Assignment 3

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

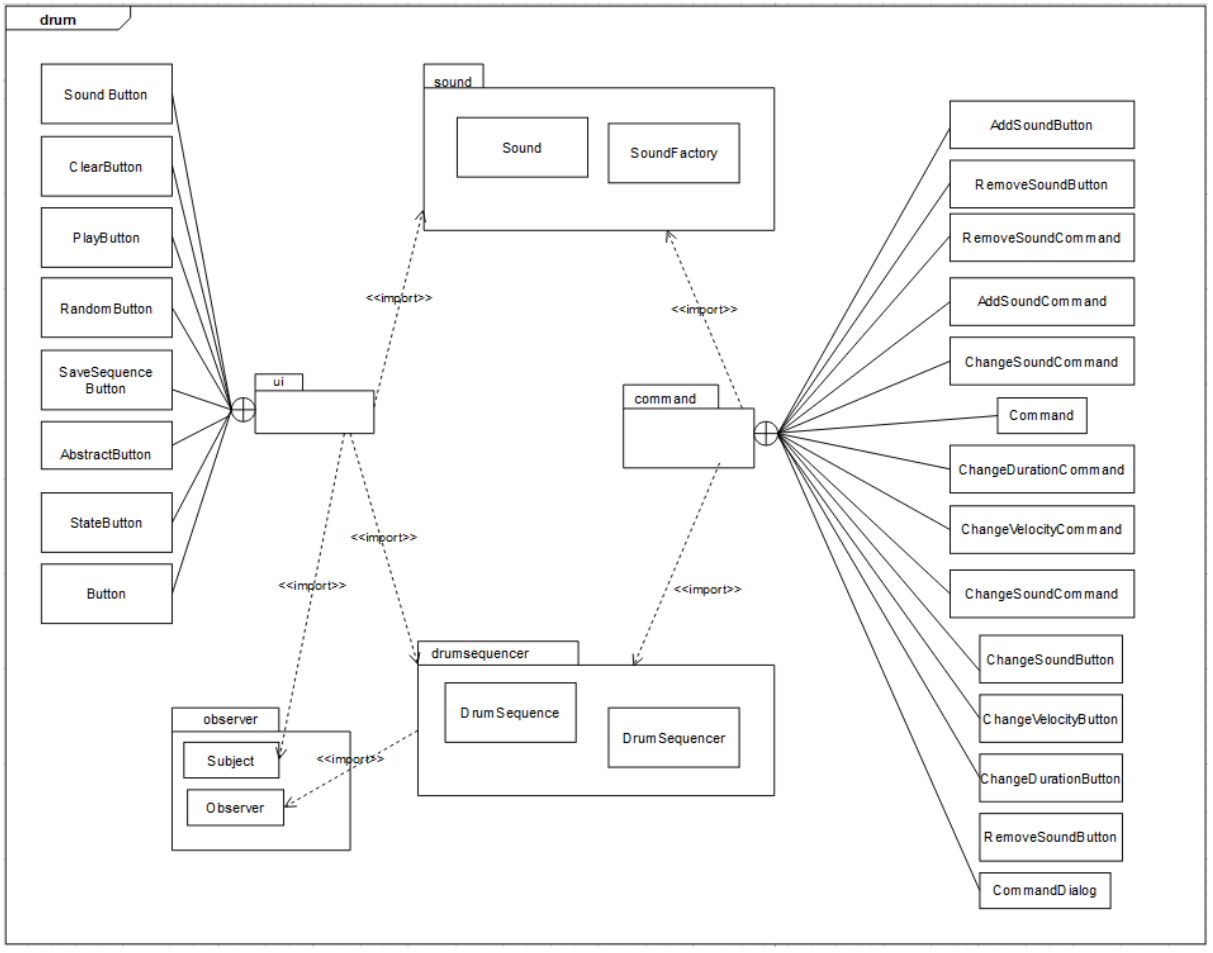
Summary of changes from Assignment 2

Missing Functionalities in Class Diagram: The most notable improvement from assignment 2 is the reorganisation of our class diagram. To make our class diagram feasible and to implement design patterns we added new classes to our class diagram (namely command buttons, sound factory for flyweight design pattern and observer interfaces). Also we added new attributes and methods inside existing classes to make our implementation feasible and more understandable from outside look. Lastly we omitted unnecessary classes from the class diagram (namely soundList class and SoundController and SoundListManager interfaces) according to feedback from assignment 2 and replaced soundList classes' purpose with a flyweight pattern design implementation.
Addition of UI class: Apart from button classes that we inherit from javafx library we created another UI class to organise all necessary UI methods in it. And implemented client code in this class. The aim of creating this class is that it will naturally be a bridge between our other computation classes and user interface.
Addition of New Packages: In order to implement pattern design such as Observer and Command in our project we introduced two new packages to the package diagram namely observer and command packages. These packages include related classes and inside these classes the design patterns are implemented respectively.
Change from Time Signature to Tempo Feature: Implementation of time signature feature meant drastic changes to our previously designed UI implementation because of the nature of the time signature feature. However, the tempo feature is almost the same result as the time signature feature without necessary UI implementation changes. Because of this reason we decided to switch to a tempo feature.
Implementation of External Libraries: Since we figure out necessary external libraries with the flow of our project, we introduced new external libraries to our project that were not mentioned in the Assignment 2. For example, we heavily used MidiSystem, Javafx, Java Utility Scheduler libraries.
Small Changes in State Machines: Even though our implementation of state machines from assignment 2 is still feasible, we changed event, action and guard names according to our code to make them more clear and readable. Also, in our revision, created setters and getters and used them according to feedback from assignment 2.

Necessary Logic Changes in Sequence Diagram: In our assignment 2, “Sound Changes Sequence Diagram” was heavily based on an “alt” frame. We concluded that separating them as different sequence diagrams is more accurate.
Mixing Class Names With the External Libraries: We changed the sequencer and sequence class names into drumsequencer and drumsequence classes because it was mixing with the external libraries.

Revised package diagram

Author(s): Kerem Boncuk, Ceren Duru Çınar



Main Changes: Added Observer package which includes Subject and Object classes. Replaced SoundList and SoundManager classes from sound package with SoundFactory class. Added the command package We moved buttons that are related to the Command objects inside the command package to not face the issue of circular imports.

Ui package: The ui package includes user interface classes. These classes include an abstract class AbstractButton, its subclasses SoundButton and another abstract class StateButton, and lastly also subclasses of abstract StateButton. This package is revised or developed only in a way that it changed what it imports after assignment 2, and it still contains the user interface components that are related with the business logic of our Project. Ui package imports, sound package, drumsequencer package and lastly the Subject class of observer package. We planned to break the cyclic import problem by importing not the whole observer package but only the class Subject in this package.

Observer package: Our observer package is newly created. We created Observer and Subject classes to implement Observer pattern design between DrumSequence and SoundButton classes. Inside our observer package Subject interface gets imported by ui package and Observer interface gets imported by drumsequencer package. The aim of our design of importing this way is to break cyclic import problem because it was not feasible to reach observer package from ui through drumsequencer package.

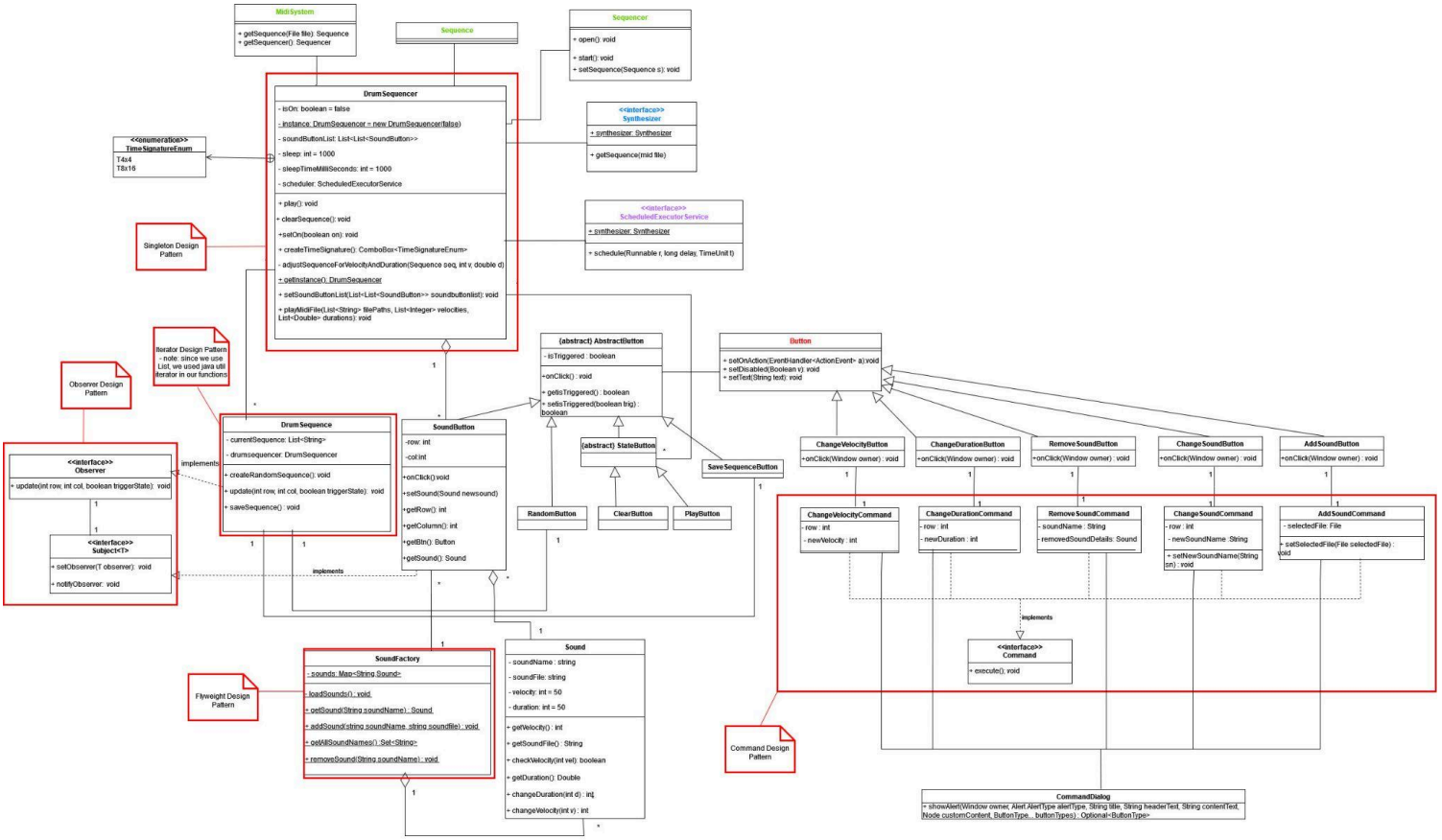
Drumsequencer package: Firstly from assignment 2 we renamed our package and class names according to feedback because it was causing misunderstanding with the external library names. Inside this package DrumSequence and DrumSequencer classes exist and with these classes this package handles the main computation of what happens behind the user interface. Moreover, the observer class inside the observer package gets imported by this package to implement observer pattern design between the DrumSequence class and SoundButton class. Lastly Command package imports drumsequencer package in order adjust velocity and duration of the sounds.

Sound Package: Our revised sound package changed a lot according to feedback from assignment 2. Currently the sound package only includes Sound and SoundFactory classes. Firstly we omitted the SoundController and SoundListManager interfaces which were unnecessary. And we replaced the SoundList class with a new class named SoundFactory. The idea behind this replacement is that we wanted to implement flyweight relation while reaching sound files from each sound button to make our project extensible and be able to create much more sound button objects if necessary with few memory needed. The Sound class still stores sound information as intended in the assignment 2. Sound package gets imported by ui and command packages. It is revised to import this way so that we are avoiding any possible cycling import problems. The ui package uses sound package for SoundButton class and command package uses sound package while adjusting available sounds and their attributes.

Command Package: Command package is a newly introduced package.The main purpose of the command package is to implement command design pattern while using ui buttons. Inside this package addSoundCommand, removeSoundCommand, changeSoundCommand, ChangeDurationCommand, ChangeVelocityCommand, ChangeSoundButton, ChangeDurationButton, ChangeVelocityButton, addSoundButton, removeSoundButton classes and Command interface is included. Command classes are respectively executed on their respective named buttons. (ex. ChangeSoundCommand class reached to ChangeSoundButton when executed). With using this package and command pattern design we aimed to make our code and design much more understandable and readable. Also by storing Command objects and their respective Button in the same package we deny the risk of cyclic import.

Revised class diagram

Author(s): Kerem Boncuk, Ceren Duru Çınar



3.1 Revised Regular Classes

DrumSequencer: We revised this class and introduced SoundButton list, sleep (integer), scheduler attributes and removed isClear. Newly added SoundButton list attribute is used while the sequence is on and checks if a SoundButton is triggered or not and plays the sound accordingly. Sleep integer attribute is used to set a certain milliseconds to sleep the thread that plays the sound. Lastly, the scheduler service attribute is used to playback all the sounds at the same time that are in the same column. In the play() function, In each line of sequence inside the method after calculating which sound buttons are triggered using the playMidiFile() method we managed to reach the designated sound of that sound button. Inside playMidiFile() we used a scheduler to playback all the sounds at the same time. Using enumeration, this class chooses between two tempo options and the createTimeSignature() method creates a dropdown menu for these tempo options. Lastly, adjustSequenceForVelocityandDuration(Sequence seq, int v, double d) method allows dynamic control over the velocity and duration of the MIDI notes. This method iterates over tracks in the sequence and for each event, if there is a needed adjustment on that event send a command using the command design pattern.

DrumSequence: The DrumSequencer class implements the Observer interface which observes changes of each SoundButton. This class's main purpose is to handle and manipulate sequences of beats according to users' commands. During revision of our class we included a DrumSequencer singleton instance inside this class to implement createRandomSequence() method. Lastly, we have

an Observer design pattern between DrumSequence class and SoundButton class. When the state of any SoundButton object changes they notify it and with the overridden update(int row, int col, boolean triggerState) method inside this class the changes are handled. With this design pattern currentSequence is updated whenever a SoundButton's state is changed.

SoundButton: This class implements Subject interface and notifies its observer. Key attributes of SoundButton class row and col integers obtain information about the position of the SoundButton in a grid. With onClick() method SoundButton toggle between triggered and not-triggered states and informs observer with the call of overridden notifyObserver() method.

Sound: ChangeVelocity() and ChangeDuration() methods are used with the aim of adjusting these properties, however it should be noted that these functions are connected with UI class ChangeVelocityButton and ChangeDurationButton using the command design pattern. With onClick() of these UI buttons a command message is sent which uses ChangeVelocity() and ChangeDuration() methods. This way we planned to represent the relationship between UI and button attributes.

SoundFactory: The SoundFactory class simplifies sound management by offering sound retrieval, addition, and removal to the drum machine. This class provides a centralised mechanism by implementing a static map attribute named sounds. This map stores sounds' names and file path as a string. At the initializations there are some predefined sounds which are added by us to this project and this sounds are loaded at the beginning of the class by loadSounds() method. Also we have getSound(string soundName) and getAllSoundName() getter functions which are used during demonstration of possible soundNames while loading sounds. Lastly, with the help of addSound(string soundName, string soundfile) and removeSound(String soundName) methods user will be able to add and remove sounds from the static map with the help of Command design Pattern that connects SoundFactory Class with UI AddSoundButton and RemoveSoundButton classes.

ClearButton: ClearButton class extends the abstract StateButton class, and overrides onClick() method. This function is used inside JavaFX button instance's setOnAction(e->{...}) to be linked with the UI.

Revised SaveSequenceButton Class: This class extends the abstract StateButton class, and overrides onClick() method. This function is used inside JavaFX button instance's setOnAction(e->{...}) to be linked with the UI.

Revised PlayButton Class: This class extends the abstract StateButton class, and overrides onClick() method. This function is used inside JavaFX button instance's setOnAction(e->{...}) to be linked with the UI.
Revised RandomButton Class: This class is the last class that extends the abstract StateButton class and overrides onClick() method. This function is used inside JavaFX button instance's setOnAction(e->{...}) to be linked with the UI.

AddSoundButton: This class extends JavaFx's Button class. It works together with the AddSoundCommand class and is connected with a Command design pattern. This class's onClick() method opens a file chooser and allows the user to choose a MIDI sound file. After that it calls the execute() method inside the AddSoundCommand class, which is a command relationship.

ChangeSoundButton: This class extends JavaFx's Button class It works together with the ChangeSoundCommand class and is connected together with a Command Design Pattern. This class's onClick() method gets all the possible sound options from the static map stored in the

SoundFactory class and creates a drop down menu using combobox feature with the options of these possible soundfiles. After receiving the result it continues to the execute() method inside the ChangeSoundCommand class.

RemoveSoundButton: This class extends JavaFx's Button class. It works together with RemoveSoundCommand class and they are connected with a Command Design pattern. This class's onClick() method includes some guards before removing the sound file. Firstly, after calling the onClick() method, if the sequencer is on it opens a warning dialog window and does not allow it to continue, otherwise, removal is allowed. After that method creates a dropdown menu using the combobox feature inside another dialog window. The string name of the chosen file is stored and the execute() method inside the RemoveSoundCommand is called.

ChangeVelocityButton: This class extends JavaFx's Button class. It works together with ChangeVelocityCommand class and they are connected with a Command Design Pattern. This class's onClick() method creates a vertical VBox and creates sliders for each row in the grid inside this VBox feature. After adjustment and when the user clicks "OK", it respectively calls setRow(int r), setNewVelocity(int v) and execute() methods of ChangeVelocityCommand class.

ChangeDurationButton: This class extends JavaFx's Button class It works together with the ChangeDurationCommand class and they are connected with a Command design pattern. This class's onClick() creates sliders for each row in the grid inside a separate window. After adjustment and when the user clicks "OK", it respectively calls setRow(int r), setNewDuration(int v) and execute() methods of ChangeDurationCommand class.

3.2 Command Classes

The features of adding, removing, changing sounds and changing velocity and duration of the sounds require a more complex user interface panel than just a button. In order to meet the requirements of these features we applied a command design pattern. We created an empty command interface with just a blank execute method and all other command classes override this execute method.

AddSoundCommand: This class implements Command interface. It's only method, execute() overrides the method from the Command interface. Its execute() method is called by the AddSoundButton class and basically the continuation of the process. The execute() method gets the name of the sound file and it's path and calls addSound(string s1, string s2) method from SoundFactory class, eventually wanted sound is added to the static map that stores all the sound files inside the SoundFactory.

ChangeSoundCommand: This class implements Command interface and its only method execute() overrides the method from the Command interface. Its execute() method is called by ChangeSoundButton class and basically sets the SoundButtons designated Sounds that are in the same row in the grid to the chosen one from the dropdown menu.

RemoveSoundCommand: Familiar with other command classes, this command class implements Command interface and its only method execute() overrides the method from the Command interface. Its execute() method is called by RemoveSoundButton class. Execute() method calls the removeSound(string soundname) method which eventually removes the unwanted sound from the static map that stores sound files.

ChangeVelocityCommand: Familiar with the previous command classes this command class implements Command interface and its only method, execute() from the Command interface. Its execute() method is called by ChangeVelocityButton class. Reaches the every SoundButton list inside that row and sets each of their velocity to user defined amount from the slider.

ChangeDurationCommand: Familiar with the previous command classes this command class implements Command interface and its only method, execute() from the Command interface. Its execute() method is called by the ChangeDurationButton class. Reaches the every SoundButton list inside that row and sets each of their duration to user defined amount from the slider.
CommandDialog: This class aims to help display alert dialogs.Its parameters contain information about the displayed windows' information. Using this window we can create other windows to get more information from the user or warn user before an important choice.

3.3 Interfaces

With revised class diagram we took out unnecessary interfaces and added them to the implementing class. And implemented interfaces while using design patterns. For example we introduces observer and subject interfaces for observer design pattern and Command interface by Command design pattern

Observer and Subject interfaces: These interfaces are used in order to implement observer design patterns between SoundButtons and DrumSequence. Inside the Observer interface there is a blank update() method that DrumSequence overrides, and inside the Subject interface there are setObserver() and notifyObserver() methods which are also blank and overridden by the SoundButton class. Using this overrides Each SoundButton object is now able to notify DrumSequence to update its information.

Command interface: This interface is used in order to implement command pattern design. Inside this interface there exist a blank execute() method that is overridden by all of the command classes, and by Command classes' designated operand classes calling this execute() method command pattern design is achieved. This way we made our code more extensible and understandable from an outside look.

3.4 External Class and Interfaces

MidiSystem: Javax.sound.midi.system class is part of a Java sound API that handles MIDI sound files. We use getSequence() and getSequencer() methods of this API in the DrumSequencer class to retrieve a midi.system sequence object from a midi file.

Sequence: Javax.sound.midi.sequence class is MIDI sequence in java. With the help of MidiSystem Class we get an instance of sequence inside the DrumSequencer class.

Sequencer: Javax.sound.midi.sequencer interface handling playback of MIDI sequences. open() method, opens the sequencer. start() method starts playback of midi sequences and lastly, setSequence(Sequence S) sets the sequence that will be played by the sequencer. With the help of sequencer class we play back the created sequences inside the DrumSequencer class.

Synthesizer: Javax.sound.midi.MidiDevice.synthesizer class is not directly used to play the notes in our project however we use it during the initialization of our sequence.

ScheduledExecutorService <<interface>>: General purpose of scheduler interfaces in java is for executing task with purposefully delayed and fixed rated times to avoid certain problems occurring. One of the branches they are useful is thread handling. In our project we used ScheduledExecutorService in order to play all the sounds in one column at the same time.

Button: Javafx.scene.control.Button class represents a class in javafx, that provides user interface buttons. In order to interact with the users we utilized this UI Button class in multiple classes (such as AbstractButton class and its subclasses, ChangeSoundButton, ChangeVelocityButton, ChangeDurationButton, RemoveSoundButton and addSoundButton.

Application of design patterns

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

	DP1:Singleton
Design pattern	Singleton
Problem	In the drum machine application, the DrumSequencer class is responsible for playing the current sequence that the user chose from the user interface. In order to do so, DrumSequencer creates a thread that plays the designated sounds when it reaches them. While working on threads it is important to take precautions that handle threads well so that there is no time mismatching between threads. The problem here is that if multiple DrumSequencer classes are instantiated by an error it might cause unexpected behaviour of the application.
Solution	The above mentioned problem can be easily solved by implementation of singleton design pattern. By making DrumSequencer a singleton object we now can guarantee that there will be only one instance of DrumSequencer class and there will be no multiple threads thus no thread mismatches.
Intended use	In the drum machine application, during the beginning of the application one DrumSequencer object will be reached using the “getInstance()” method from the DrumSequencer class. The client cannot create DrumSequencer objects because the constructor is private. In the continuation of our application the only way to create/ use a DrumSequencer object is to use the static method getInstance().
Constraints	The only constraint that this design pattern enforces is that there will be only one sequence playing at a moment. It will not be possible to create multiple sequencers that play at the same time asynchronously. However, this was the reason that we implemented this design pattern in this project. So this could be considered as a positive tradeoff.
Additional remarks	In the future if this project wanted to be changed to that it will be able to play two different sequences at the same time asynchronously this design pattern should be aborted.

	DP2: Observer
Design pattern	Observer
Problem	In this application, the user interacts with the sequence via SoundButton objects’ visual Buttons. However, to store the preset sequences and allow the user to save the sequences they desire, another approach is needed. Since data is stored in a “.txt” file, currentSequence is a List<String> and this List needs to be updated when there is a change in the SoundButtons. This problem is tackled with this pattern.

Solution	In the app, there is an Observer and a Subject interface. SoundButton implements Subject and DrumSequencer implements Observer. Subject has an Observer, so a SoundButton has a DrumSequencer and whenever onClick() of the SoundButton is executed, DrumSequencer is notified with notifyObserver(). By this way, whenever the state of the SoundButton is changed from triggered to not triggered or vice versa, DrumSequencer will be notified and take necessary actions (updates the list accordingly). This solution has made things easier for the project because whenever clearSequence() or randomSequence() is called, the SoundButtons change and without any additional calls, currentSequence is updated.
Intended use	The interface has a setObserver(T observer) and notifyObserver() functions. The Observer interface has an update(int row, int col, boolean triggerState) function. SoundButton and DrumSequence overrides these functions. At the main part of the code, when the soundButtonList matrix is initialized, all SoundButton objects call setObserver(DrumSequence seq). After this initialization, SoundButton objects are visible in a GridPane and the user can interact with them. Whenever a user clicks on a SoundButton, the onClick() function is executed since setOnAction(e->{onClick()}) is listening for user action. In the onClick() function, the observer is notified with notifyObserver() and update(int row, int col, boolean triggerState) is called with the row, column and the isTriggered values of the Subject SoundButton.
Constraints	The Observer interface has some constraints since its update function takes some specific parameters from the Subject. This means that the observing action for this interface is limited.

	DP3:Iterator
Design pattern	Iterator
Problem	In the drum machine application, there exist multiple user interface button objects from the nature of this project, namely SoundButton objects, since SoundButton objects basically creates a matrix. So, we need to create multiple SoundButton objects, acquire these objects' states, create a list from these objects' states and eventually update them if user decided to change any states. However this results in extensive usage of loops which are hard to follow up and cope up with.
Solution	We can solve the above mentioned problem easily and clearly by implementing iterator design pattern in this project. Doing so we can implement iterator inside the UI client code which needs to create multiple SoundButton objects since each of them has the attributes of their designated row and column numbers iterator design pattern will be helpful here. Moreover, with the help of iterator we can add Sound Button objects' states inside a list and use them when necessary without using any loops. Also we can implement iterator design pattern while clearing sequence and creating random sequences inside DrumSequencer and DrumSequence classes respectively. To summarise since we have multiple number of objects and know the amount that we want to create we can use iterator design pattern to make our code much more clearer and less error prone.
Intended use	Iterator design pattern is used multiple times in our projects to handle the general states of the sound button objects. Firstly, we intended to use iterator in the client UI code to create multiple SoundButton objects. As mentioned in the problem part to create matrix like iterator we created two iterators named rowIterator, colIterator and

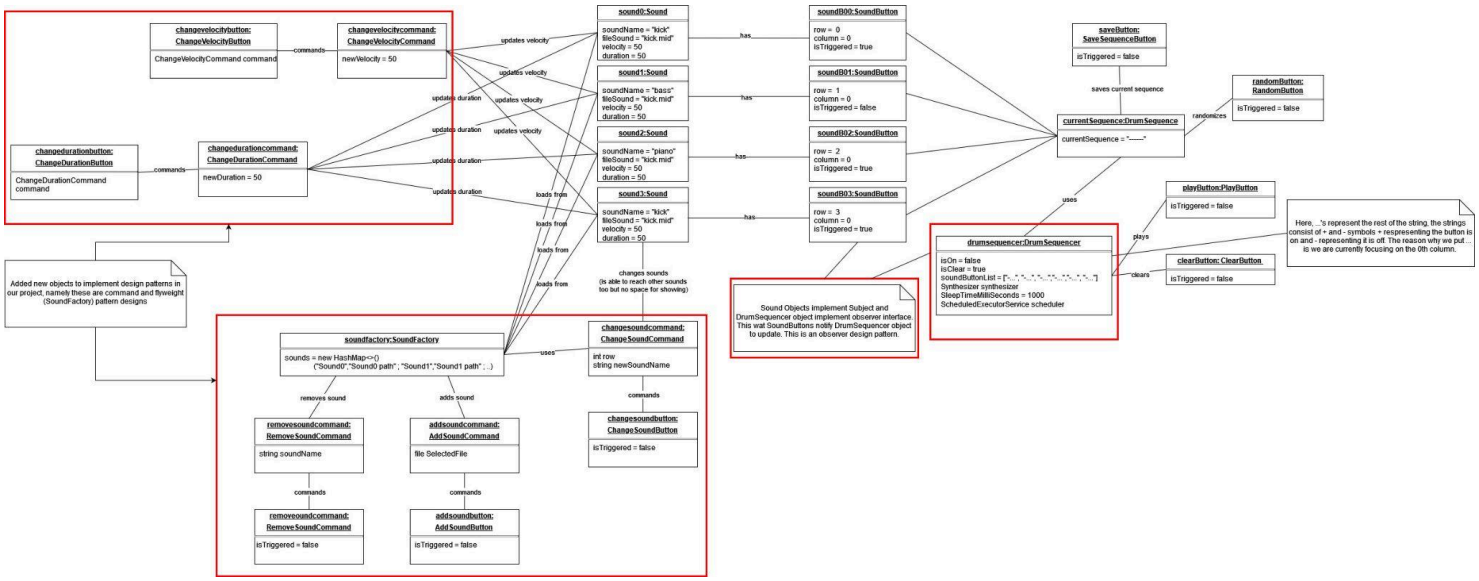
	iterated through them to create SoundButton objects. Moreover, we intended to use iterator design pattern inside DrumSequencer class's clearSequence() method by iterating through every SoundButton object and setting their state to notTriggered. Lastly we used iterator design pattern inside DrumSequence class's createRandomSequence method by iterating through every SoundButton object again and setting their states randomly. As mentioned by doing so we get away from extensive usage of loops.
Constraints	This design pattern does not result with any constraint to out application. Instead of this design pattern we could also be using for loops however avoiding unnecessary loops was the main reason to chose this design pattern.

	DP4: Command
Design pattern	Command Design Pattern
Problem	Our application contains many buttons. Each button had their business logic and user interface logic in their respective classes. This resulted in Tight Coupling between these two logics. The resulting code was hard to maintain and understand. Modifying and extending the buttons were challenging. Any addition to both interface and event handling logic was hard for us to both implement and use.
Solution	By encapsulating each button's behaviour within its corresponding Command object, and separating the user interface logic into distinct classes, we minimised the coupling between the user interface and business logic. In our main Application class, in alignment with the Single Responsibility Principle, parameters are passed to the Command objects rather than directly to the buttons. As a result, buttons are solely user interface components. Additionally, undo/redo functionalities can be very useful additions in the future for our remove sound, change sound/duration/velocity features. And implementing these additions are very easy with command design pattern implementations.
Intended use	This logic is shown in the first part of sequence diagrams for all of our Command Classes. For a specific example, ChangeSoundCommand object is defined with row and sequencer information in the main Application class. Subsequently, it is given as a parameter to the ChangeSoundButton object. Thus, the button object does not know parameters for logic handling and only holds a command object. When a user clicks CangeSoundButton object, they choose new sound's name. Upon confirming their selection with OK button, the newSoundName variable of the ChangeSoundCommand object is set and the execute() function within the command object is invoked. This function handles the actual process of changing the sound, utilising the gathered parameters. By using this setup, we made a clear separation between the button and the actual action, making it easier to change or add new actions later on and resulting in efficient handling of user interactions.
Constraints	One constraint is the need to create separate command classes for each distinct operation in the application. This can lead to an increase in the number of classes and can complicate the code management. Therefore, we used the Command pattern for the features with the harder handling logic and suitable for future extensibility of undo/redo actions.

	DP5: Flyweight
Design pattern	Flyweight Design Pattern
Problem	The problem is the potential inefficiency of creating and managing separate sound objects for each button within the same row, since buttons in the row share the same sound. For instance, when the Sound of one row is changed we need to create a new Sound object for every button in the row. Additionally, if we want to increase row and column numbers in the future this will result in a big performance problem.
Solution	By using the Flyweight pattern, we ensured that only one instance of the sound object was maintained for each row, regardless of the number of buttons referencing it. This approach eases the changes in the sound's duration and velocity, minimises memory usage and improves the performance of our application for further expansions by avoiding the unnecessary duplication of sound objects.
Intended use	The SoundFactory class serves as a central repository for holding and managing sound objects used throughout the application. When the application starts, the static block within the SoundFactory class initialises the sounds map with sound names as keys (Map<String,Sound>) using predefined sound objects. At execution, when a sound object is required, the SoundFactory's getSound method retrieves the corresponding sound object from the sounds map by using the sound name as a parameter. Additionally, the SoundFactory provides methods for adding, removing, and retrieving the names of all available sounds. These methods allow for dynamic and central management of the sounds at runtime.
Constraints	Centralisation of all sound management operations may cause some problems in the future. It might get harder to group sound if necessary in the future. Which means its constrain is that it limits extensibility. However, most design patterns have their tradeoffs and benefits. By applying flyweight in our current project we thought it is appropriate.

[optional] Revised Object Diagram

Author(s): Kerem Boncuk



This object diagram represents a snapshot of the Sequencer when the 0'th column is being played and the programme has just started. 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.

Revised from assignment 2's object diagram there has been some drastic changes in this diagram. Firstly new objects are introduced as shown in the diagram with red highlights. The main focus to add these new objects was to implement design patterns in our project and make our project much more extensible in the long run.

First of all in order to add sound control features (adding sound, removing, adjusting velocity…) We introduced new button objects and their command objects. The reason for adding command objects is to make our code readable and implement a single responsibility principle to our classes. Moreover, we changed the SoundList object with our SoundFactory object to implement a flyweight design pattern.

Another revision we made from assignment 2 is that while obtaining information from SoundButton objects to DrumSequence objects we introduce Subject and Observer interfaces to implement observer design pattern relationship between them. The reason we chose to do so is that it makes code much more readable and coped with when using notify() and update() methods. Lastly,

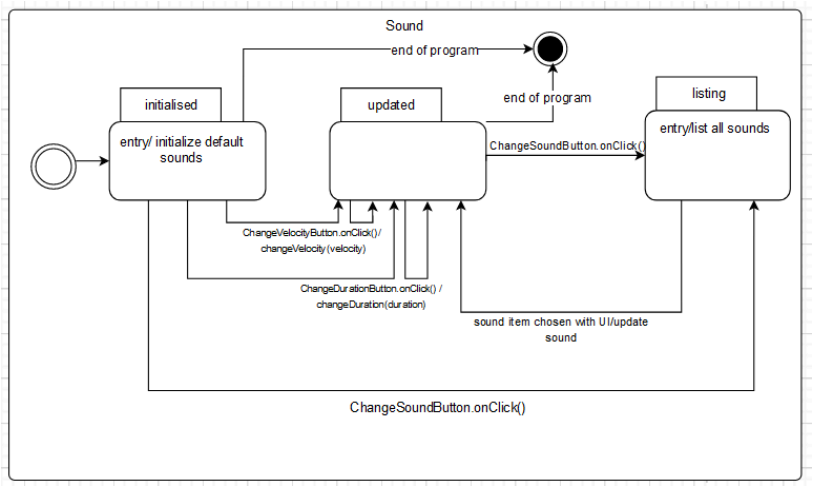
our DrumSequencer we added new attributes to the DrumSequencer object because extended classes, libraries and interfaces were needed to create sequences and handle MIDI sound files. In order to do that we added Sytnthesizer and scheduler attributes to it.

To summarise how object cycle works, through sound adjustment buttons it is possible to command “Command Objects” to execute certain methods and manage sound files. Also all sounds are loaded from SoundFactory flyweight at the start of the programme. Moreover, familiar to assignment 2 each SoundButton object has their distinct Sound objects if they are in a different row in the grid. To continue, in our revised object diagram DrumSequence object and SoundButton objects are connected with an observer design pattern. This way each SoundButton object notifies the DrumSequence object if any changes happen to them. Lastly after we reached the DrumSequence object this object is played back by the DrumSequencer object. The DrumSequencer object is managed by the user through the help from UI buttons that are similar to the object diagram from assignment 2.

Revised state machine diagrams

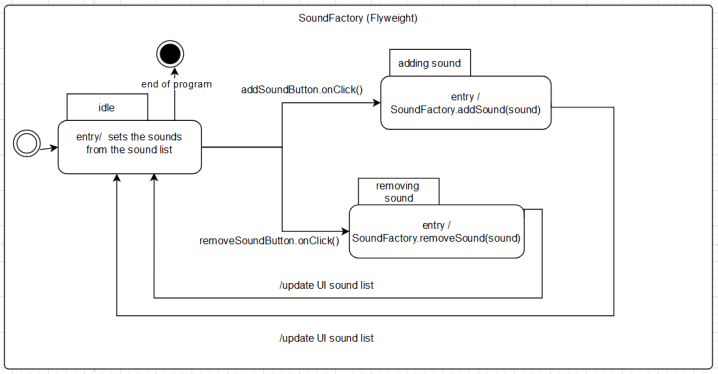
Author(s): Kerem Boncuk, Sena Deniz Avukat

Sound Class State Machine: The state machine diagram shows the internal behavious of the Sound Object. From the starting state, the Sound moves to the “initialized” state where Sound’s are loaded from the default sounds from the static map inside the SoundFactory class. From the “initialized” state onClick() method of ChangeVelocityButton class changeVelocity() method is called as an action and the state machine switches to “updated” state. Similarly from “initialized” state, the state machine switches to “updated” state with the onClick() method of ChangeDurationButton class while taking changeDuration() method as an action. However, from “initialized” state onClick() method of ChangeSoundButton class switches state to “listing” and with entry of the “listing” state all sounds get listed and with the “sound item chosen with UI” event state machine switches to “updated” state with update sound action. Also from the “updated” state we can revisit the “update” state again or “listing” state according to its onClick() method. Lastly, according to feedback from assignment 2, it is clear that Sound object reaching end state from the updated state, however we included a reach possibility from initialized state to end of program incase user closes the program without any changes made in the sound object.

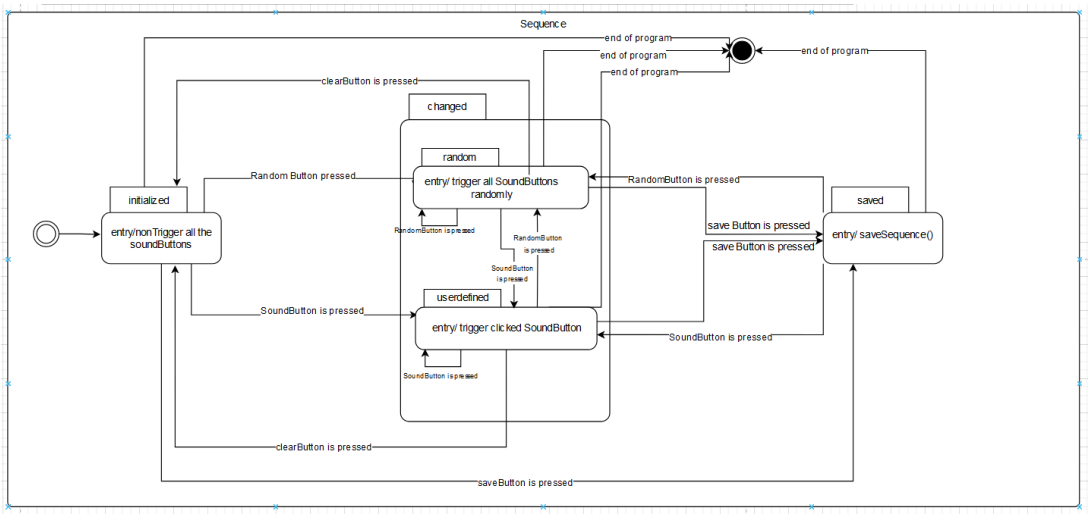


Sound Factory Class State Machine Diagram: This state machine gives information about possible internal behaviour states of the SoundFactory object. SoundFactory object’s behavioural states are crucial to understand how default and user defined sounds are implemented in our application since this class stores sound files. Following part verbally describes the state's behaviours: From the starting state, the SoundFactory moves to the “idle” state where Sound objects are set using the sound list. From this state, with the onClick() method of addSoundButton class, the state changes to “adding sound” state and addSound(Sound) method is called as the machine enters this state. From there the SoundFactory class immediately returns to “idle” state with the action of updating the UI

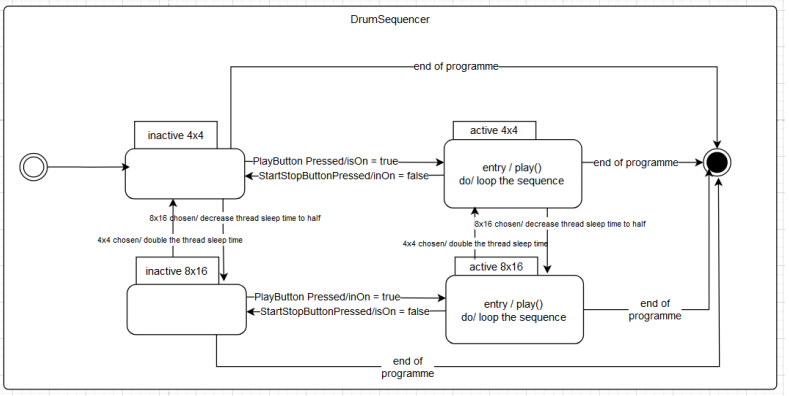
sound list which is stored in the static map inside the SoundFactory class. Again from “idle” state with the onClick() method of removeSoundButton class the state changes to “removing sound” and removeSound(Sound) method is called as the machine enters this state. From there the SoundFactory class immediately returns to “idle” state with the action of again updating the UI sound list. The machine switches to the end state from the “idle” state when the program ends, because “idle” state is reached immediately after “adding sound” and “removing sound” states and machine will be in the “idle” state in the end of the program.



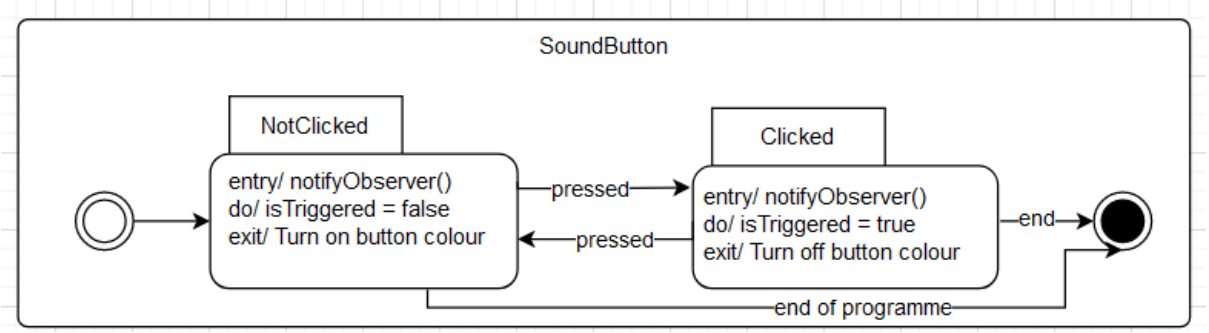
DrumSequence Class State Machine: This machine mainly represents how our Drum Machine’s behaviours would change according to the user's inputs. As mentioned in the previous assignment we did not use many guards because we want users to be able to change the current sequence as freely as possible for their experience. And we even removed the guard that checks whether a saved sequence already exists in the sound map because of the idea that users might want to save the same sequences multiple times. From the starting state, the DrumSequence moves to the “initialised” state and during entry we set all the SoundButtons to not-triggered states. This “initialised” state can be reached from all other states when a “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. With the entry of the “random” state all SoundButtons are triggered randomly 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 by triggering the clicked SoundButton object 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. We removed any guards here in order to let users to save multiple instances of the same sequence.



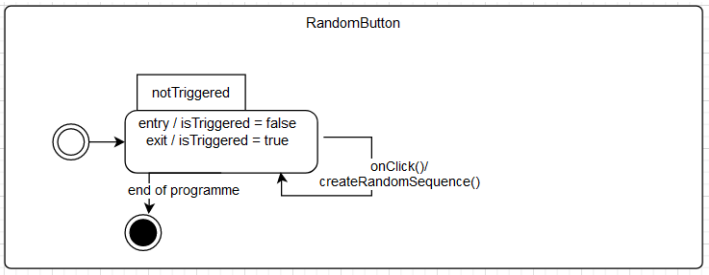
DrumSequencer State Machine: This state machine represents the behaviour of our Drum Machine's inactive/active and tempo status. Our revised state machine introduces two new states for the other tempo option. "4x4" is our first tempo option and "8x16" is the second one that plays the sequence doubly faster. From the starting state the DrumSequencer moves to the "inactive 4x4" which is default for our application. With the "PlayButton pressed" events the drum machine will be able to toggle between inactive and active status whilst toggling isOn attribute as an action. Since the play() method is continued whilst only isOn attribute is true there is no need to use any stop methods inside this state machine. Moreover, the state machine will be able to toggle between tempo options with the "4x4 chosen" or "8x16 chosen" events whilst doing so will manage tempo by rearranging thread sleep time as an action. The DrumSequencer will be able to reach the end state from any existing states if the program ends.



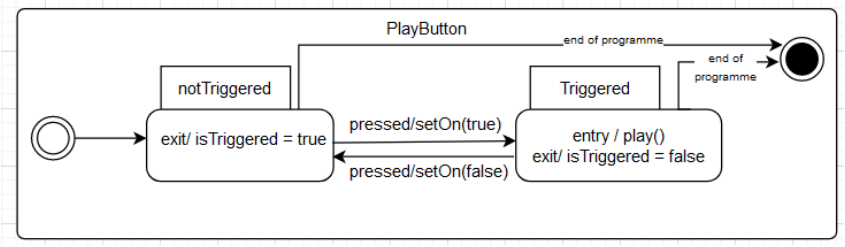
SoundButton Class State Machine: All SoundButtons will have two states: they are either in the state of "NotClicked" or "Clicked". When they enter the "Clicked" or "NotClicked" state they will call the method called notifyObserver() and the DrumSequence class will be updated accordingly with the observer design pattern. Moreover, while in the "Clicked" state isTriggered value will be set to true and vice versa in the "NotClicked" state. Lastly, SoundButtons will turn off their button colour when they exit the "Clicked" state and turn on their button colour when they exit the "NotClicked" state. The sound buttons will be able to reach the end state from any state when program ends.



RandomButton Class State Machine: This revised state machine represents the behavioural change in the RandomButton object. Our revised RandomButton class has only one state called “NotTriggred” after our evaluation of its implementation. From the start of the application RandomButton enters the “NotTriggered” state and isTriggered attribute is set to false. With the onClick() method of RandomButton class, the object leaves the state and isTriggered flag becomes true and the createRandomSequence() method from the DrumSequencer class is called. And when object reenters to the state isTriggered attribute reset to false. RandomButton can reach the end of the state if the program ends from the only state that it has.



PlayButton Class State Machine: This state machine represents the behavioral change in PlayButttton class. This class has two states namely “notTriggered” and “Triggered”. At the beginning of the application PlayButton starts in the “notTriggered” state. By pressing the PlayButton, “notTriggered” state is left and isTriggered attribute set to true. Also setOn(true) action is called from the DrumSequencer class. When entered into “Triggered” state, the play() method from the DrumSequencer is called and the sequence starts playing. If pressed again to PlayButton, with the exit of “Triggered” state isTriggered attribute becomes false and the setOn(false) setter method is called from the DrumSequencer class as an action. By doing so, arranging isOn attribute with the setOn setter and we can play and stop the sequence as we like without any stop() method. Lastly, PlayButton class is able to reach to end state if end of programme happens

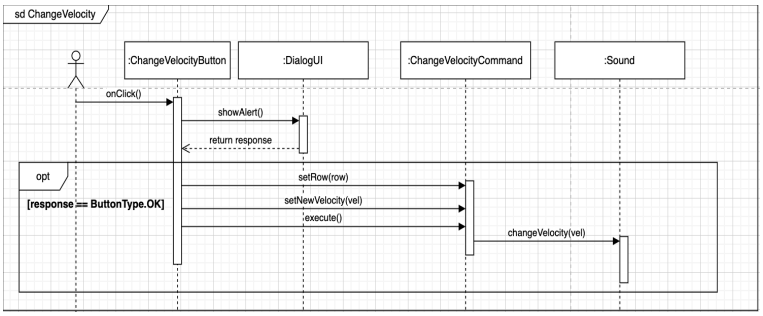


Revised sequence diagrams

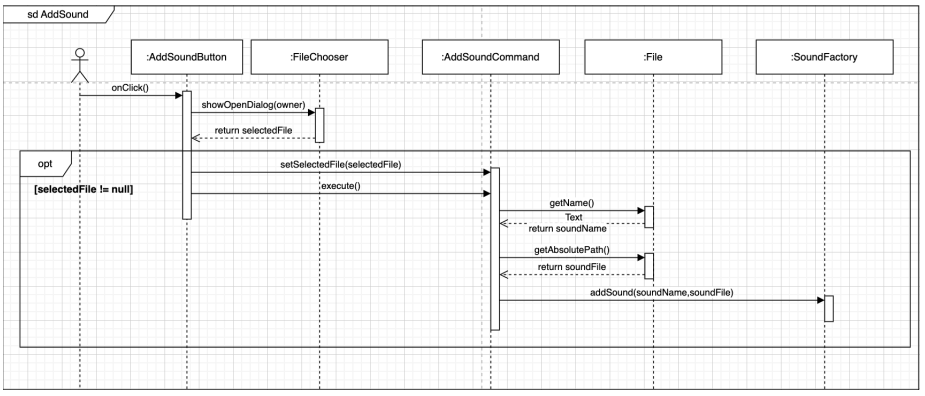
Author(s): Ceren Duru Çınar, Sena Deniz Avukat

In our sound changes sequence diagram for Assignment 2, we represented all the changes in a sound in the same sequence diagram and thought that we can handle the cases with alt fragment and enumerations. However, as our application is highly user dependent, we need users to press buttons to make those changes. For the sake of the user experience, we designed separate buttons for all that purposes and that buttons are observers of the click action. Since there is no way for us to check which button is pressed, we separated these changes as different sequence diagrams.

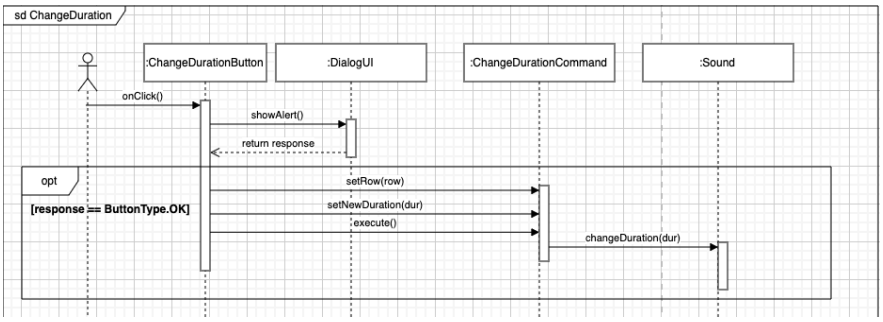
ChangeVelocity: When the user presses ChangeVelocityButton, the onClick() function is executed. Here, DialogUI's showAlert(...) function is called and a dialog screen is opened. If the user clicks OK button (response is true), ChangeVelocityCommand's execute() function is triggered and velocity is changed by ChangeVelocity(vel).



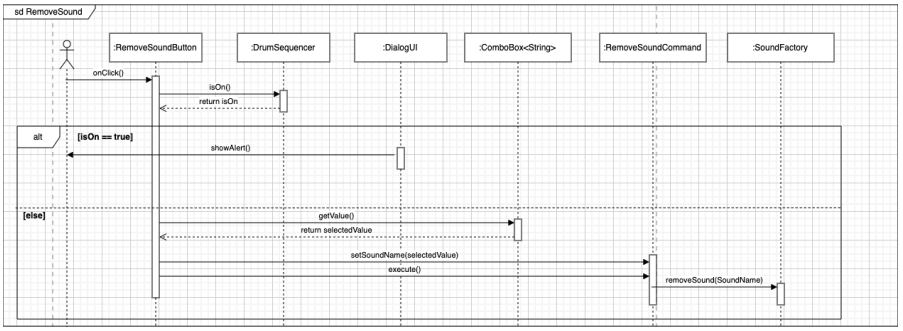
AddSound: When the user presses AddSoundButton, the onClick() function is executed. Here the showOpenDialog(owner) function of the FileChooser is executed. Then FileChooser returns selectedFile. If the selectedFile is not null, setSelectedFile(selectedFile) and execute() of AddSoundCommand are executed. Then selectedFile's name and absolute path is obtained via getters. Then addSound(soundName, soundFile) of SoundFactory is executed.



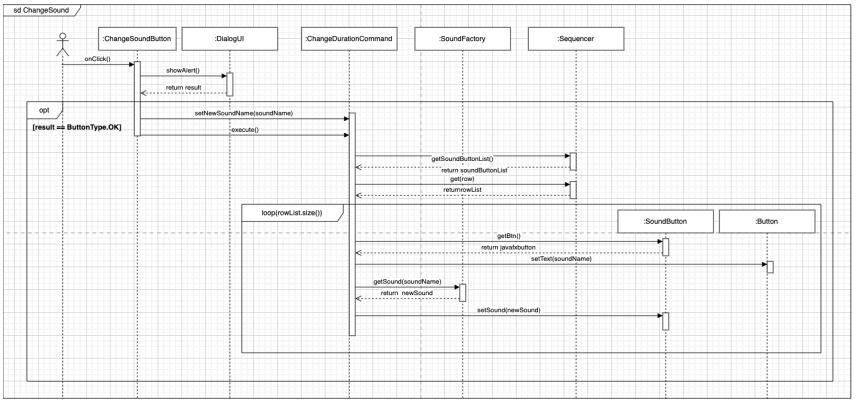
ChangeDuration: When the user presses ChangeDurationButton, the onClick() function is executed. Here, DialogUI's showAlert(...) function is called and a dialog screen is opened. If the user clicks OK button (response is true), ChangeDurationCommand's execute() function is triggered and duration is changed by ChangeDuration(dur).

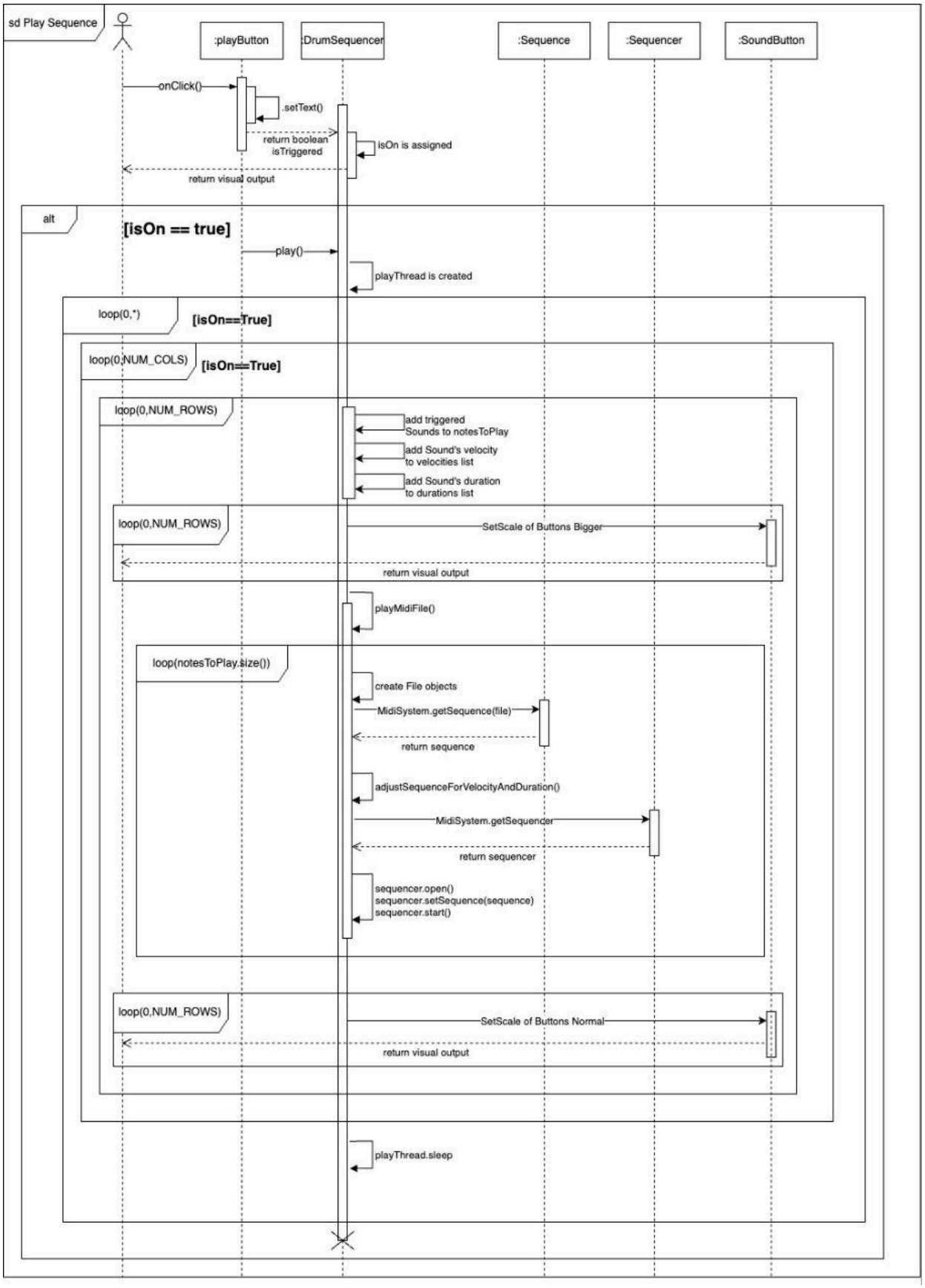


RemoveSound: When the user presses RemoveSoundButton, the onClick() function is executed. If DrumSequencer is on, the user is alerted saying that the sound cannot be removed when the sequencer is on. Else, the selected value from ComboBox<String> is obtained with getValue(). This selected value is used to set the name of the Sound to be removed and the execute() function of RemoveSoundCommand is called. Finally, removeSound(soundName) of SoundFactory is called.



ChangeSound: When the user presses ChangeSoundButton, the onClick() function is executed. Here, DialogUI's showAlert(...) function is called and a dialog screen is opened. If the user clicks OK button (result is true), setNewSoundName(soundName) and execute() of ChangeSoundCommand are called. Then the rowList of the soundButtonList is obtained. Here, we loop on all the SoundButton objects in the rowList and execute getBtn() and obtain a javafxbutton.We change the soundName of that button with setText(soundName). We obtain the corresponding Sound from SoundFactory flyweight and call the setSound(sound).




sd Play Sequence
:playButton
:DrumSequencer
:Sequence
:Sequencer
:SoundButton
onClick()
.setText()
return boolean
isTriggered
isOn is assigned
return visual output
alt
[isOn == true]
play()
playThread is created
loop(0,*)
[isOn==True]
loop(0,NUM_COLS)
[isOn==True]
loop(0,NUM_ROWS)
add triggered
Sounds to notesToPlay
add Sound's velocity
to velocities list
add Sound's duration
to durations list
loop(0,NUM_ROWS)
SetScale of Buttons Bigger
return visual output
playMidiFile()
loop(notesToPlay.size())
create File objects
MidiSystem.getSequence(file)
return sequence
adjustSequenceForVelocityAndDuration()
MidiSystem.getSequencer
return sequencer
sequencer.open()
sequencer.setSequence(sequence)
sequencer.start()
loop(0,NUM_ROWS)
SetScale of Buttons Normal
return visual output
playThread.sleep

Play Sequence: In our PlaySequence in the Assignment 2, Sequence and Sequencer classes were defined by us but in this Diagram they are predefined Java Classes. When user presses the Play button, the button's name changes from “Start” to “Stop”. Then, onClick() method sends a message to the *DrumSequencer* to assign isOn variable accordingly and calls the play() method of the DrumSequencer class. With an alt frame, whether to play or stop the sequence is decided. If the isOn == true, the Drumsequencer’s play() function will firstly create the new playThread for the audio playback. And after the drumsequencer object will step inside a loop whose condition is also isOn == true, this way if isOn becomes false the sequence will stop.

Inside the main sequence loop frame there will be another sequence loop frame that repeats column number of the grid times. The main reason for this loop is to create a sequence by sleeping the thread at the end of this loop. Inside this frame there will be another frame that repeats the number of row of the grid times. Inside this loop, the soundbutton objects of the current row’s are checked if they are triggered or not and added to notestoplay for that point of sequence, with the help of “add triggered sounds to notesToPlay” message, also sounds velocity and duration are added to their list by the “add Sound’s velocity to velocities list” and “add Sound’s duration to duration list” respectively incase if user changed these values. Afterwards with the help of another loop frame all buttons scales are set to a little bigger than their usual scale in order to show to the user that where is the current sequence is at with the help of “set Scale of buttons bigger” message to soundbutton objects and sound objects respond to the ui and updates their scales.

After end of setting scales, drumsequencer objects sends a message to itself and runs the playMidiFile() method. This method opens another loop that repeats the size of the notesToPlay number of times. Inside the loop sound files are created with the “create file objects” message and drumsequencer object sends a message named MidiSystem.getSequence(file) to sequence object in order to get the sequence and jaba plugin sequencer object responds the message with the return of the sequence. After getting sequence adjustSequenceForVelocityAndDuration() method is called from drumsequencer to itself. This method checked if there are any changes in velocity or duration of the sounds and adjusted them if there any changes. Then familiar with the java plugin sequence object, drumsequencer objects sends a message called MidiSystem.getSequencer() to java plugin sequencer object to get the sequencer. And sequencer object responded returning the wanted sequencer object. After obtaining sequencer and sequencer, drumsequencer object opened the sequencer set the wanted sequence into it and played back the sequencer with the following messages respectively sequencer.open(), sequencer.setSequence(sequence) and sequencer.start(). After repeating this steps for every notes that are going to be played for a certain column in the grid the loop is left.

Lastly every soundbuttons on the current row was set a little bigger that the usual to show the user in which part of the grid the sequencer was. After the loop previously mentioned another loop is created just to set the scales of the soundbutton objects back to normal, drumsequencer objects sends a message called “setScale Buttons Normal” to soundbutton objects and they return the respond to user interface by setting the scale of the buttons back to normal. Lastly the thread created at the begging of this loop is slept for a duration to create somewhat sequence like duration. And the outer loop is repeated as long as isOn attribute is set to true.

Implementation

Author(s): Kerem Boncuk, Ceren Duru Çınar

Maximum number of pages for this section: 4

We really tried to stick to our UML diagrams when coding the project. However, one of the feedbacks we received was that our model was a little too complicated and we may not be able to implement them with 3 people. Our main focus was simplifying the implementation while not giving up much from the planned features.

From the start of the project, our first challenge was to import JavaFx libraries. Since we are using github and template of a current project, it was not an option to create a new Javafx application. We overcame this problem by creating a Javafx and importing our whole project inside this module. Doing so we managed to implement Javafx classes without creating our project initially as an Javafx application.

While coding the classes, we tried to implement the “steady and easy” classes first and tackle others later. While adding them, we also added other classes that are necessary for design patterns and that will ease our implementation. After our implementation, we added new classes and interfaces to our model that we could not foresee.

We did not change our state machine diagrams that much and their changes are mostly implemented inside functions like setters or onClick() functions. Even though we did not change the diagrams’ logic, to implement state changes in the Sound class, we needed to add more functionalities like Command design pattern. We used this Command design to adjust sound properties such as velocity, duration and sound file itself. Yet, this did not change our model.

Our package diagram needed to be revised especially to get rid of circular imports and our newly added logics. Thus, we coded and changed our package diagram. When coding the DrumSequencer’s play() we really tried to stick to the idea behind our sequence diagram but we also added some improvements. We still get the activated rows in each column and add them to a list and then play them in a loop. We needed to add Thread and scheduler logic to improve the sound experience. Sound changes in the sequence diagram could not be implemented as we planned. We saw later that our plan on sound adjustment features were not feasible. We improved it with the use of UI and click actions and separated the diagrams for each change using Javafx UI tools. Moreover, we removed the need of guards while coding because we are using a slider and the user can only choose between values we predetermined, so there was no need to check for sound properties after adjustment. Even Though giving strict restraints to users is not good for application’s creativity, it is important that we provide a healthy and working application to users. By restraining users during adjustment of the sounds we escaped from possible bugs that could be caused by user inputs.

While coding we benefited from our class diagram only at the beginning of our project. After implementing the already taught classes to our project and finishing our fundamentals of our project’s code, the previous class diagram was irrelevant to our project because we needed some improvements and implementation of design patterns was causing too many changes to our previous class diagram. However, to keep up with the changes we updated our class diagram with our changes because it is much easier to understand and remember previous changes from the class diagram.

The first key solution that we implemented in our project is how the DrumSequence class was being updated when SoundButton objects were triggered. During assignment 2, we thought to store the whole information as a string and make a copy of this string when a SoundButton object is Triggered. However, we quickly realized that this implementation was causing a bunch of unnecessary copies and would probably result in inefficient results. And we decided to send the information through an observer design pattern. This way we made our code much more extensible and easy to understand.

Another key solution that we implemented in our project is how the sound files were being stored. We were planning to add the MIDI sound files directly through file however this way it would not be possible for sound files to be changed. In order to do so, we created a new class called SoundFactory and implemented a flyweight design pattern to this class. The reason that we applied flyweight is to make our code much more extensible. Since the same row of SoundButton objects play the same sound it would be meaningless to store these sound files more than once. With this idea we implemented flyweight pattern design and kept the sound files inside a static map inside the SoundFactory class.

One more key solution that we implemented in our project is how were we going to adjust sound properties. Our feature required a way to adjust sound velocity, duration, adding, removing and changing sounds. It was important to understand how we were going to link the user interface and our computation in our code. In this regard we introduced one more class to each class as a command class and implemented Command design pattern to linkage. This way our coe became more extensible regarding that we could implement more features that require user input and link them using Command design pattern in the future development of our application.

The last key solution that we implemented in our project is about how to create the executable jar file. Even though creation of a jar file is somewhat easy most of the time, in our case we could create our project as a JavaFx application since we had crucial problems whilst creating the jar file. Our resource files were not reachable in the jar file because of the way we introduced the javafx imported classes which were implemented as a module. So in the end we found a solution by redirecting the source path that we use in the folder where the jar file exists.

The location of the main java class code: src/src/main/java/drum/src/DrumMachineMain.java

The location of the executable jar file: Executable_Jar/software-design-vu-2024.jar

Link of the short video: https://youtu.be/2LdTjuDK4ks

Time logs

Team number	16		
Member	**Activity**	**Week number**	**Hours**
Ceren Duru Çınar	Worked on creating classes	6	3
Sena Deniz Avukat	Worked on creating classes	6	3
Kerem Boncuk	Worked on creating classes	6	3
Ceren Duru Çınar	Worked on sound playing logic	6	3
Sena Deniz Avukat	Worked on UI design	6	3
Kerem Boncuk	Created interfaces	6	3
Ceren Duru Çınar	Worked on observer and iterator design patterns	7	4
Sena Deniz Avukat	Worked on flyweight and command design patterns	7	4
Kerem Boncuk	Worked on singleton design pattern	7	4
Ceren Duru Çınar	Implemented Sound Control feature	7	5
Sena Deniz Avukat	Implemented Expandible Sound Library	7	5
Kerem Boncuk	Wrote report	7	5
Ceren Duru Çınar	Worked on Sequencer feature	7	4
Sena Deniz Avukat	Worked on Sequencer feature	7	4
Kerem Boncuk	Worked on Sequencer feature	7	4
		TOTAL	57