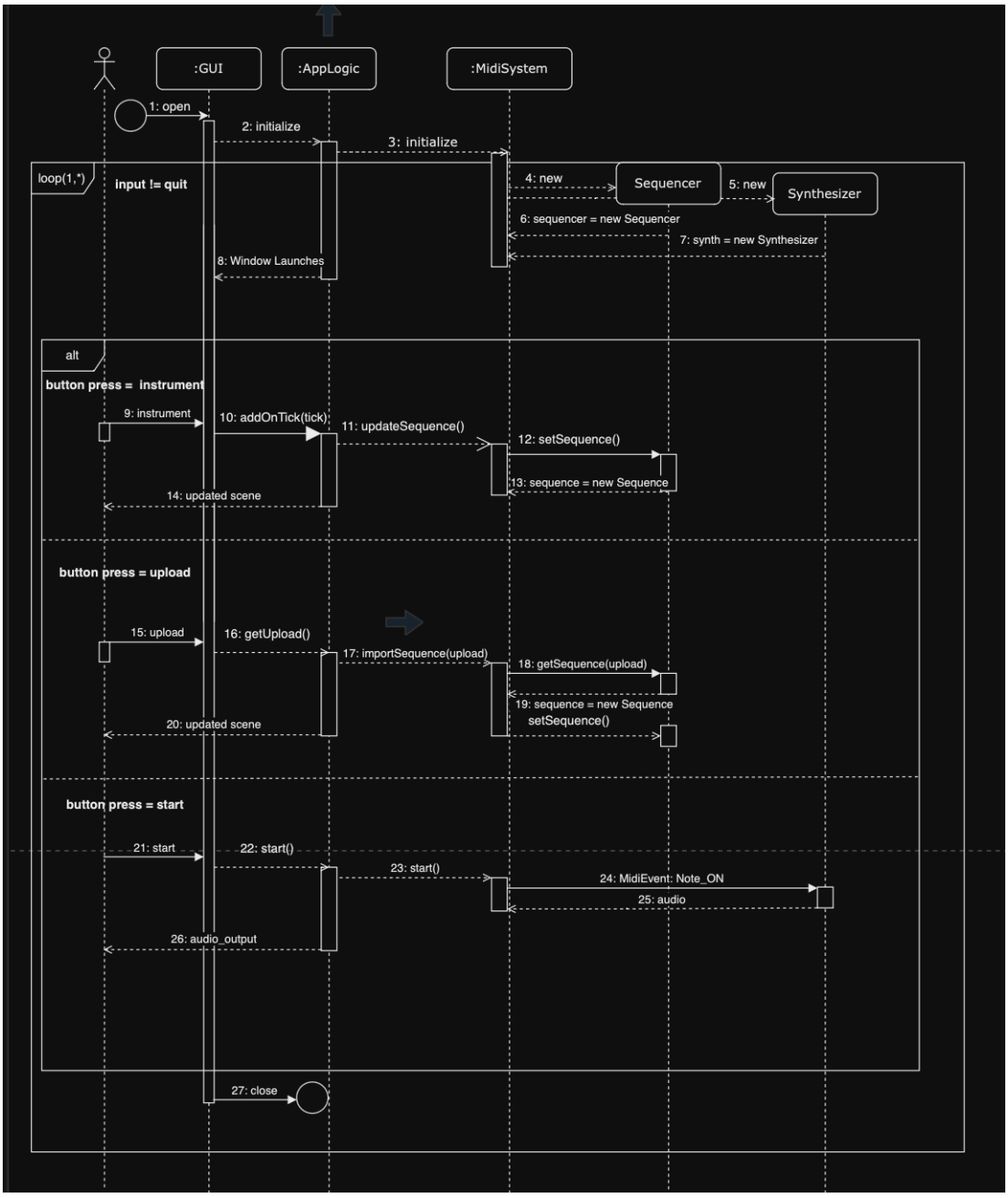

Figure 5. UI With Application Sequence Diagram

This diagram demonstrates the interaction between our User, GUI and MidiSystem classes. Many methods perform actions not described in the explanation

below; only relevant interactions are described. This interaction begins with the launch of our program noted as open. Messages (2,8) display the initialization of the application starting with initialization of the GUI and Applogic classes. The midiSystem creates a Synthesizer and Sequencer objects. When all is complete message 8 aka the window launch occurs and is available for the user. Then we enter a loop state through the Sequencer, only closed by the user quitting or closing the application, where the user is free to interact with the drum machine. This leaves the application with many alternative cases. For this diagram we chose to focus on some key aspects of the drum machine's usability. More specifically, when the sequence is manually altered by pressing an instrument button, when the user presses the upload beat(sequence) button and finally when the user presses the start button for the sequence.

For the **first case**, when one of our instrument buttons is pressed we must alter the visuals and sequence. This begins with the button press which is linked to the **addOnTick**() method, which updates our sequence using the updateSequence() method that calls the setSequence() method on the sequence object created by the **midiSystem**. Once this new sequence is completed the sequence concludes by updating the scene (contents of Window) of our application.

For the **Second Case**, when the user presses the upload button they provide a file to be sequenced. This is done through getUpload() which then calls **importSequence**() in the Midi. Once in the midi this upload is sent to the sequencer via **getSequence**(upload) which returns a new Sequence object that is then sent back to sequencer via **getSequence**() which replaces the current Sequence with uploaded one. Then just like case one the scene is updated to the new sequence.

For the **third** and final **case** described, when the start button is pressed we path to midiSystem through our chain of start() methods. All MidiEvents within the Sequence that are under that status **Note_On** then produce sound according to the logic defined by the current sequence, this includes channel, tempo and other specifics defined in midiEvent. This sound is returned through the **User** via their method of **audio output**.

Adding Events To A Track Sequence Diagram

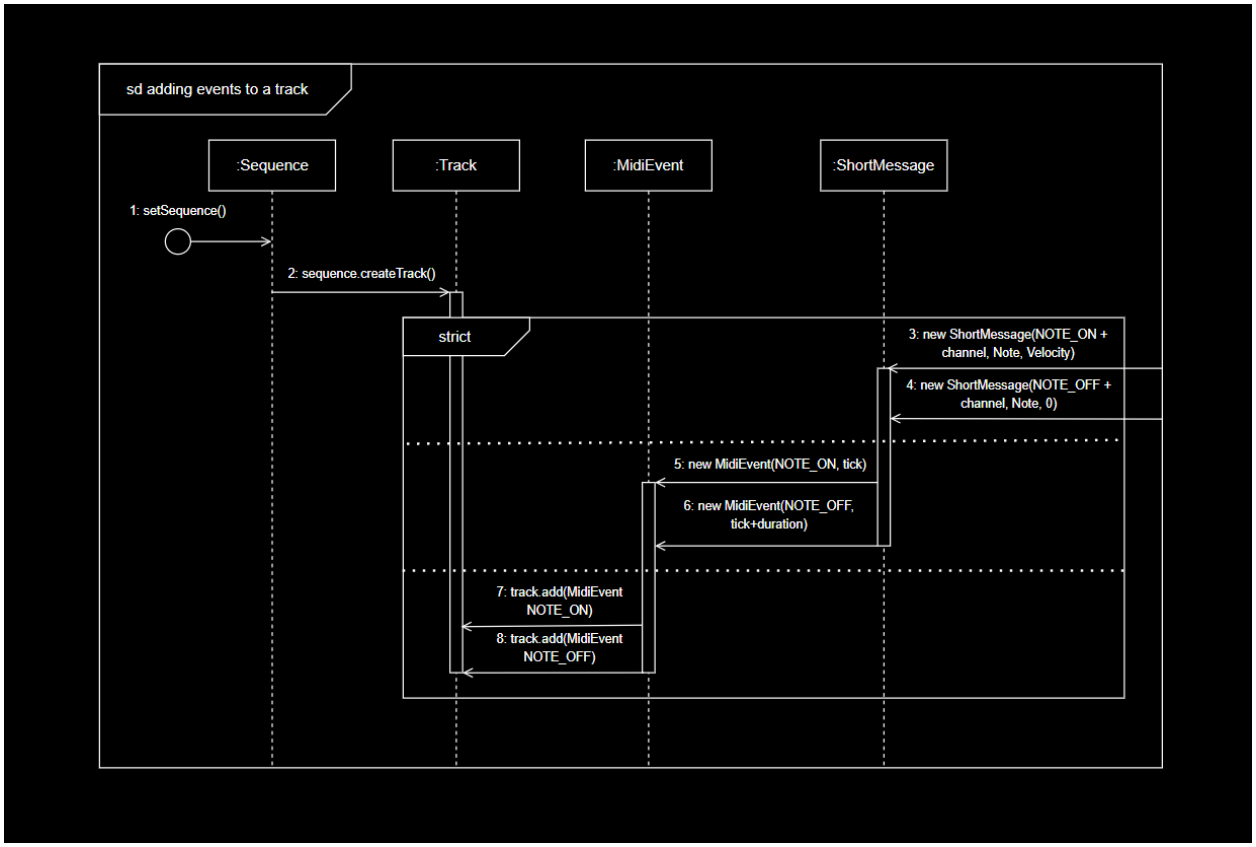

Figure 6. Adding Events Sequence Diagram

The sequence diagram in figure 6 depicts the sequence of methods which are needed to add a drum hit to a track stored in a sequence which is saved to the sequencer. This can be imagined as a single drum component being played and having the sound saved in the drum machine system. The role of the following classes are demonstrated in this sequence: Sequence, Track, MidiEvent and ShortMessage. The sequence begins with the (1) setSequence() method which sets the sequence upon which the sequencer is currently operating. A new track is then created using the (2) sequence.createTrack() method. These first two methods would occur automatically when the sequencer is initiated by the user, after that the user would be able to click on the drum machine and sequencer to create a new drum hit. The method calls that follow the initial two method calls demonstrate how a drum hit is added to the drum machine.

After the new track is created, two (3 & 4) shortMessage objects are created and stored in the ShortMessage class. The first shortMessage instance includes the NOTE_ON MidiMessage which indicates the beginning of the musical event, the second shortMessage instance includes the NOTE_OFF MidiMessage which indicates the end of the musical event. These new shortMessage instances also include information on the following parameters: the channel or drum component upon which the drum hit is played which must match between the two shortMessage objects, the note (which is not relevant to drum machines but would be used to indicate the pitch from 0 to 127 on a digital keyboard), and the velocity which indicates how hard a drum component is played. The shortMessage object including the NOTE_ON message can take anny velocity value between 1 and 127 while the shortMessage object including the NOTE_OFF message must take a velocity value of zero to indicate the end of the playing of the drum component.

The two shortMessage objects are then used within the MidiEvent constructor to create (5 & 6) two MidiEvents that are stored in the MidiEvent class. These constructors include the tick timestamp of the MidiEvents which indicate their position within the track. The difference in the tick value of the MidiEvent with the NOTE_ON message and the tick value of the MidiEvent with the NOTE_OFF message indicates the duration of the drum hit, the tick value of the second MidiEvent must be greater than that of the first. The MidiEvents are then added to the track using the (7 & 8) track.add(MidiEvent event) method. This indicates the successful addition of a drum hit in the Track class. Each time the user “hits” a drum component on the sequencer this process will be repeated, the structure of the drum machine will allow them to click the channel they desire and to adjust the duration and initial velocity of each drum hit.

Time log:

A	B	C	D
Team number:	30		
Member	**Activity**	**Week number**	**Hours**
Brian Vera	Asg. 1 Corrections, Package Diagram, Class Diagram, Planning	4	5
Sammy Fehr	Package Diagram, Planning	4	5
Jose Gutierrez	Package Diagram, Planning	4	5
Shota Goginashvili	Package Diagram, Class Diagram, Planning	4	5
Brian Vera	Class Diagram, Sequence Diagrams, Descriptions	5	10
Sammy Fehr	Sequence Diagrams, Descriptions, Editing	5	10
Jose Gutierrez	State Machine Diagrams, Descriptions	5	10
Shota Goginashvili	Package Diagram, Class Diagram, Descriptions	5	10
		TOTAL	60