Assignment 3

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

IMPORTANT: In this assignment you will fully model and implement your system. The idea is that you improve your UML models by (i) applying a subset of the studied design patterns and (ii) adding any relevant implementation-specific details (e.g., classes with "technical purposes" which are not part of the domain of the system). The goal here is to improve the system in terms of maintainability, readability, evolvability, etc.

Do NOT describe the obvious in the report (e.g., we know what a `name` or `id` attribute mean), focus more on the **key design decisions** and their "**why**", the pros and cons of possible **alternative designs**, etc.

Format: establish formatting conventions when describing your models in this document. For example, you style the name of each class in bold, whereas the attributes, operations, and associations as underlined text, objects are in italics, etc. Consistency is important!

Summary of changes from Assignment 2

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

Package Diagram:

- Corrected the syntax for fit java use.
- Added "<<import>>" where appropriate

Class Diagram:

- Added Design Pattern Indicators where possible
- Added relationship names for clarity

State Machine Diagrams:

- The main changes for the State Machine Diagrams arose from mistakes in following UML convention as well as making the diagram easier to read. For example, some actions were moved to states to be an exit or entry condition, making it easier to understand the role of a state. A common change for following the UML convention was updating guards to have brackets, and sometimes changing an action to a guard.

- There were no architectural changes in the State Machine Diagrams from Assignment 2.

Sequence Diagrams:

- We edited the format of the descriptions of our sequence diagrams to make them easier to understand, i.e. separating descriptions of important events, using bullet points and bold font.
- The strict operand has been deleted from the 2nd sequence diagram as it was not necessary. It was originally intended to indicate the strict order of messages, however, the order of messages is maintained without the use of the strict operand due to the lifelines being connected by consecutive messages with a clear order. The highlighted portion on the graph indicates the deletion of the strict operand.

Provide a bullet list summarizing all the changes you performed in Assignment 2 for addressing our feedback.

Maximum number of pages for this section: 1

Revised Package diagram

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

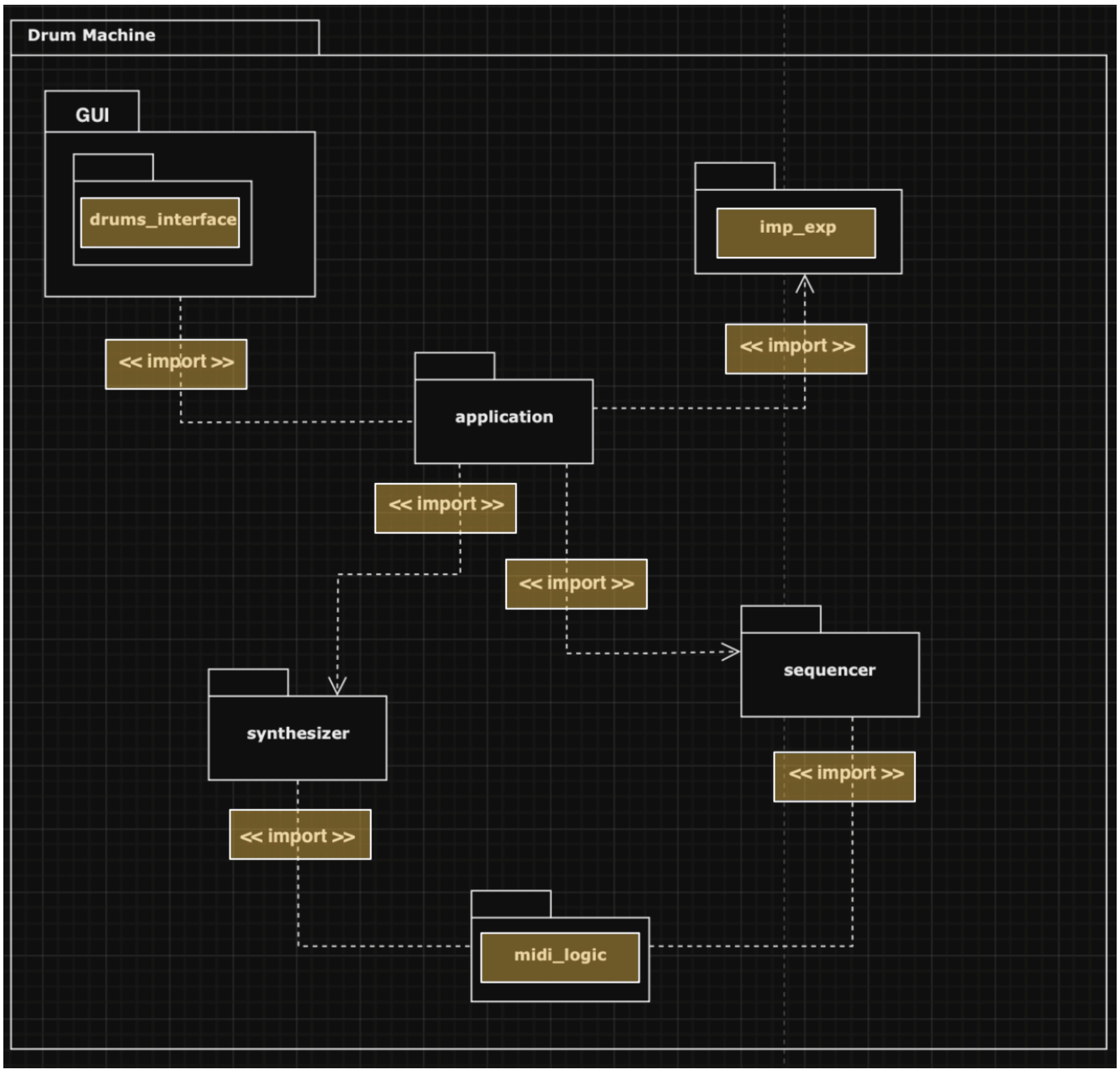
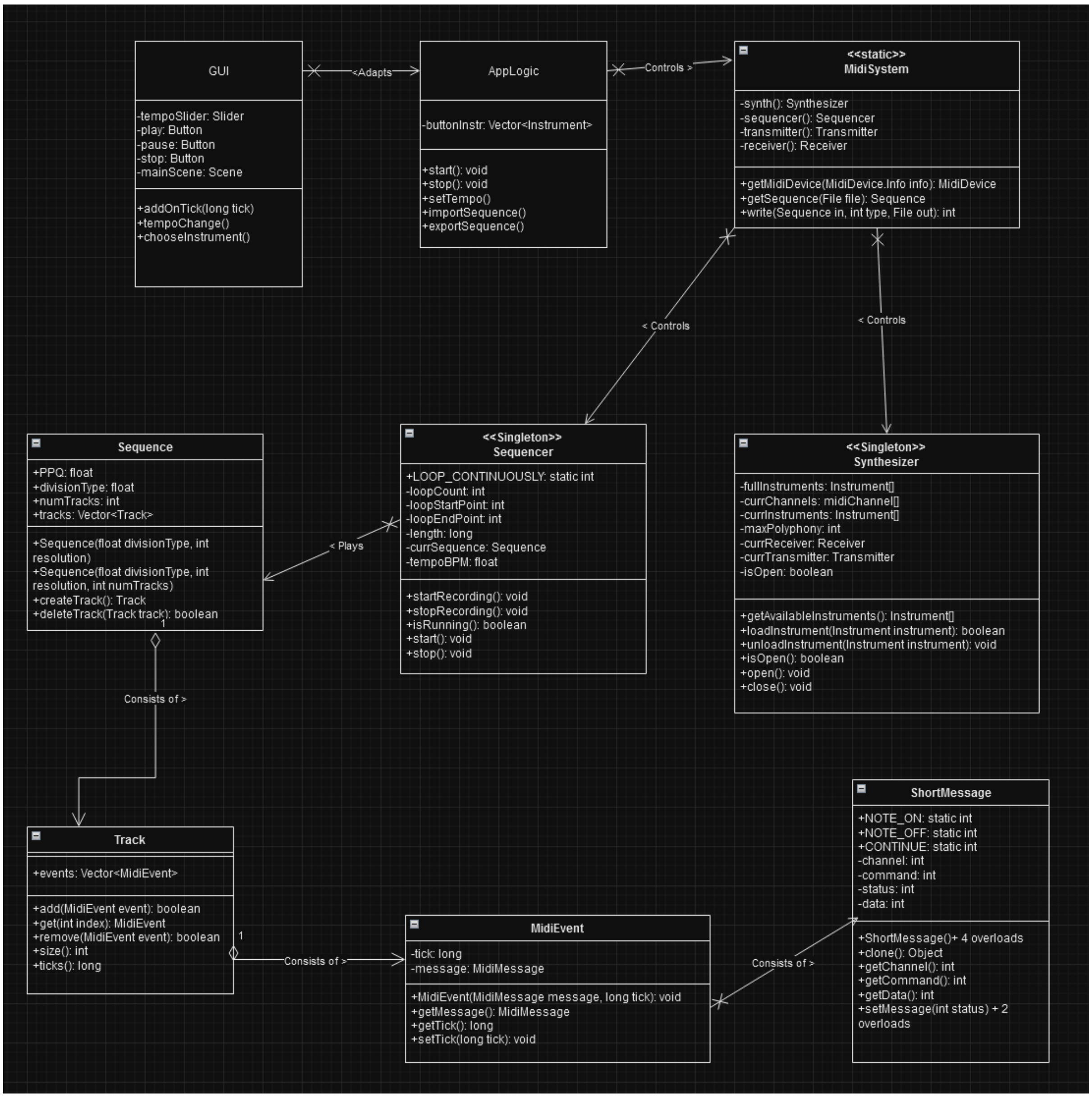


Figure 1. Package Diagram

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.

Revised class diagram

Author(s): Shota



The revised class diagram showcases the Singleton design pattern as explained below, as well as adds relationship names for more clarity.

Maximum number of pages for this section: 4

Application of design patterns

Author(s):Brian

	DP1
Design pattern	Singleton
Problem	We need our application to have exclusively single instances of a few high level classes such as synthesizer and sequencer as these are the basis of our operation. We need them to be able to respond to change immediately and controlled access through only this single instance.
Solution	These key parts fulfilled the description of a singleton immediately as they required exclusivity for controlled access. Theft will only be called once, the initialization, in our program limiting the number of instances to one.
Intended use	As seen in our first sequence diagram when initializing our application 1 instance of sequencer and synthesizer will be created and will be used for the duration of our applications active period.
Constraints	NA
Additional remarks	NA

	DP2
Design pattern	Adaptor
Problem	We need our GUI to have interconnectivity with our musical events, midi classes, as the events from the user interface needed to be translated into real instances of change for our sequences.
Solution	We created a method which would be given an index and would change the note from off to on based on this click. Through a Logic class dedicated to bridging our GUI and musical classes.
Intended use	As seen in our first sequence diagram when a user input is received our app logic defines this event into changes for our Midi system, synthesizer and sequence.
Constraints	The events for our GUI are limited by what a typical device offers like keyboards and a mouse/touchpad with no parameters other than the events occurrence.
Additional remarks	NA

	DP3
Design pattern	Chain of responsibility

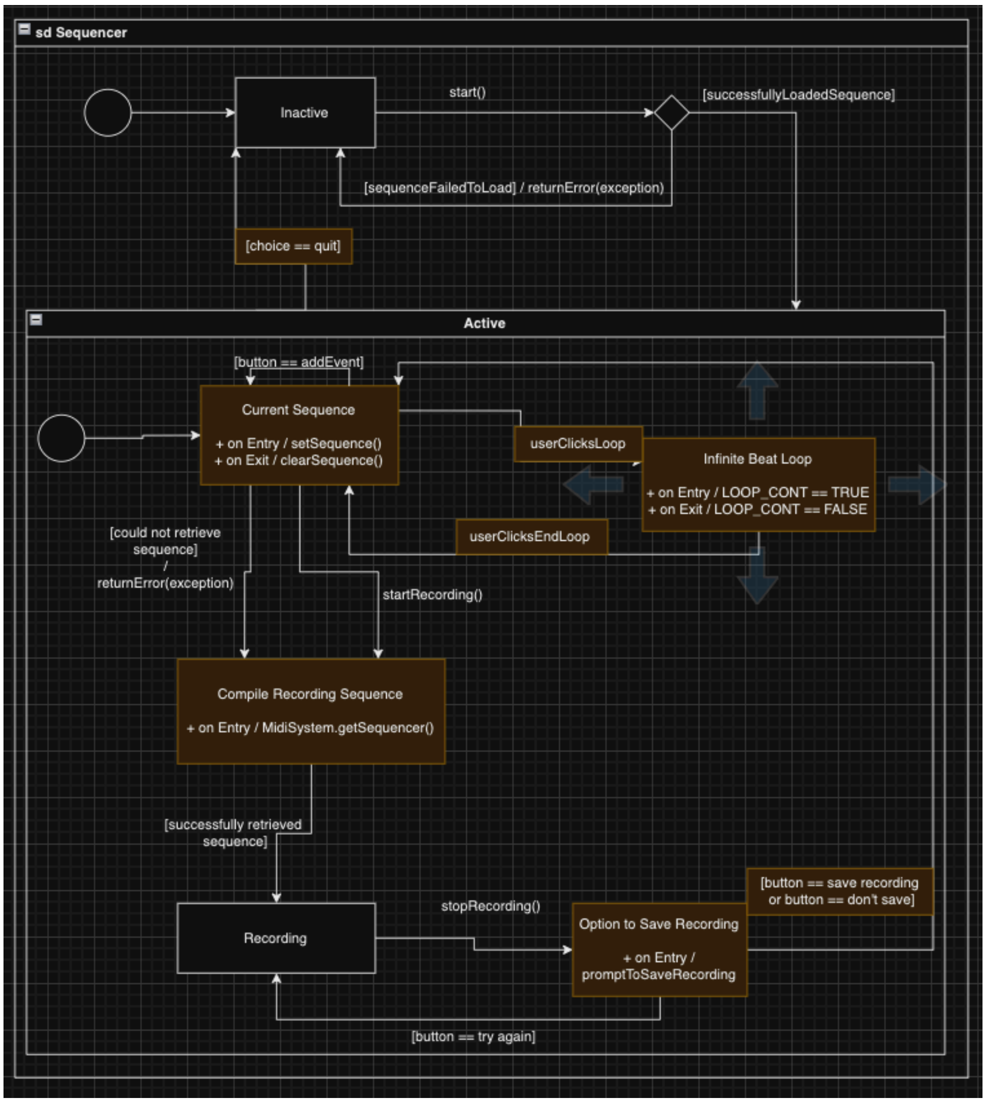
Problem	The musical classes used, the midi package, work in a chain as they build upon each other. The sequencer uses a sequence which is composed of tracks composed of midi events. Thus leads to a chain structure for any changes made to individual musical events e.x. Turning a note on or off.
Solution	The classes have methods enabling them to go down this chain and easily find and make necessary changes even at the bottom level of our hierarchy which is much needed for the usability of our project. Each class has a clear responsibility in the chain and only handles requests within its scope.
Intended use	As seen in our second sequence diagram when adding a track to our sequence we must go down the chain of midi classes in order to make these changes starting with creating a sequence if one does not exist, then track(s), then midi events and concluding with a short message.
Constraints	NA
Additional remarks	NA

Maximum number of pages for this section: 4

Revised State Machine diagrams

Author(s): Jose Gutierrez

Revised State Machine Diagram for Sequencer Class

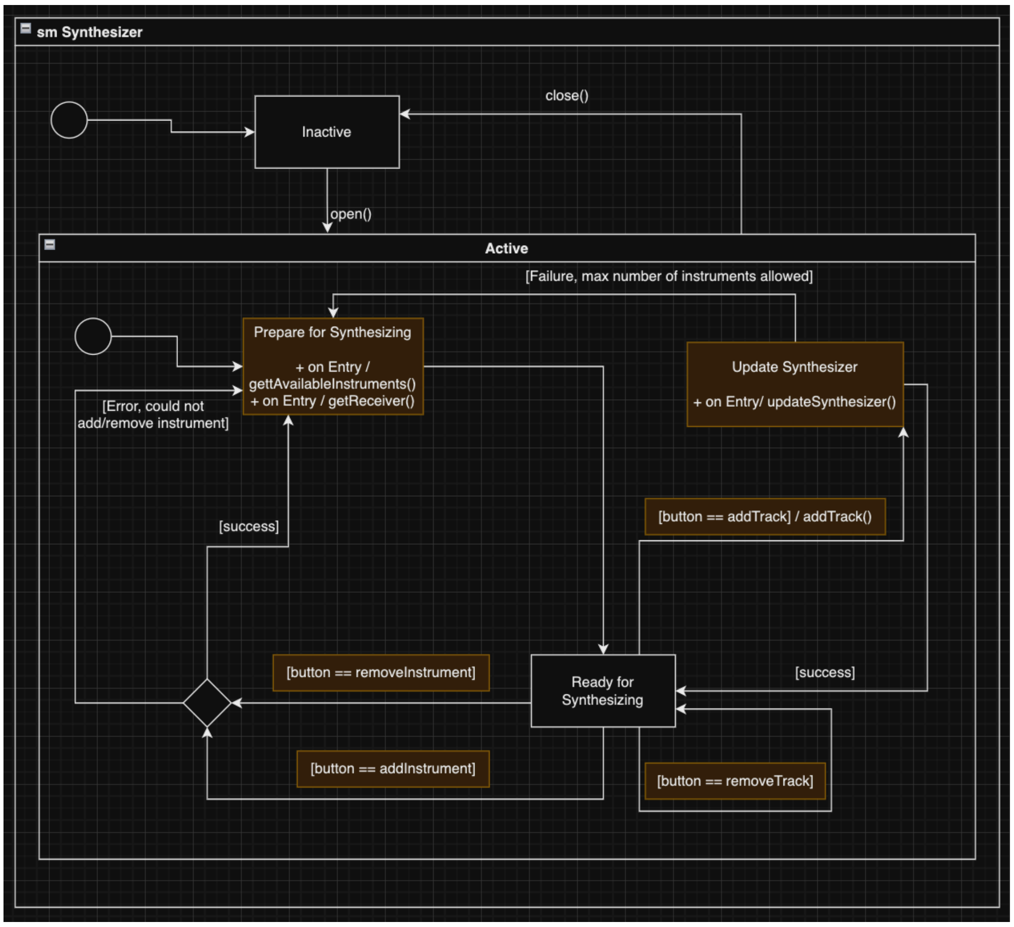


sd Sequencer
Inactive
start()
[successfullyLoadedSequence]
[sequenceFailedToLoad] / returnError(exception)
[choice == quit]
Active
[button == addEvent]
Current Sequence
+ on Entry / setSequence()
+ on Exit / clearSequence()
userClicksLoop
Infinite Beat Loop
+ on Entry / LOOP_CONT == TRUE
+ on Exit / LOOP_CONT == FALSE
userClicksEndLoop
[could not retrieve sequence] / returnError(exception)
startRecording()
Compile Recording Sequence
+ on Entry / MidiSystem.getSequencer()
[successfully retrieved sequence]
Recording
stopRecording()
Option to Save Recording
+ on Entry / promptToSaveRecording
[button == save recording or button == don't save]
[button == try again]

Changes made:

In the state machine diagram for the Sequencer class, the main changes arose from following proper and consistent conventions. More specifically, we added entry and exit events to the Current Sequence, Infinite Beat Loop, Compile Recording Sequence, and Option to Save Recording states. This made the purpose of the states more clear, as one can better understand how the object behaves when in a particular state. Many of these entry and exit conditions were derived from the "effect" portion of the "trigger [guard] / effect" convention used in assignment 2's Sequencer state machine diagram. Moreover, corrections were made relating to proper conventions. For example, "choice == quit" was changed to be a guard condition, “[choice == quit].” Besides changes made to make the diagram clearer, the architecture of the state machine diagram remains unchanged.

Revised State Machine Diagram for Synthesizer Class


sm Synthesizer
close()
Inactive
open()
Active
[Failure, max number of instruments allowed]
Prepare for Synthesizing
+ on Entry / gettAvailableInstruments()
+ on Entry / getReceiver()
Update Synthesizer
+ on Entry/ updateSynthesizer()
[Error, could not add/remove instrument]
[success]
[button == addTrack] / addTrack()
[button == removeInstrument]
Ready for Synthesizing
[success]
[button == addInstrument]
[button == removeTrack]

Changes made:

Similarly to the changes made for the Sequencer Class state machine diagram, the main changes for the Synthesizer class related to clarity in the diagram rather than major architectural changes. More specifically, the “Prepare for Synthesizing” and “Update Synthesizer” received entry events to make it easier to understand the purpose of the state and how it changes/interacts with an object. Additionally, the guards and effects were updated to conform to UML convention. Overall, the structure is the same but the diagram was updated to be easier to understand.

Revised Sequence diagrams

Author(s): Brian and Sammy

User interface with Application Sequence Diagram

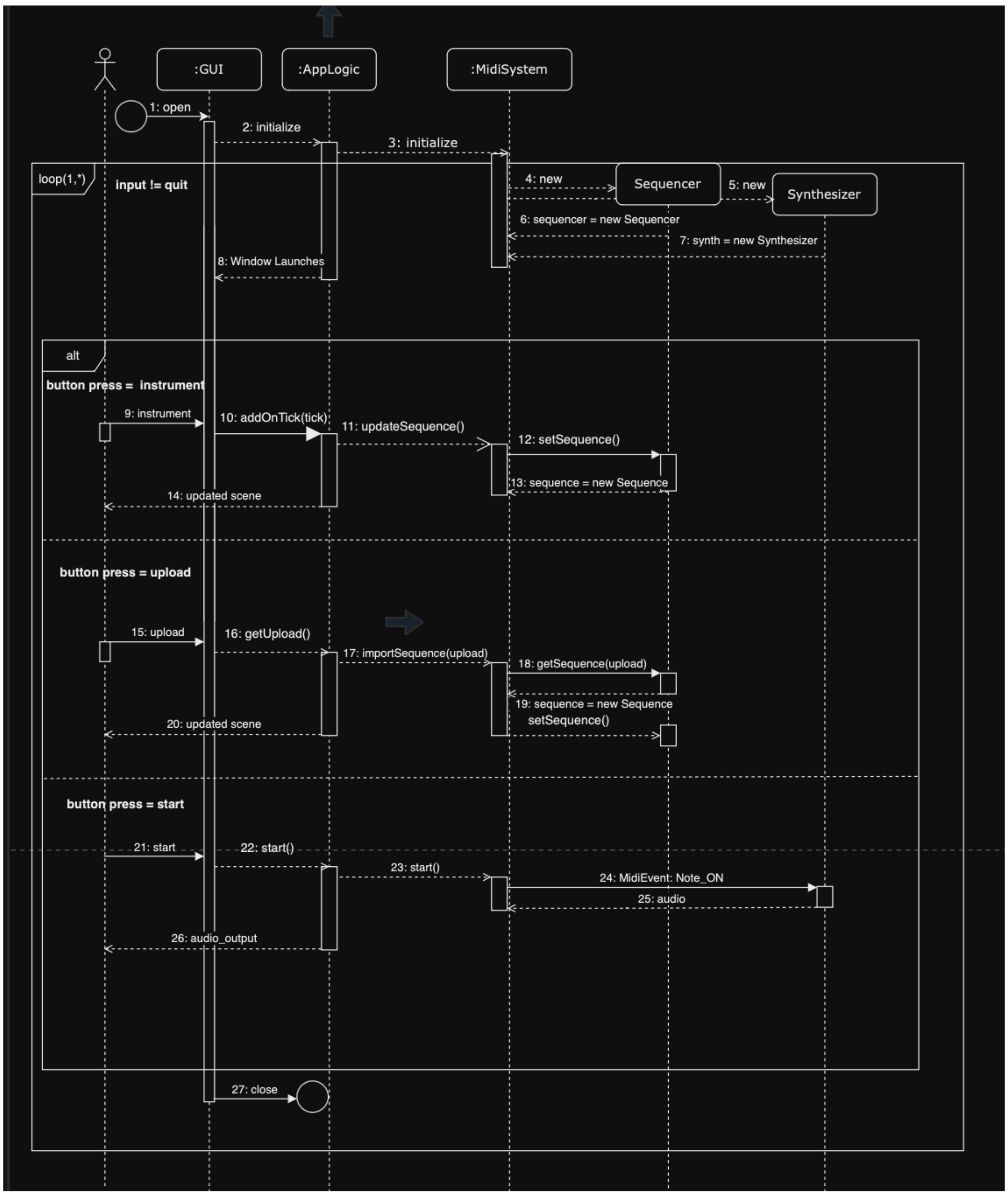


Figure 5. UI With Application Sequence Diagram

Overview:

This diagram demonstrates the interaction between our User, GUI and MidiSystem classes. For this diagram we chose to focus on some key aspects of the drum machine's usability, specifically 3 cases of use. For the **first case**, when one of our instrument buttons is pressed we must alter the visuals and sequence. For the **Second Case**, when the user presses the upload button they provide a file to be sequenced. For the **third** and final **case** described, when the user initiates the playback of the Drum Machine. Many methods perform actions not described in the explanation below; only relevant interactions are described.

Launch, Messages 2-8:

This interaction begins with the launch of our program noted as open. **Messages (2-8)** display the initialization of the application which initializes the **GUI** and **Applogic** classes. The **midiSystem** creates a **Synthesizer** and **Sequencer** objects. When all is complete **message 8**, the window launch occurs and is available for the user.

Loop, Messages 8-27:

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.

First Case, Messages 9-14:

Begins with the button press which is linked to the **addOnTick()** method, which updates our musical sequence. This in turn uses the **updateSequence()** method which calls the **setSequence()** method on the **sequence** created by the **midiSystem**. Once this new sequence is completed and updated the sequence concludes by updating the **scene** (contents of Window) of our application.

Second Case, Messages 15-20:

Begins with the **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.

Third Case, Messages 21-26:

When the start button is pressed we path to midiSystem through our chain of **start()** methods. All MidiEvents (individual beats) 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

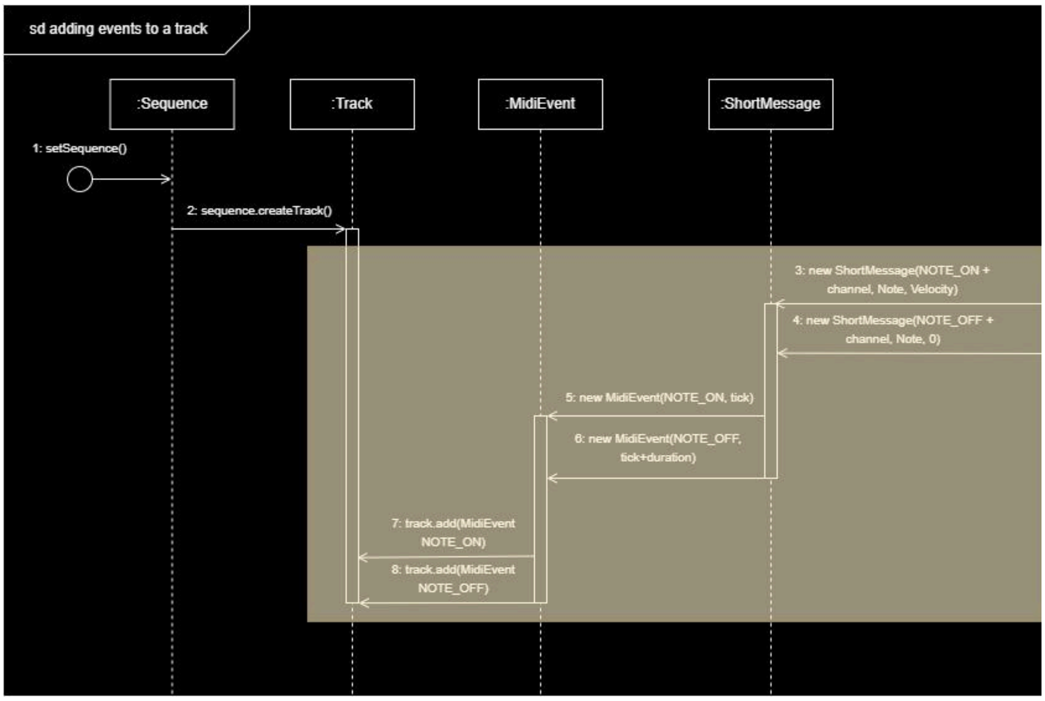

sd adding events to a track
:Sequence
:Track
:MidiEvent
:ShortMessage
1: setSequence()
2: sequence.createTrack()
3: new ShortMessage(NOTE_ON + channel, Note, Velocity)
4: new ShortMessage(NOTE_OFF + channel, Note, 0)
5: new MidiEvent(NOTE_ON, tick)
6: new MidiEvent(NOTE_OFF, tick+duration)
7: track.add(MidiEvent NOTE_ON)
8: track.add(MidiEvent NOTE_OFF)

Figure 6. Adding Events Sequence Diagram

Overview:

The sequence diagram in figure 6 depicts the sequence of methods which are required to add a drum hit to a track which is stored in a sequence that is saved to the sequencer. The following classes are accessed during this sequence: Sequence, Track, MidiEvent, ShortMessage. The steps of the sequence are elaborated upon below the Changes section.

Initialization, Messages 1 & 2:

The sequence begins with the setSequence() method which sets the sequence upon which the sequencer is currently operating. A new track is then created using the sequence.createTrack() method. With the sequence set and track created the user can then proceed to add drum hits to the drum machine. The Short Message Creation and Midi Event Creation sections below explain how the process of creating a drum hit works while also providing important details on the structure of the methods used.

Short Message Creation, Messages 3 & 4:

In order to create a drum hit two shortMessage objects must first be 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).
- The velocity indicates how hard a drum component is played.

The shortMessage object containing the NOTE_ON message can take any velocity value between 1 and 127 while the shortMessage object containing the NOTE_OFF message must take a velocity value of zero to indicate the end of the playing of the drum component.

Midi Event Creation, Messages 5 & 6:

The two shortMessage objects are then used within the MidiEvent constructor to create two MidiEvents that are stored in the MidiEvent class. These constructors include the tick timestamp of the MidiEvents which indicates 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.

Adding Events to the Track, Messages 7 & 8:

The two MidiEvents are finally added to the track using the track.add(MidiEvent event) method. This finalizes the successful addition of a drum hit in the Track class. Each time the user adds a drum component on the sequencer this process will be repeated. There will be a track for each 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.

Implementation

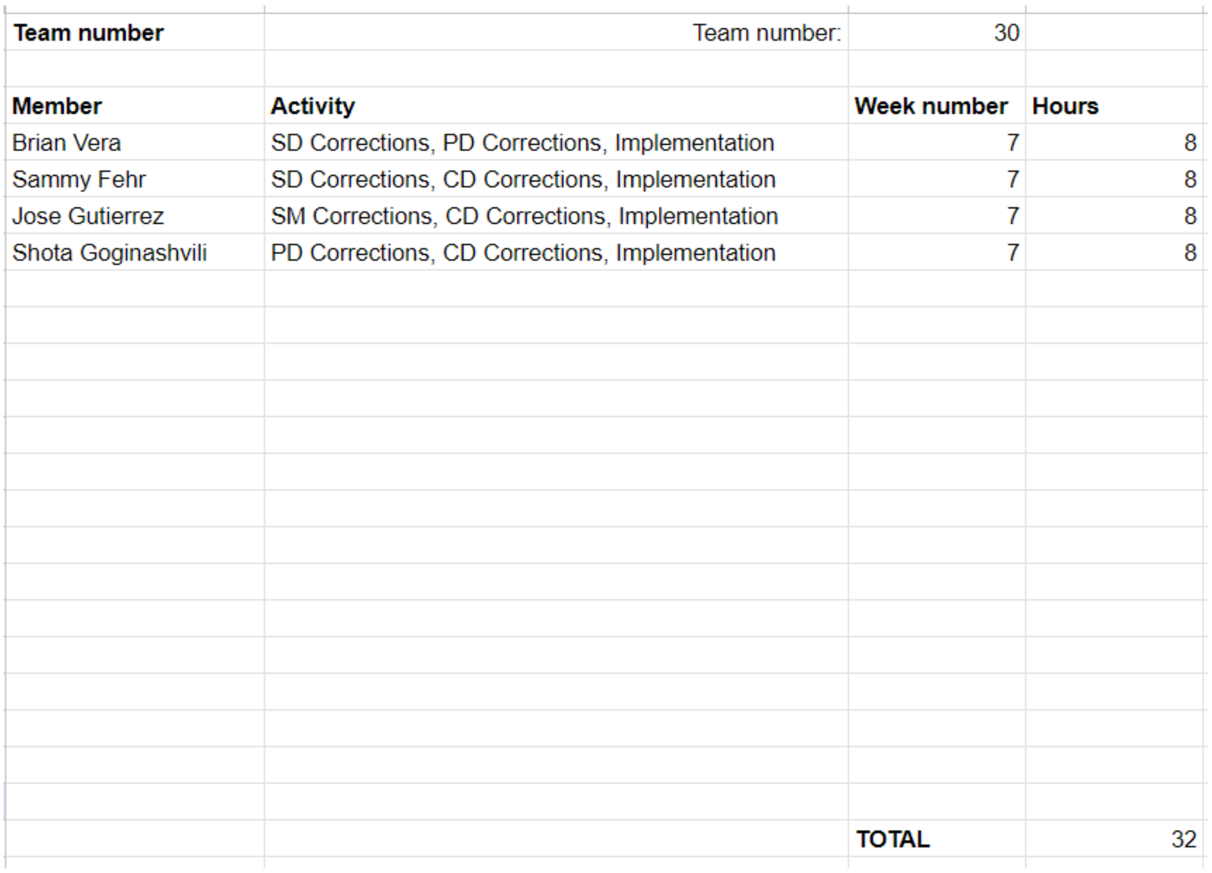
Author(s): Sammy, Brian, Shota and Jose

We adapted our UML models as closely as possible as the design was based on the documentation already provided by the midi package and javafx. There was already a semblance of design patterns visible in our UML models which created an ease in transition. We applied the simplest solutions for all issues we came across as mentioned above. We applied singleton, adaptor and chain of responsibility as our core solutions for the implementation. There was already a sign of these patterns in our models.
The Main.java file launching our application is located under /src/main/java/nl.vu.cs.software design.The jar file is located in /src/main.
Implementation was not fully completed and these files are not complete.

IMPORTANT: remember that your implementation must be consistent with your UML models. Also, your implementation must run without the need to access any other externally running software. Failing to meet this last requirement means 0 points for the implementation part of your project.

Maximum number of pages for this section: 4

Time logs



Team number	Team number:	30	
Member	**Activity**	**Week number**	**Hours**
Brian Vera	SD Corrections, PD Corrections, Implementation	7	8
Sammy Fehr	SD Corrections, CD Corrections, Implementation	7	8
Jose Gutierrez	SM Corrections, CD Corrections, Implementation	7	8
Shota Goginashvili	PD Corrections, CD Corrections, Implementation	7	8
		TOTAL	32