
# Software Design Assignment 2

**Team number:** 16

**Team members:**


| Name | Student Nr. | Email |
|---|---|---|
| Ceren Duru Çınar | 2835659 | c.d.cinar@student.vu.nl |
| Kerem Boncuk | 2836559 | k.boncuk@student.vu.nl |
| Sena Deniz Avukat | 2835605 | s.d.avukat@student.vu.nl |
| | | |

# 1. Summary of changes from Assignment 1

*Author(s): Ceren Duru Çınar*

- Since Realtime Playback Interface/Soundboard can take too much time and effort, we excluded it from our project and decided to focus on implementing the sequencer in a more qualitative way.
- Since velocity and duration are features of sound, we added these to Feature 2 (SoundController) instead of Sequencer. As SoundController handles the changes in the sound, the change in velocity and duration will be handled in this feature. For this we added velocity and duration to the **Sound** class. We added the functions that update them to the interface **SoundController** which is implemented by **Sound**. This way, the features and operations related to sound are not included in **Sequencer** and our classes will be more compact.
- We received feedback that mentioned, having a lot of time signatures may be hard for us in the future implementation part. For this, we added 2 timeSignatures as enumeration to **Sequencer** for now. If we feel that we can manage more while coding, we will add more options.
- We excluded tempo from our model for now, as our mentor's feedback mentioned that it may complicate things.

# 2. Package diagram

*Author(s): Ceren Duru Çınar - Sena Deniz Avukat*

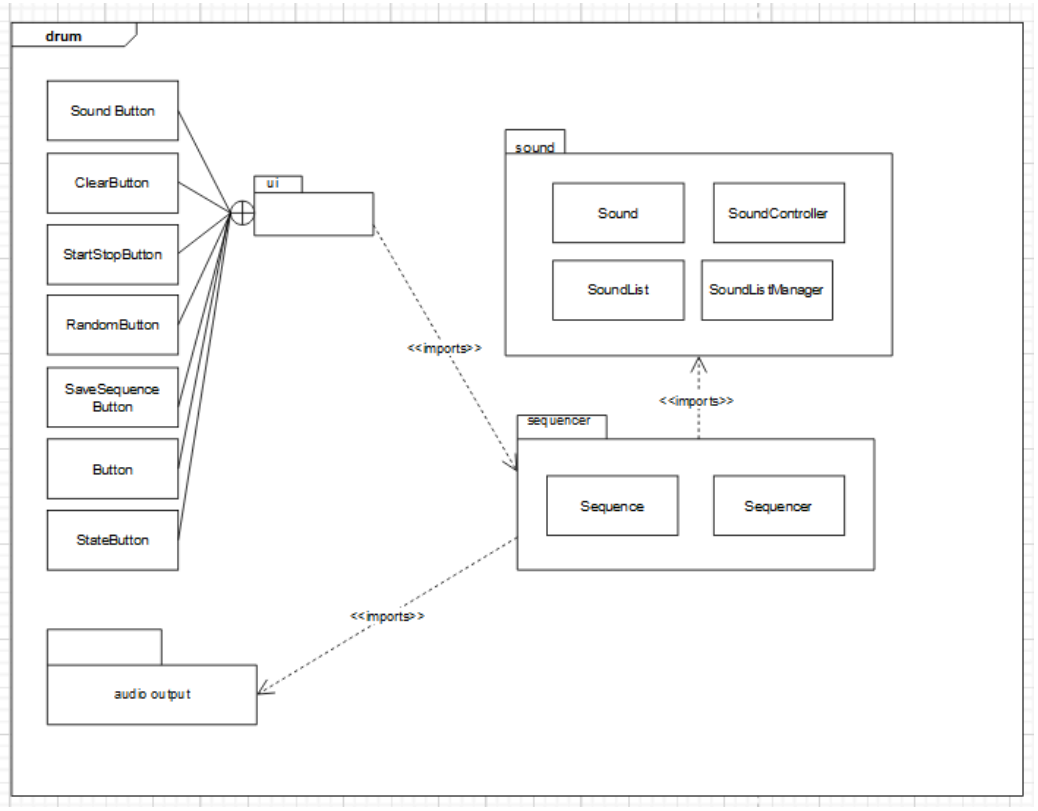



Figure 1. Package Diagram

**ui Package:** The ui package has user interface classes. We may need to add more classes to this package for user interface, but for now we only added user interface components that are related to the business logic of our system. These classes include an abstract class **Button**, its subclasses **SoundButton** and another abstract class **StateButton**. Also, it has the subclasses of **StateButton**. These buttons will be shown to the user and the user will interact with the system using these components. This package imports the sequencer package which already imports the sound package since they will be represented in a user interface and users will be able to interact with the program to update sounds, choose sounds from sound list and see the sequencer's loop on the screen. By importing packages in an ordered way we avoid getting into an import loop problem.

**sound Package:** The sound package has **Sound, SoundList** classes and **SoundController, SoundListManager** interfaces. This package has sound and components of the program as well as listing, keeping, adding and removing sounds to/from the system. This package does not import any other packages, but it is used by sequencer package.

**sequencer Package:** The sequencer package has **Sequence** and **Sequencer** classes.This package handles the **Sequencer**'s functions as well as the sequence that it plays. The package imports the sound package since when the **StartStopButton** is pressed, the sounds that belong to a triggered **SoundButton** need to be played. Hence, the **Sequencer** needs to access sounds. Moreover, this package imports the audio output package which handles outputting sounds from the device. **Sequencer** uses this package to play the sequence to the user via speakers of the device.

**audio Package:** This package has classes to organise and handle the audio outputting mechanism in the device. It is accessed by the sequencer package for playing a sequence. The main aim of this package is to increase the modularity of the program.

# 3. Class diagram

*Author(s): Ceren Duru Çınar, Kerem Boncuk*

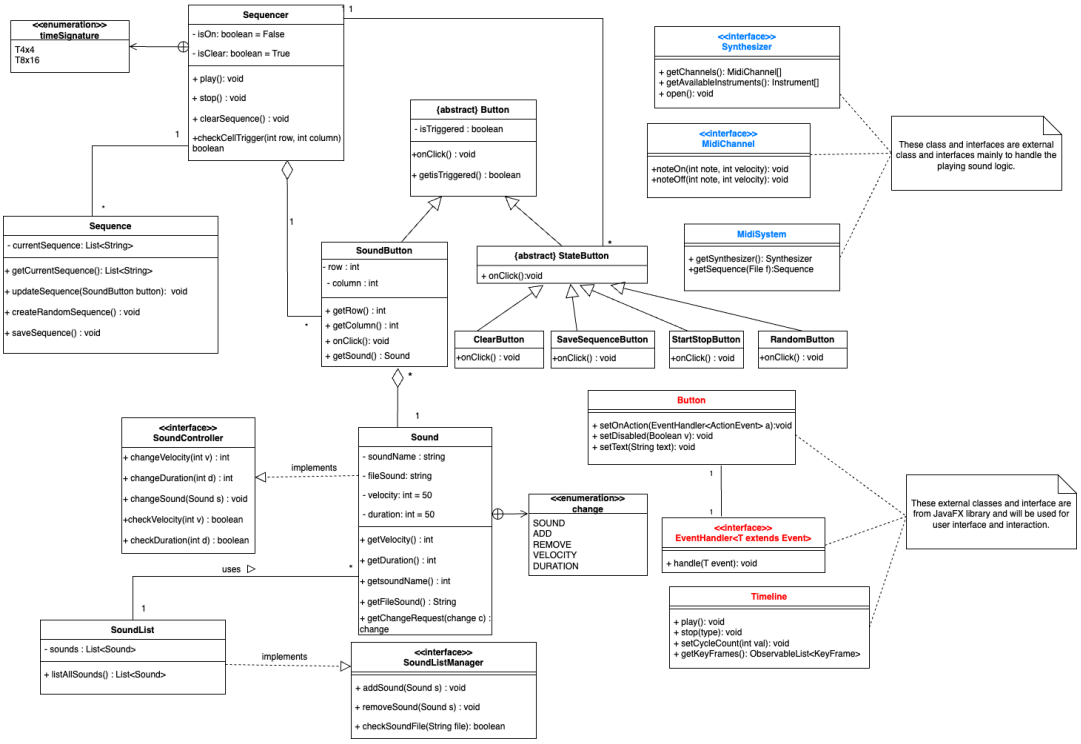



Figure 2. Class diagram

## 3.1 Regular Classes

**Sequence:** The **Sequence** class represents a sequence of sounds. It has a currentSequence attribute as a List<String> type which represents the matrix of the sound buttons and there is a + if button is triggered, - otherwise in the string. For example, [“+-+---+”, “--++--+”] represents two rows and 7 columns of buttons. Keeping a list of strings will make it easy to store and save the sequences in a .txt file. For saving the sequence, saveSequence() operation will be used. The creation of a random sequence will be handled by createRandomSequence() operation which will choose a random number of buttons from each row to be triggered, and the sequence and buttons will be updated. The sequence can be updated with the updateSequence(SoundButton button) operation. Here, the argument is of type **SoundButton** and to update the sequence, the row and the column (coordinates) of the sound button will be used for reaching that specific button’s information in the currentSequence list. There is a getter for currentSequence since its visibility is private and Sequencer needs to access it to play the sequence. There is a 1 to many association with Sequencer because all **Sequence** instances can have only one **Sequencer** but a Sequencer can have many **Sequence** instances. Many sequence instances include the current sequence to be played as well as the preset sequences.

**Sequencer:** The Sequencer class manages the loop that the Drum Machine plays on, isOn attribute is set to False by default; however it will be toggled between True and False with the help of **StartStopButton.** isClear attribute is set to True by default and shows the current sequence is empty (for example [“----”,”----”,”----”,...] in a four column sequence). When the sequence is updated, isClear attribute changes to False, isClear only changes back to True when **ClearButton** is pressed. To keep the sequencer in a loop and show which **SoundButton**s are being played

currently currentPlayingColumn attribute is used, by default currentPlayingColumn is set to “0” and it can only take “0” to “column number minus one” values (for example if a sequence has 4 columns currentPlayingColumn can take only “0”,”1”,”2” and “3” values) and loop will be keep going between these values. The Sequencer classes’ methods are pretty straightforward: play() method starts the loop of the sequence and stop() method stops, **StartStopButton** class onClick() method is used to toggle between these two states. ClearSequence() method is used to create an empty sequence and this method will be called when **ClearButton** class onClick() method is called. Lastly checkCellTrigger(int row, int column) returns True if a **SoundButton** is pressed in its designated row and column. **Sequencer** class is associated with 1 to 1 relationships with MIDI package classes(**Synthesizer**, **MidiChanel** and **MidiSystem**) because it uses MIDI files. Also it is in aggregation relation with 1 to many **SoundButton** classes and associated with 1 to many **Sequence** classes.

**Sound:** **Sound** class represents the tune played when the **Sequencer**’s loop is currently on a specific SoundButton object. It has a MIDI file, default velocity and duration of 50 and a name. There are getters for velocity, duration, sound name and sound file path to be used in **MIDI** and **Synthesizer** operations. This class implements the **SoundController** interface, whose relation will be discussed in part 3.3.There is an 1 to many aggregation between **Sound** and **SoundButton** because a **SoundButton** can only have one **Sound** while a **Sound** can have many **SoundButtons** since all **SoundButton** instances that are on the same row have the same **Sound**. The relationship is aggregation because in the program, a **Sound** can exist without a **SoundButton** in the **SoundList**. **Sound** class uses **SoundList** to get a chosen **Sound** object among all the Sound instances listed with listAllSounds() operation from the **SoundList** class. Then, that specific **Sound** object can be used as an argument to the function changeSound(Sounds) which changes the **Sound** object that called this operation to the provided **Sound** object. Finally, getChangeRequest(change c) is used to get the user's choice of changing a **Sound** object’s aspect, add or remove it.

**SoundList:** This class is used for storing the **Sound** instances in the program. It has sounds of type List<Sound> and all **Sound** instances, preset or added are stored in this list. For adding and removing sounds to the program, this class implements the SoundListManager interface, whose relation will be discussed in 3.3. The class **Sound** and this class use each other. **SoundList** uses **Sound** to add or remove **Sound** instances to its list. The relationship is 1 to many association because **SoundList** necessarily has many Sound instances, yet a Sound object can only belong to one SoundList.

**SoundButton:** The **SoundButton** class represents the sound buttons that the user presses and triggers to be played in a sequence. This class extends the **Button** abstract class and overrides the onClick() operation. This operation is in accordance with the setOnAction() function of the **JavaFX Button**. The operation onClick() is executed when the **SoundButton** object is clicked and it will update the current sequence’s list. It also has row and column of type int. The coordinates of a specific **SoundButton** object of all **SoundButton** instances (which is represented as a matrix in the code and grid in the user interface) is stored by these attributes. Using these features, updating the List<String> currentSequence becomes much more efficient and easier.

**ClearButton:** This class extends the abstract **StateButton** class, and overrides **StateButton** classes' onClick() method. Overridden onClick() method will create an empty/clear sequence and update the sequence according to that.

**SaveSequenceButton:** This class extends the abstract **StateButton** class, and overrides the **StateButton** classes' onClick() method. Overridden onClick() method will save the current sequence .txt data and the user will be able to import it later.

**StartStopButton:** This class extends the abstract **StateButton** class, and overrides the **StateButton** classes' onClick() method. Overridden onClick() method will be responsible for play() and stop() methods in the **Sequencer** class. When started **Sequencer** will be in an inactive state by default, however **StartStopButton** will call the play() method from **Sequencer** if the sequencer is inactive, and stop() method vice versa.

**RandomButton:** This class extends the abstract **StateButton** class, and overrides the **StateButton** classes' onClick() method. Overridden onClick() method will call the createRandomSequnce() method which generates a random sequence from the **Sequence** class and will update the sequence according to this newly generated sequence.

## 3.2 Abstract Classes

**Button:** This class represents an abstract **Button** that implements an observer pattern behaviour to catch user interactions and take an action accordingly. The class has an unimplemented onClick() operation, which will be overridden by its subclasses for a specification in the behaviour. It also has a getter for isTriggered to be used by other classes so that they can take actions in each scenario, for example if a **StartStopButton** object's (subclass of **Button**) isTriggered is true and the button is clicked, then stop() operation needs to be invoked in **Sequencer**.

**StateButton:** This class is a subclass of **Button** but it is also abstract. This class represents the abstract button that is used to change the behaviour of a specific part of the program. Its subclasses are buttons that control clearing, starting or stopping, saving or random sequence creation. It only has onClick() inherited from its superclass **Button**. This operation will be overridden by all its subclasses. It has a 1 to many association with Sequencer because a Sequencer object can have many **StateButton** objects while a **StateButton** object can only have 1 **Sequencer** object. The relationship is association because **Sequencer** operations such as play(), stop() and clearSequence() need to be done when a related StateButton object is clicked.

## 3.3 Interfaces

**SoundListManager:** This interface provides functionality to manage operations on **SoundList**'s sounds. The responsibility of sounds' is to store **Sound** objects, but users can add or remove **Sound** objects to this list and this is the interface's main functionality. These functionalities are provided by addSound(Sound s) and removeSound(Sound s) operations respectively. Moreover, the operation checkSoundFile(String file) checks the format of the file before adding the **Sound** object to the list. This interface is implemented by **SoundList** and by this way, **SoundList** gains the ability to manage its list.

**SoundController:** SoundController interface is responsible for managing sound properties. Sound class implements SoundController interface so, with the help of SoundController interface, sound class will be able to manage sound properties too. Attributes of sound such as velocity, duration and soundFile will be managed respectively with interface methods changeVelocity(int v), changeDuration(int d) and changeSound(Sound s). Also with the help of checkVelocity(int v) and checkDuration(int d) method sound velocity and duration could be checked if they are at a certain point.

## 3.3 External Classes

**Button:** JavaFX provides a Button class specifically for GUI. We are planning to use JavaFX Button class on our classes such as: **SoundButton**, **ClearButton.**The overridden onClick() method will be called with the setOnAction(EventHandler<ActionEvent>) method. We will also be able to name the Button object with the help of the setText(string text) method in order to be more understandable for our users.

**Timeline:** JavaFX provides a **Timeline** class for animating **KeyFrames** sequentially. We are planning to use this class to represent the **Sequencer**'s loop animation. **Timeline**'s play() and stop() method will be connected with Sequencer class' play() and stop() methods so that when the loop starts, the animation will also start. The animation will run until the user wants it to stop, with the help of setCycleCount(int val) method.

**MidiSystem:** This class provides access to the installed MIDI system resources, including devices such as synthesizers. To initialize the Synthesizer object, `synthesizer = MidiSystem.getSynthesizer()` code can be used. Moreover, to extract a MIDI sequence from a specified file, getSequence(File f) can be used.

## 3.4 External Interfaces

**MidiChannel:** This interface has functions to control various aspects of a specific MIDI channel such as notes and instruments within a **Synthesizer**. The noteOn(int note, int val) and noteOff(int note, int val) functions play and stop the specified note with the specified velocity.

**Synthesizer:** This interface generates sound based on MIDI instructions it receives. It has a collection of MidiChannel's and another collection of Instrument's. From that collection, a specific object can be used to produce a sound.

**EventHandler<T extends Event>:** This is a functional interface that handles when an event occurs and an action needs to be performed. It only has a handle() function and it will be overridden by the object who needs to observe the event and take the action.

## 3.5 Enumerations

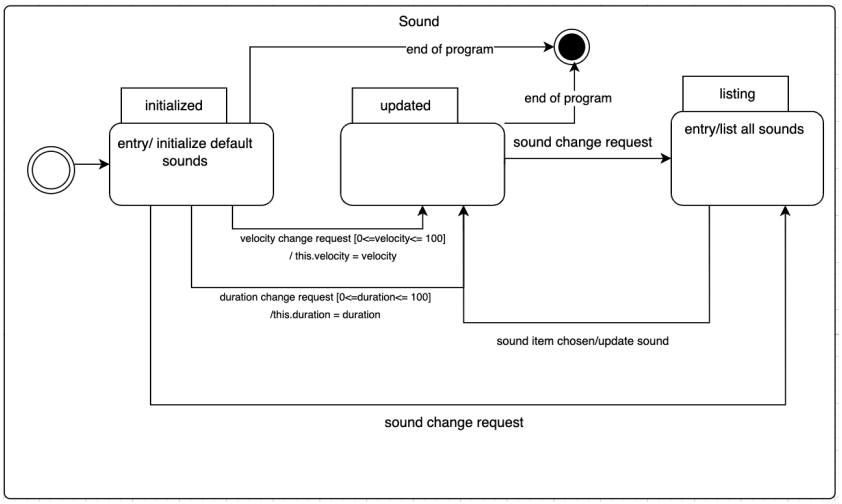
**timeSignature:** This is an enumeration inside the **Sequencer** class. This represents the musical time signature/bar that the *Sequencer* plays the *Sequence* object. The reason why enumeration is used for this, is not having any unwanted complications of using a String and having restricted options for our program.

**change:** This is an enumeration inside the **Sound** class. This represents the user's decision on which aspect of the Sound to change, add or remove. The reason why enumeration is used for this, is not having any unwanted complications of using a String and increasing the understandability of our program for future use.

# 4. State machine diagrams

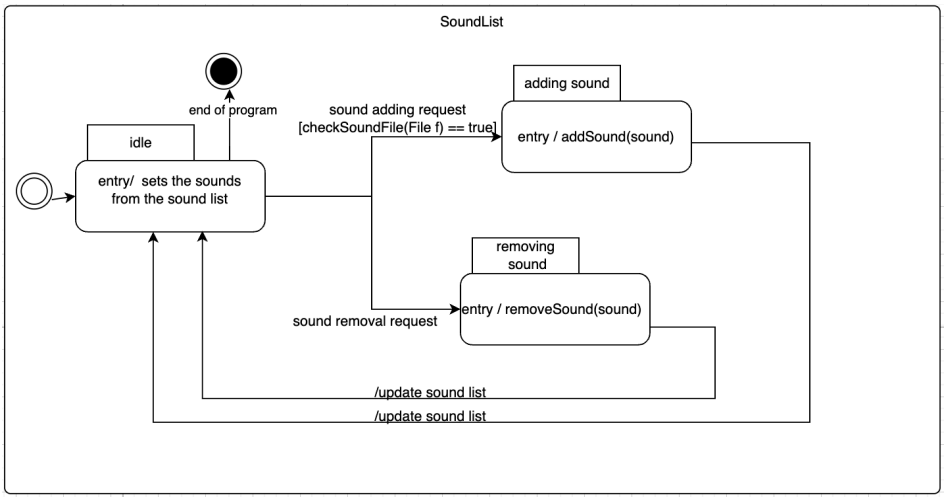
*Author(s): Kerem Boncuk, Ceren Duru Çınar*

**Sound Class Machine:** This state machine diagram shows the internal behaviour of the *Sound* object. From the starting state, the *Sound* moves to the "initialized" state where *Sound*'s are initialized from the default sounds. From this state, when a "velocity change request" or "duration change request" event occurs, first the provided parameter velocity or duration is checked, if true then the value of velocity or duration attribute is updated and the state is changed to "updated". From "updated" or "initialized" state, if a "change sound request" event occurs, the state changes to "listing" and sounds are listed as the machine enters this state. In this state, if a "sound item chosen" event occurs, then the *Sound* is updated and the state is changed to "updated". The machine can switch to the end state from "initialized" or "updated" states when the program ends because the user can just use the *Sound*'s with the defaut settings or update them.

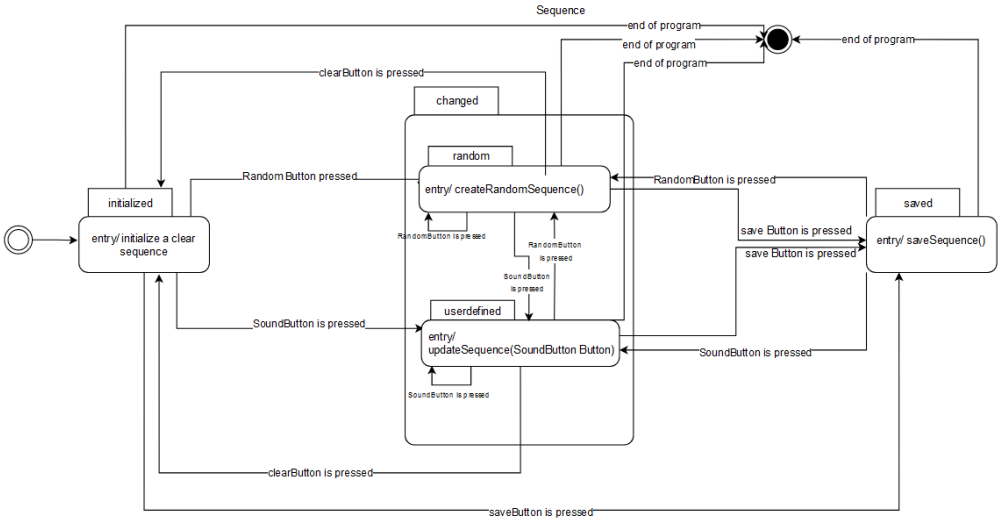



**SoundList State Machine: SoundList** Class state machine gives information about possible internal behaviour states of the *SoundList* object. *SoundList* object's behavioural states are crucial to understand how default and user defined sounds are implemented in our application. Following part verbally describes the state's behaviours: From the starting state, the *SoundList* moves to the "idle" state where *Sound* objects are set using the sound list. From this state, if a "sound adding request" event occurs, first the file format is checked with checkSoundFile(File f) and if the returned format is true, the state is changed to "adding sound" state and the addSound(sound) function is called as the machine enters this state. From this "adding sound" state, immediately the sound list is updated and the state is changed back to "idle" state with the action of updating the sound list. From the "idle" state, if a "sound removal request" event occurs, the state is changed to "removing sound" state and the removeSound(sound) function is called as the machine enters this state. From this state, immediately the sound list is updated and the state is changed back to "idle"

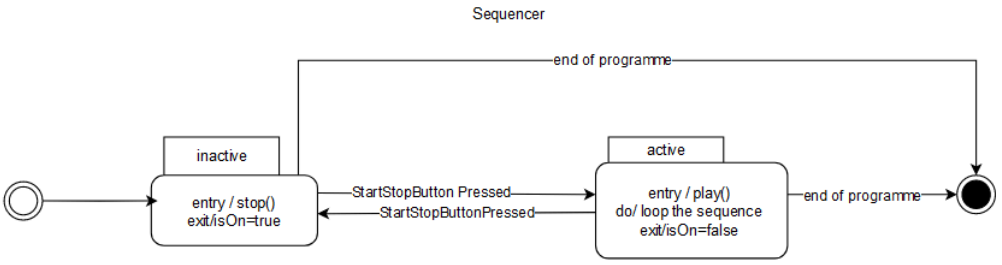
state. The machine switches to the end state from the “idle” state when the program ends because no “idle” state is reached immediately after “adding sound” and “removing sound” states.



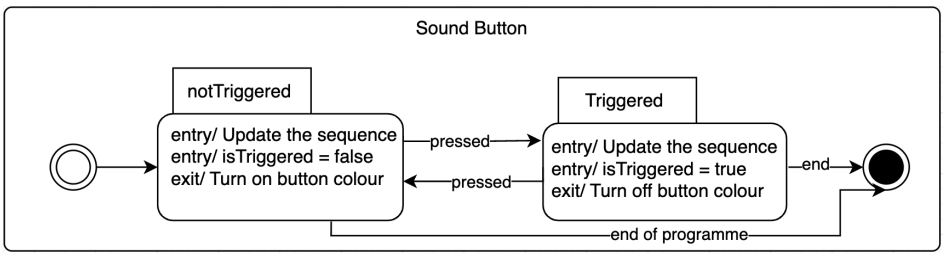

**Sequence State Machine: Sequence** This mainly represents how our Drum Machine’s behaviour would change according to the user’s input. In this state machine, we did not use many guards because we want users to be able to change the current sequence as freely as possible for their experience. Only the saved sequences have guards because saving the same sequence multiple times will decrease the performance and quality of the program. From the starting state, the *Sequence* moves to the “initialized” state where a clear sequence (a list of strings containing “-------...”) is initialized. This state can be reached from all other states when the “clear button pressed” event occurs. The “changed” state represents a change in the initial state and includes substates “random” and “user defined” states. “changed” state can be reached either by “random button pressed” or “sound button pressed” events. “random” state is a substate of “changed” state. In “random” state a random sequence is created with createRandomSequence() method and “random” state can be reached from any possible state by “random button pressed” event. Similar to “random” state, “user defined” state is a substate of “changed” state. In “user defined” state sequence will be updated according to user’s input with the updateSequence(SoundButton button) method and “user defined” state can be reached from any possible state by “sound button pressed” event. The “saved” state represents the state when the user wants to save the current sequence. It can be reached from all other states if the “save button pressed” event occurs and the guard “sequence not in saved sequences” is true. Since we wouldn’t want to save the same sequence twice.



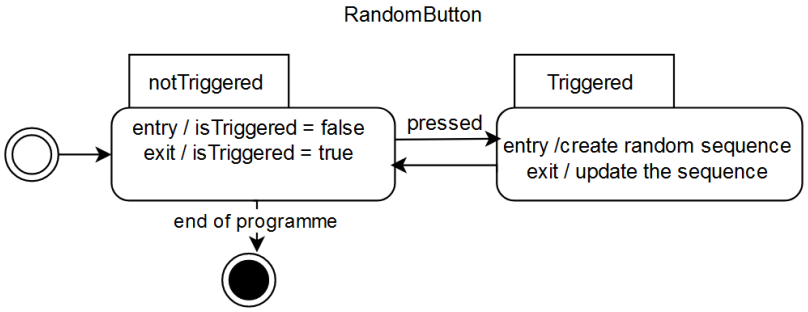

**Sequencer State Machine:** *Sequencer* State Machine represents the behaviour of our Drum Machine's inactive/active status. From the starting state the *Sequencer* moves to the "inactive" state that represents the start of the application, where the *Sequencer* is inactive and *StartStopButton* has never been pressed or it has been pressed an even number of times. *StartStopButton* being pressed even number of times means that the *Sequencer* has stopped. The function stop() from the *Sequencer* class is called as the machine enters "inactive" state and isOn attribute set to true exit of the state. From this state if the user presses *StartStopButton***,** an event occurs and the state changes to "active". This state represents that *Sequencer* is active and looping on the current sequence. The function play() from the *Sequencer* class is called as the machine enters an "active" state and the program loops the sequence until this "start/stop button pressed" event occurs and the state is left. Lastly, exit of the "active" state isOn attribute of the sequencer is set to false.



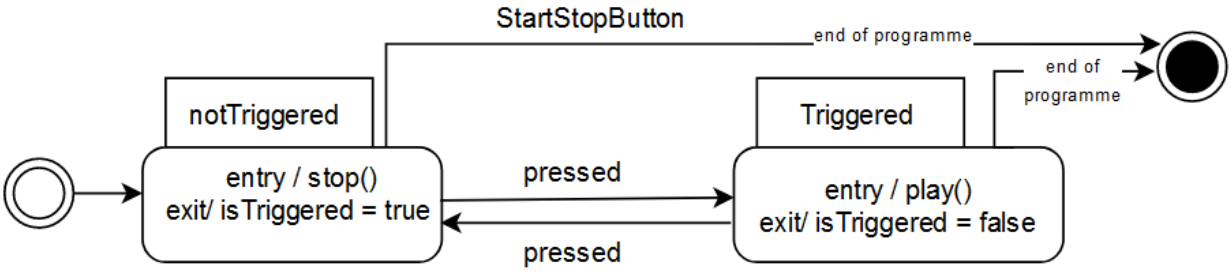

**SoundButton State Machine:** All *SoundButton's* will have two states: they are either in the state of "notTriggered" or "Triggered". When they enter the "Triggered" state they will update the sequence and change isTriggered's value to true. The button's color is turned off when this state is left. Vice versa when they enter the "notTriggered" state they will update the sequence and change isTriggered's value to false. The button's color is turned on when this state is left. Change between these two states is maintained by pressing the *SoundButton* which was mentioned by "pressed" event.




**RandomButton State Machine:** This state machine represents the behavioural change in *RandomButton* object. *RandomButton* Class has two states ”notTriggered” and “Triggered” states. At the start of the application *RandomButton* starts in the “notTriggered” state. If *RandomButton* is pressed, it enters to the “Triggered” state and a random sequence is created; however, it automatically re enters to the “notTriggered” state after a random sequence created.




**StartStopButton State Machine:** This state machine represents the behavioural change in *StartSopButton* class. This class has two states similar to the *RandomButton* states namely “notTriggered” and “Triggered” states. At the beginning of the application *StartStopButton* starts in the “notTriggered” state. By pressing the start/stop button the state will toggle between these two “Triggered” and “notTriggered” states. When entered into a “notTriggered” state stop() method in the *Sequencer* class will be called similarly, when entered into the “Triggered” state play() method in the *Sequencer* class will be called. It will be possible to close the application while *StartStopButton* is in any state.



# 5. Sequence diagrams

*Author(s): Ceren Duru Çınar, Sena Deniz Avukat*

**Sound Changes Sequence Diagram:**

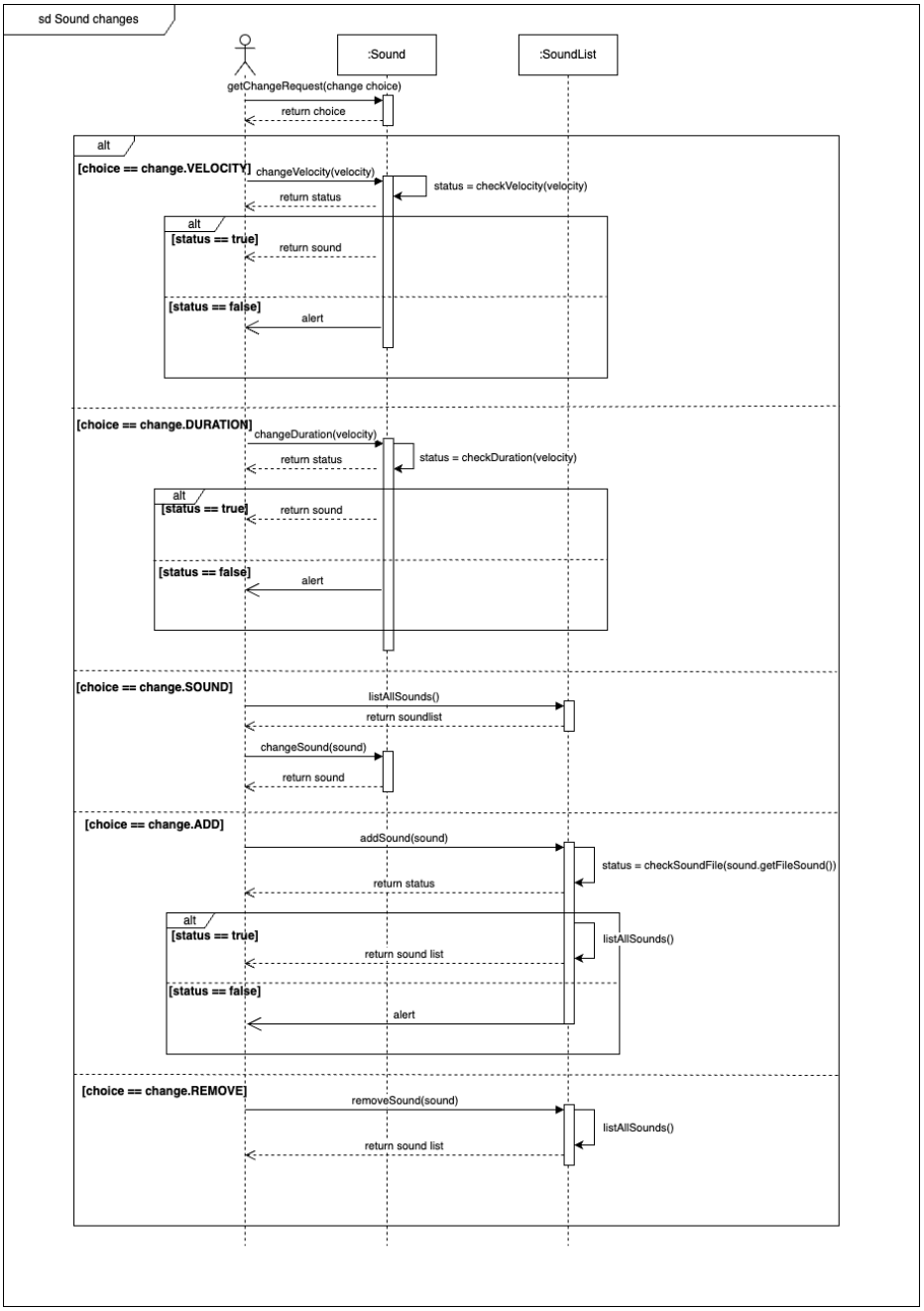



Figure 10. Sound changes sequence diagram

This sequence diagram represents what happens when a *Sound object* or its components need to change. This mechanism is handled by **Sound** and **SoundList** classes that implement interfaces SoundController and SoundListManager respectively, so the *Sound* and *SoundList* are two of the interaction partners. Another interaction partner is the user, who is the actor that interacts with the system. User is added as an interaction partner because we want the user to decide when to

change the *Sound*, add a new *Sound*, remove a *Sound* and change a specific Sound's velocity or duration.

Since the user decides to change, add or remove the sound and change duration or velocity, operations are operands to the alt fragment. Using the alt fragment makes sure that alternative operations are executed based on the user's decision. The user's request is obtained by the getChangeRequest(change choice) operation. This operation takes a change enumeration which represents the type of request. Then, choice is returned to be checked.

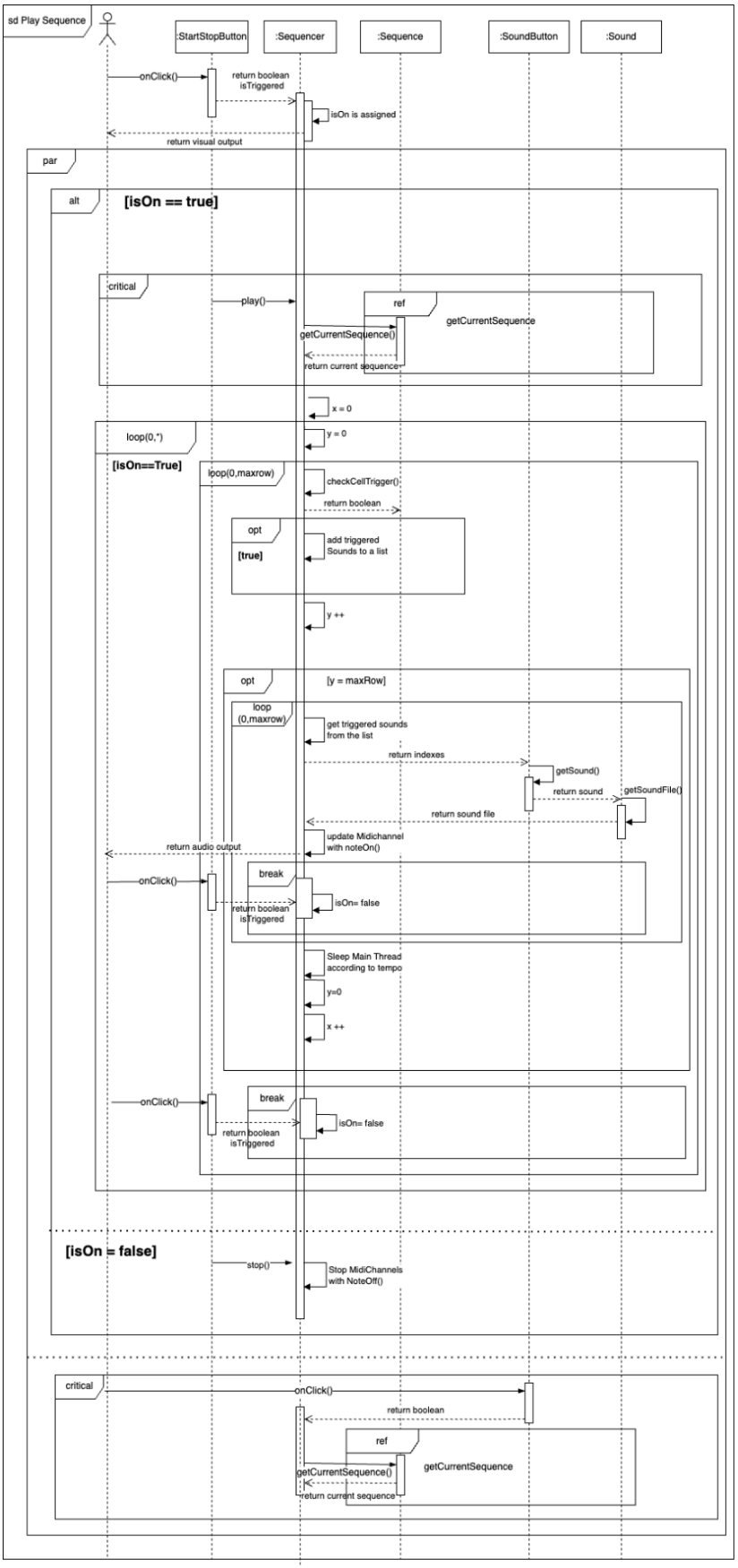
If choice is equal to change.VELOCITY or change.DURATION, *Sound*'s changeVelocity(int v) or changeDuration(int d) operation is called. *Sound* checks if the velocity or duration is in the correct bounds with checkVelocity(int v) or checkDuration(int v) and returns a boolean to the user. Here there is an alt fragment to take action according to the result of the returned boolean. If it is true, then updated *Sound* will be returned, else the user will be alerted. These two operands are write together but it is important to highlight that they are separate operands in an alt fragment, they are just written together because their functions are nearly identical.

If choice is equal to change.SOUND, the user will call the listAllSounds() operation from *SoundList* and *SoundList* will return the soundlist to the user. Then the user will select a *Sound* object from the list to replace the old *Sound* with and provide it to *Sound* to call changeSound(Sound sound). Then the *Sound* object can return the updated sound to the user.

If choice is equal to change.ADD, the user calls *SoundList*'s addSound(Sound s) operation. *SoundList* checks the format of the file with checkSoundFile(s.getFileSound()) and returns a boolean to the user. Here there is an alt fragment to take action according to the result of the returned boolean. If it is true, *SoundList* calls its listAllSounds() operation and returns the updated list to the user, else the user will be alerted.

If choice is equal to change.REMOVE, the user calls *SoundList*'s removeSound(Sound s) operation. Then *SoundList* will immediately remove the specified *Sound* from the list and call its listAllSounds() operation. Then the updated list is returned to the user.

## Play Sequence and GetCurrentSequence Sequence Diagrams:



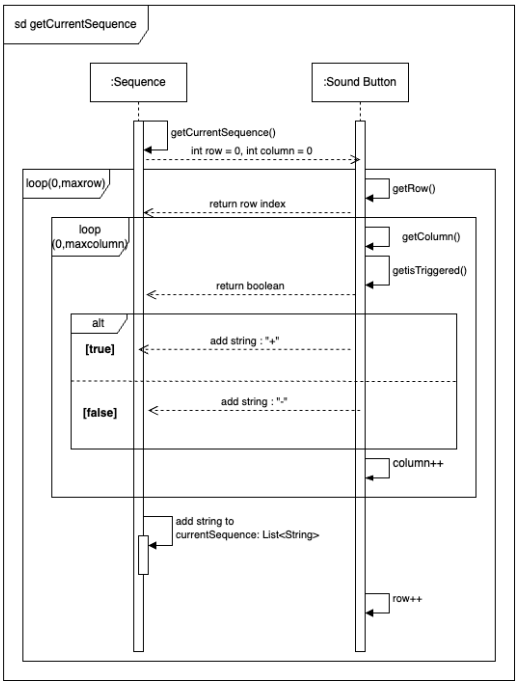




Figure 11. Get Current Sequence sequence diagram

Figure 12. “Play Sequence” sequence diagram

This sequence diagram represents how Sequencer plays and stops the sequence. This mechanism starts with user interaction with the *StartStopButton*. User is the interaction partner because the user is the one who decides when to stop and play the sequence. With onClick() the button sends a message to the *Sequencer* and the isOn variable is assigned accordingly. With an alt frame, whether to play or stop the sequence is decided.

When isOn is true, firstly the play() function is executed and current conditions of *SoundButtons* are taken within the ref frame getCurrentSequence sequence. As shown in the separate diagram (Figure 11), when the getCurrentSequence() function is called, first the *SoundButton* with the row=0 column=0 is checked. Then,as in the alt frame, if the button is triggered a ”+” char, if the button is not triggered a “-” is added to the string.This is repeated for each column.

Each string represents a row. For example, the “+--+-+” string means that the 0,3 and 5th *SoundButtons* on that row are pressed. And each row is assigned with a *Sound*. That means that every string holds one sound’s pattern. When every column is read, the string is added to the list of strings variable *currentSequence*. This is repeated for each row.

After getting the *sequence*, the outer loop is for iterating over every column and when the user triggers the stop button the loop breaks. Variables x and y are local variables inside the play() function.Variable y represents rows(each string in the currentSequence) Variable x represents columns(index within each string).

The loop inside of it is for iterating over each row of a column. Each loop checks whether the specific index of every string contains the char ”+” and if it does, the y (row index) is added to a local list inside the play() function. The reason for that is to be able to play every pushed *Sound* of a column at the same time. After the last row of the column is checked(ensured by the opt frame), the list of the pushed buttons indices for the specific column is read in a loop. Loop takes the indices one by one, then reaches a specific *SoundButton* because the *Sound* object is reachable through the *Sound Button* object. Sound object then returns the sound file assigned to *Sequencer***.** *Sequencer* is responsible for updating MIDI channels with retrieved sound files. This can be done with the noteOn() function of the **MIDIChannel** interface. Break frame ensures that this loop also can be terminated with user interaction.

Then, the main thread sleeps for a time to increase the sense of synchronization and timing precision. Then the next column’s check of the triggered cells starts and the same steps are repeated until the user presses the *StartStopButton*. When, user stops the sequence isOn becomes false and stop() function makes channels silent with noteOff() function of the MIDIChannel interface.

Par frame is used to show that the user can change the sequence while it is being played. When the user changes the *sequence* by pressing a *SoundButton*, onClick() of the *SoundButton* sends a message to the *sequencer* to update the current *sequence*.

# 6. Object diagrams

*Author(s): Ceren Duru Çınar, Sena Deniz Avukat*

This object diagram represents a snapshot of the Sequencer when the 0'th column is being played. The *Sound Button* objects that *sequencer* uses are the 0th indexes of every row. Since, when playing the *currSequence*'s currentSequence, *sequencer* checks one column at a time, this snapshot is only focusing on one column that is being played. At this time a *sequencer* has many *Sound Buttons* and each sound button has one specific *Sound* object.

*currSequence* which is a *sequence* object holds the isTriggered information of sound buttons. As seen in the diagram, every string in the currentSequence list starts with the isTriggered value of each row's first sound button. If isTriggered is true "+", if it is false "-".

Every sound button object has a sound. Since these are sound buttons from different rows, they have distinct sounds. For instance, the *sound0* object represents the first row's sound and the *sound1* represents the second row's sound. Each sound object has soundName, fileSound, velocity and duration variables which are explained in the Class Diagram part.

*soundList* object holds the Sound objects in a list named sounds. There are 6 sounds in the diagram but the user can add more sounds.

Additionally, button objects in the diagram are *randomButton*, *clearButton*, *saveButton* and *playButton*. Since the snapshot is from a playing state, *randomButton*, *clearButton*, and *saveButton* are not triggered (isTriggered =false). Only the *playButton* is triggered (isTriggered= true).

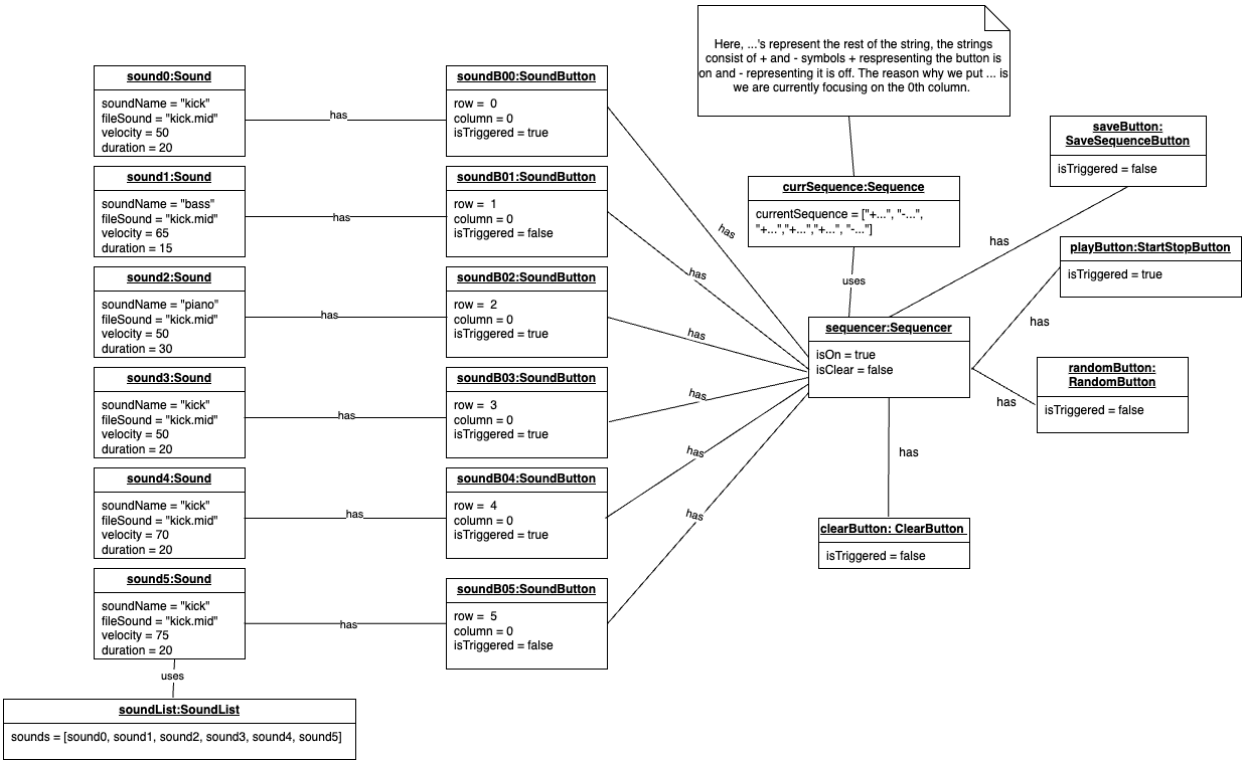



Figure 13 "Play Sequence" sequence diagram

# 7. Time logs

| **Team number** | 16 | | |
|---|---|---|---|
| | | | |
| **Member** | **Activity** | **Week number** | **Hours** |
| Ceren Duru Çınar | Created a draft for class diagram | 3 | 1.5 |
| Sena Deniz Avukat | Worked on class diagram | 4 | 4 |
| Kerem Boncuk | Worked on class diagram | 4 | 4 |
| Ceren Duru Çınar | Worked on class diagram | 4 | 4 |
| Sena Deniz Avukat | Worked on package diagram | 4 | 1 |
| Kerem Boncuk | Worked on package diagram | 4 | 1 |
| Ceren Duru Çınar | Worked on package diagram | 4 | 1 |
| Sena Deniz Avukat | Worked on state machine diagrams | 4 | 2 |
| Kerem Boncuk | Worked on state machine diagrams | 4 | 2 |
| Ceren Duru Çınar | Worked on state machine diagrams | 4 | 2 |
| Ceren Duru Çınar | Created sound changes sequence diagrams | 5 | 2 |
| Sena Deniz Avukat | Created play sequence and get current sequence sequence diagrams | 5 | 6 |
| Kerem Boncuk | Wrote the report, updated Assignment 1's slides | 5 | 2 |
| Ceren Duru Çınar | Wrote the report | 5 | 2 |
| Sena Deniz Avukat | Wrote the report | 5 | 2 |
| Kerem Boncuk | Finalized all the diagrams | 5 | 1 |
| Ceren Duru Çınar | Finalized all the diagrams | 5 | 1 |
| Sena Deniz Avukat | Finalized all the diagrams | 5 | 1 |
| Sena Deniz Avukat | Created the object diagram | 5 | 2 |
| Kerem Boncuk | Created time logs | 3,4,5 | 1 |
| | | | |
| | | | |
| | | **TOTAL** | 42.5 |