Bilkent University

Department of Computer Engineering

Senior Design Project

Coda

Low Level Design Report

Merve Kılıçarslan, Yağız Efe Mertol, Ege Özcan, Çağla Sözen, Murat Tüver

Supervisor: Prof. Dr. Uğur Güdükbay
Jury Members: Prof. Dr. Halil Altay Güvenir, Prof. Dr. Fazlı Can

Innovation Expert: Prof. Dr. Veysi İşler (SimSoft)

Low Level Design Report
December 30, 2019

This report is submitted to the Department of Computer Engineering of Bilkent University in partial fulfillment of the requirements of the Senior Design Project course CS491.

Contents

Project High Level Design

Coda

1 Introduction

Music as a form of art has been the common interest of people used to express feelings and identity through a composition of rhythm, timbre and melody. In different forms and sounds by making use of the variety of instruments, music is present in almost every context for numerous purposes. Besides the pleasure of listening, it's been proven that playing instruments have positive effects on brain development, especially for spotting statistical patterns enabling the learner to better predict what would happen next in a pattern, so every child has the right to better themselves with the help of instruments insomuch as discovering their musical talents [1].

Over the last 20 years, the number of children learning to play an instrument or playing an instrument has increased significantly [2]. However 26% of children and 49% of adults in the UK stated that they've given up playing instruments although they've learned to play or started to [2]. Most common reasons for this are loss of interest, instrument cost, lesson costs and competing pressures from school [2]. Furthermore, the fact that some instruments are highly immobile by nature makes practicing very challenging for both individuals and for groups of people who practice together. In most cases, instruments become idle and forgotten because of the impracticalities mentioned. As a result, buying instruments may be seen as an unnecessary expense. When the cost of learning instruments and the instruments itself is taken into consideration in addition to immobility, it can be stated that instruments can be made further accessible.

Therefore, there is need for solutions to make playing and learning instruments more sustainable by making them more accessible in several aspects like cost and mobility and we believe that with the increasing number of smartphones, mobile phones can be used to address this problem.

This report explains the details of the proposed Low-Level System Design including *Object Design Trade-Offs, Interface Documentation Guidelines, Engineering Standards, Packages, Class Interfaces* with the help of UML for proper documentation of the system and clarity of the system design. We aim to come up with a System Design that complies completely our requirements as explained in the Analysis Report to construct a system that fulfills the need of our users while providing a usable and efficient application.

1.1 Object Design Trade-Offs

1.1.1 Functionality vs. Usability

Usability is essential for *Coda* since the fundamental *idea* of Coda is to mimic the experience of playing an instrument intuitively. The success of the system is determined by the ability to mimic such real-life, tangible experience. To be able to allow the user to interact with the instruments as intuitively as possible, some functionalities will be left out. Nevertheless, all the provided functionalities will be easy to use. Hence, usability is favored over functionality during design.

1.1.2 Use Time vs. Usability

As discussed in *Section 1.1.1* usability is an essential for *Coda* for intuitive interactions with the instrument. To be able to allow the user to interact with the instruments as intuitively as possible *Coda* creates an immersive virtual environment, as a result use time of *Coda* is limited to avoid loss of the notion of spatial awareness. Hence, usability is favored over use time during design.

1.1.3 Cost vs. Performance

As discussed in the previous documents, one of the distinguishing features of Coda is not requiring any advanced or additional hardware such as controllers or advanced VR glasses. This is for serving one of the primary missions of *Coda* which is to ease access to instruments in terms of cost and to be accessible to all financial levels. Therefore, although using additional hardware or camera would increase performance, no additional hardware is required in the design other than the smartphone itself and a simple VR cardboard. Hence, cost is favored over performance during design.

1.1.4 Extendibility vs. Mobility

To serve one of the primary missions of *Coda* which is to ease access to instruments in terms of mobility, mobility is an important aspect of the system design. Although *Coda* could have been more extendible in terms of addition of different instruments, instruments which require a different FOV than the one provided by the VR cardboard and the smartphone are left out since such an instrument would require the use of an external camera such as guitar, violin and so forth. Hence, mobility is favored over extendibility during design.

1.1.5 Compatibility vs. Extendibility

Although compatibility of *Coda* with multiple operating systems is important for increasing ease of access, considering the variety of phones with Android as the operating systems is significantly higher than other operating systems we consider Android to be a sufficient platform. Furthermore, extendibility of the system is important for providing multiple instruments and providing updates for solving bugs and increasing performance since performance and usability is of important aspects of *Coda*. Additionally, *Coda* uses novel state-of-art CV techniques and API's which were developed very recently, so it is important to keep the system extendible with updates that will enhance *Coda*. Hence, extendibility is favored over compatibility during design.

1.2 Interface documentation guidelines

In the documentation of *Coda*, classes are explained in detail with their names, properties and methods. For each class, there exists a *ClassName* indicating the name of the class, a list of attributes and a list of methods. All attributes of the class are indicated along with their *access modifiers*, *types* and *names*. Similarly, all methods of the class are indicated along with their *access modifiers*, *types*,*names* and a list of *parameters*.

An outline of the class descriptions in this document is given below:

class *ClassName*			
Class Description			
Attributes			
Access Modifiers	*Type*	*Attribute Name*	
Methods			
Access Modifiers	*Return Type*	*Method Name*	*Parameters*

methodName: Method description.

Table 1: Class Description Outline

1.3 Engineering standards

All documentation of *Coda* including the class interfaces, diagrams, scenarios, use cases, subsystem decompositions and hardware descriptions follows the UML guidelines [3]. All citations in the documentation including this report follows the IEEE citation format [4]. These formats are preferred for standardization and for the readability of the documentation.

2 Packages

2.1. Updated Package Diagram

Subsystem Decomposition of *Coda* is updated as follows according to design changes.

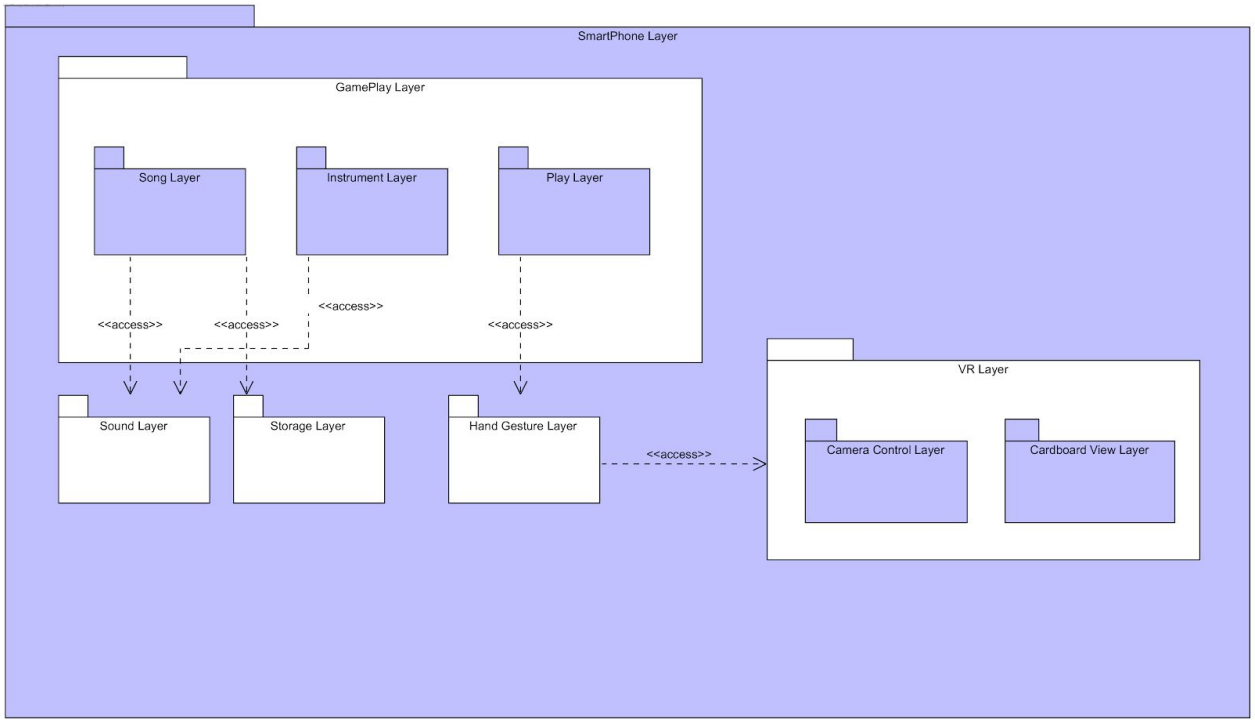

Figure 1: Subsystem Decomposition *Coda*

Coda's architecture is limited to the smartphone itself. All sublayers are within the *SmartPhone Layer*, which is the most comprehensive layer of the system. In this architecture, there is no distinction between the client side and the server side since both sides depend on the local system only. However this architecture may be subject to change if there occurs issues about the processing power of the smartphone.

Detailed description of packages and classes can be found in the proceeding sections.

2.2.Play Layer

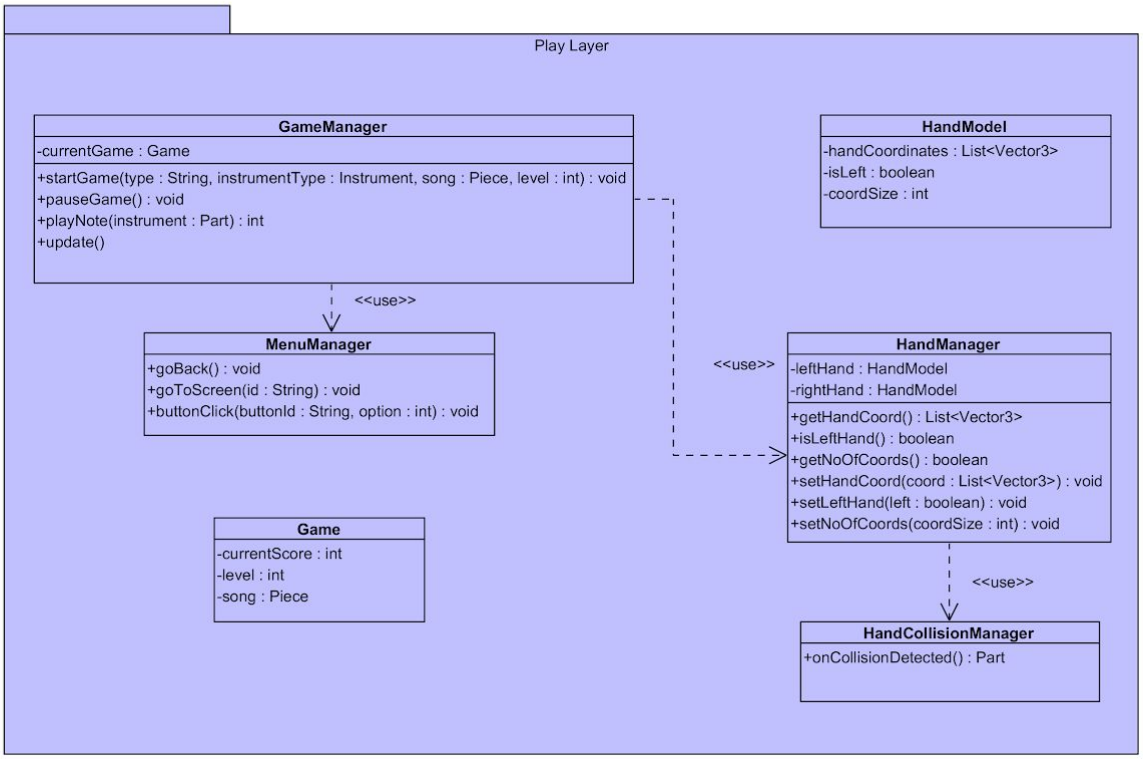

Figure 2: Subsystem Decomposition for the Play Layer of *Coda*

All game operations and user interactions are handled in this layer. This layer will handle the interactions of the user with menus, initialization of the game, representation of the hands as an object and interaction with the game objects.

Game: A model class for the game itself. *Song* and *level* chosen by the user is stored in this class as well as the *current score* of the user.

GameManager: A controller class for the *Game* model. Instantiates the *Game* and performs all other Game related operations like pausing, updating and playing a note by communicating with the relevant layers (i.e. HandGesture Layer).

MenuManager: A controller class for the *Menu* in the application. Handles menu interactions as changing screens and clicking menu buttons according to the button clicked.

HandModel: A model class for representing and rendering the *hands* of the user according to the data received from the HandGesture Layer. Hands are represented with a list of maximum 21 3-dimensional coordinates for Hand Landmarks as received from MediaPipe. This model

also stores which hand is the model representing (i.e. Left or Right), and number of Hand Landmarks currently received for the hand.

HandManager: A manager class for the *Hand* model. Instantiates the hands of the user as two *Hand* objects one being right and the other being left. Contains the relevant methods for getting and setting the data in the Hand Model.

HandCollisionManager: A manager class for detecting collisions between either of the hands of the user and the instrument in the game, and then managing the interactions with the relevant classes responsible from the auditory and visual feedback.

2.3. Sound Layer

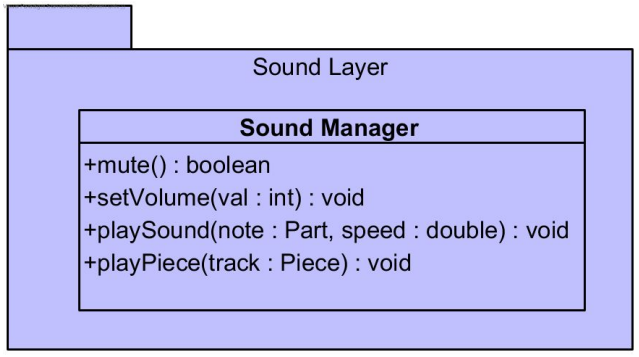

Figure 3: Subsystem Decomposition for the Sound Layer of *Coda*

All sound operations are handled in this layer. This layer will handle the playing of the songs, auditory feedback given by the application to the user depending on the instrument played and the speed of the interaction. This layer also handles the sound settings in the application.

SoundManager: A controller class for the sound operations in the application. This class handles the sound settings like setting the volume of the auditory feedback and muting all sounds. This class also controls the instrument piece and hit speed dependant sounds according to user interactions. Finally, recorded and library pieces are played in this class.

2.4. Storage Layer

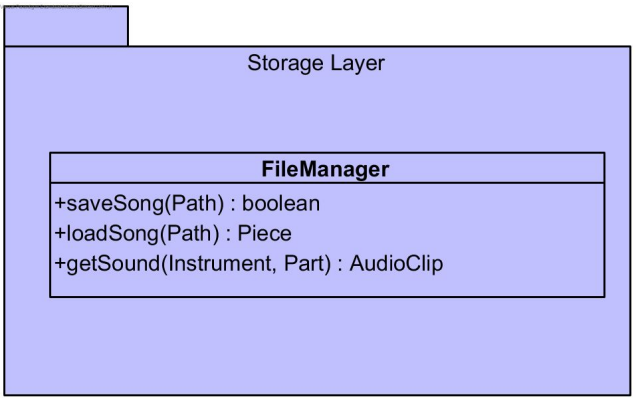

Figure 4: Subsystem Decomposition for the Storage Layer of *Coda*

All storage operations are handled in this layer. This layer will handle the saving of recorded songs, loading saved and library songs. This layer also handles the loading the sounds defined for each instrument part.

FileManager: A controller class for the storage operations in the application. This class handles the operations regarding the file system of the smartphone. Using this class, recorded songs are saved to a path, saved and library songs are loaded from a path. Sounds defined for each instrument part which are again stored in the file system are accessed using this class.

2.5. Instrument Layer

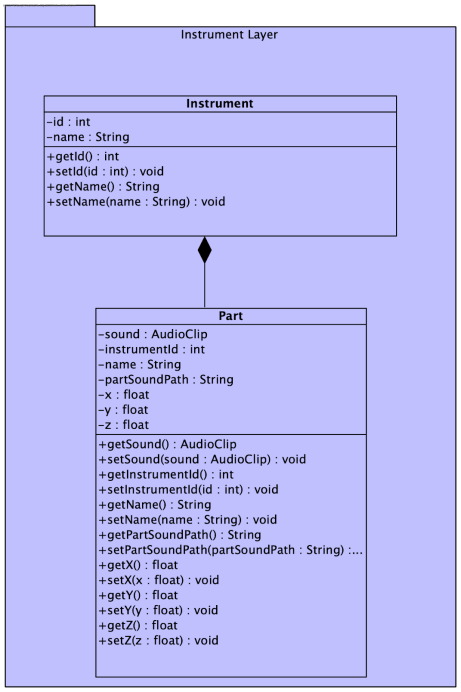

Figure 5: Subsystem Decomposition for the Instrument Layer of *Coda*

This layer represents the *Instruments* in *Coda*. It contains the *Instrument* class which is composed of the *Part* class. This layer contains the instrument structures definitions and the parts that the instruments are composed of, providing the relevant functionalities depending on the instrument structure.

Instrument: A model class for the general instrument structure in *Coda.Id* and *name* of the instruments are commonly contained in this structure. Each instrument is composed of *Part*s that provide instrument specific functionalities.

Part: A model class for constructing instruments by parts in *Coda.* Each *Part* of an *Instrument* has a unique functionality. These unique functionalities are contained as structures in this class. This class contains the *name*, *instrumentId*, *path* to the sound in the file system, *sound* as audio and 3-dimensional *coordinates* of the *Part*.

2.6. Song Layer

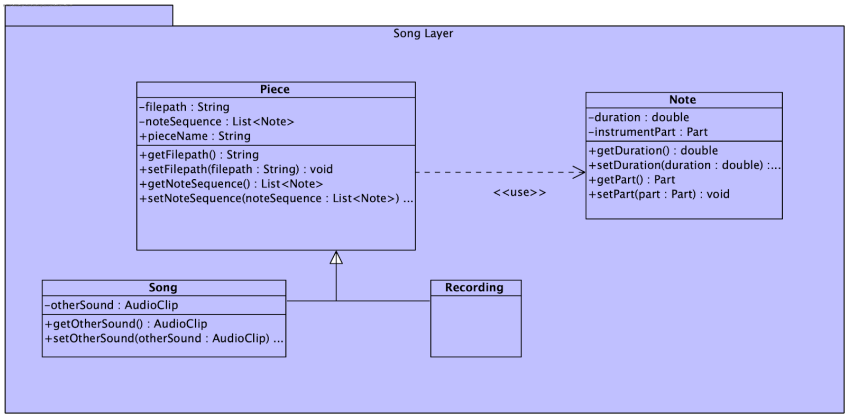

Figure 6: Subsystem Decomposition for the Song Layer of *Coda*

This layer represents all types of *Songs* in *Coda*. It contains the *Piece* class which is used for representing both library *Songs* and *Recordings*. This layer consists solely of the models which will be discussed below.

Piece: A model class for the general Piece structure in *Coda.* Attributes that are common to all pieces in Coda, either recorded by the user or in the library by default, are contained in this class. All *Piece*s have a *filepath* in which they are stored, a *noteSequence* represented as a list of *Note* objects which are crucial for the functionalities of *Instrument*s.

Song: A model class for the library Songs that are stored by default in *Coda.* In addition to the *noteSequence* that depends on the *Instrument* played, this class contains sounds of the other instruments in the Song.

Recording: A model class for the *Recordings* made by the user*. Recordings* are stored as a sequence of *Notes* played by the user during the game.

Note: A model class for the *Note*s for constructing all *Pieces* in Coda. Each *Note* has a *duration* that defines how long this note will play and an *instrumentPart* that defines which part of the instrument makes the sound of this *Note*.

2.7. Hand Gesture Layer

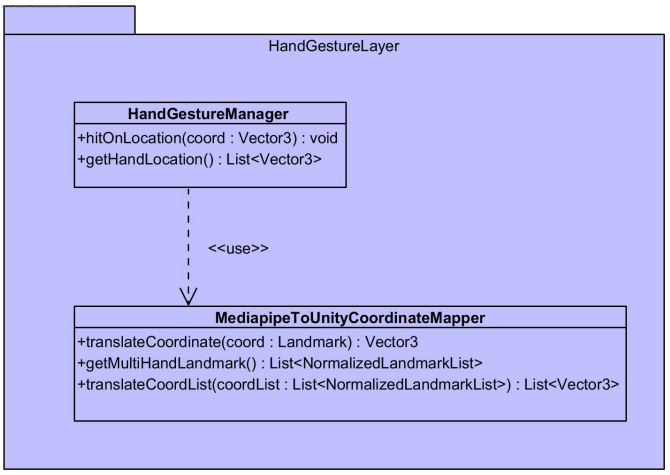

Figure 7: Subsystem Decomposition for the HandGesture Layer of *Coda*

This layer is responsible from tracking the hand gestures of the user using the MediaPipe library and detecting these gestures. Also, coordinates of the hands are received from this later and mapped to the coordinates of Unity accordingly.

HandGestureManager: A controller class for interacting with the MediaPipe library to get the location of the hands continuously for detecting hits on the instrument and stimulating the relevant layers that handle hits on Parts.

MediapipeToUnityCoordinateMapper: A mapper class for mapping the hand coordinates received from the camera to the coordinates of Unity according to the relative coordinate of the camera object in Unity. MediaPipe returns coordinates as a native structure *LandMark*, this class converts these coordinates to a *Vector3* representation to map to Unity in a correct way.

2.8. CameraControl Layer

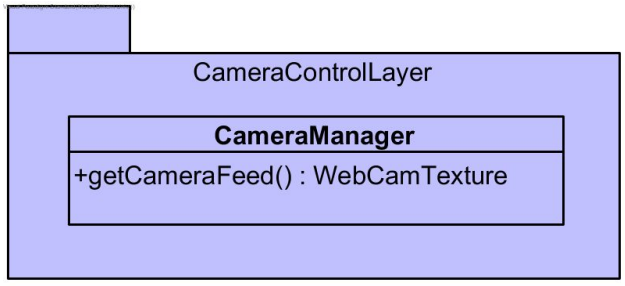

Figure 8: Subsystem Decomposition for the CameraControl Layer of *Coda*

Camera operations are handled in this layer. This layer will handle the invokation of the camera to get the feed of the camera to be analyzed.

CameraManager: A controller class for the camera operations in the application. This class invokes the camera to get the camera feed as a WebCamTexture to be analyzed by other layers which use camera data to arrange interactions (i.e. HandGestureManager).

2.9. CardboardView Layer

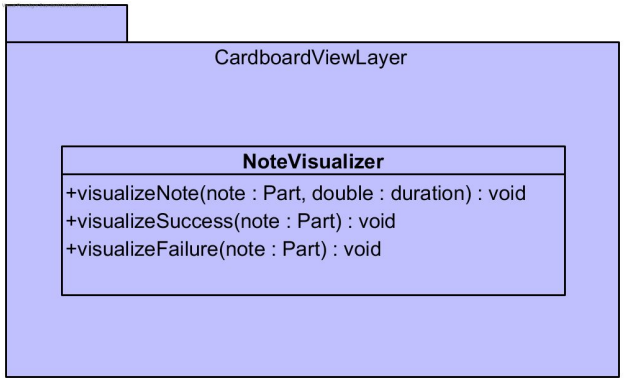

Figure 9: Subsystem Decomposition for the CardBoardView Layer of *Coda*

This layer is responsible from rendering the notes on the screen in addition to the structure views which are excluded from the diagrams for simplicity and as a result of the simplicity of objects in Unity.

NoteVisualizer: A class for rendering notes on the instruments. Notes will be visualized for two different purposes: for guiding the user to play the instrument and for indicating whether the user hit the correct note or not.

3 Class Interfaces

3.1.Play Layer

class *GameManager*			
A controller class for the *Game* model. Instantiates the *Game* and performs all other Game related operations like pausing, updating and playing a note by communicating with the relevant layers (i.e. HandGesture Layer).			
Attributes			
private	*Game*	*currentGame*	
Methods			
public	*void*	*startGame*	*type: String, instrumentType: Instrument, song: Piece, level: int,*
public	*void*	*pauseGame*	*none*
public	*void*	*playNote*	*instrument: Part*
public	*void*	*update*	*none*

Table 2: GameManager Class Description

startGame(): A method for instantiating the Game with a particular level,game and song.

pauseGame(): A method for pausing the game by opening the in-game menu.

playNote(): A method for playing a note of an instrument part specified .

update(): A method for updating the game for each change.

class *HandModel*		
A model class for representing and rendering the *hands* of the user according to the data received from the HandGesture Layer.		
Attributes		
private	*List<Vector3>*	*handCoordinates*
private	*boolean*	*isLeft*
private	*int*	*coordSize*

Table 3: HandModel Class Description

class *HandManager*			
A manager class for the *Hand* model. Instantiates the hands of the user as two *Hand* objects one being right and the other being left.			
Attributes			
private	*HandModel*	*leftHand*	
private	*HandModel*	*rightHand*	
Methods			
public	*List<Vector3>*	*getHandCoord*	*none*
public	*boolean*	*isLeftHand*	*none*
public	*boolean*	*getNoOfCoords*	*none*
public	*void*	*setHandCoord*	*coord: List<Vector3>*
public	*void*	*setLeftHand*	*left: boolean*
public	*void*	*setNoOfCoords*	*coordSize: int*

Table 4: HandManager Class Description

getHandCoord(): A method for getting the current coordinates of the hands.

isLeftHand(): A method for checking which hand it is.

getNoOfCoords(): A method for getting the number of currently available coordinates.

setHandCoord(): A method for changing the coordinates of the hands.

setLeftHand(): A method for defining the hand as the left hand.

setNoOfCoords(): A method for changing the number of currently available coordinates.

class *HandCollisionManager*			
A manager class for detecting collisions between either of the hands of the user and the instrument in the game, and then managing the interactions with the relevant classes responsible from the auditory and visual feedback.			
Methods			
public	*Part*	*onCollisionDetected*	*none*

Table 5: HandCollisionManager Class Description

getHandCoord(): A method for detecting collisions between either of the hands of the user and the instrument in the game, and then invoking the relevant classes responsible from the auditory and visual feedback

class *Game*		
A model class for the game itself. *Song* and *level* chosen by the user is stored in this class as well as the *current score* of the user.		
Attributes		
private	*int*	*currentScore*
private	*int*	*level*
private	*Piece*	*song*

Table 6: Game Class Description

class Menu*Manager*			
A controller class for the *Menu* in the application. Handles menu interactions as changing screens and clicking menu buttons according to the button clicked.			
Methods			
public	*void*	*goBack*	*none*
public	*void*	*goToScreen*	*id: String*
public	*void*	*buttonClick*	*buttonId: String, option: int*

Table 7: MenuManager Class Description

goBack(): A method for navigating back to the previous screen.

goToScreen(): A method for navigating to the specified screen.

buttonClick(): A method for indicating a specific button and invoking its functionality.

3.2. Sound Layer

class *SoundManager*			
A controller class for the sound operations in the application. This class handles the sound settings, song sounds and auditory feedback from instruments.			
Methods			
public	*boolean*	*mute*	*none*
public	*void*	*setVolume*	*val: int*
public	*void*	*playSound*	*note: Part, speed: double*
public	*void*	*playPiece*	*track: Piece*

Table 8: SoundManager Class Description

mute(): A method for muting all sounds in the application.

setVolume(): A method for setting the volume level according to the value specified.

playSound(): A method for playing the sound of the Part specified with the specified speed.

playPiece(): A method for playing a piece from the storage .

3.3. Storage Layer

class *FileManager*			
All storage operations are handled in this layer. This layer will handle the saving of recorded songs, loading saved and library songs and getting sounds of instrument pieces.			
Methods			
public	*boolean*	*saveSong*	*Path: String*
public	*Piece*	*loadSong*	*Path: String*
public	*AudioClip*	*getSound*	*instrument : Part, Path : String*

Table 9: FileManager Class Description

saveSong(): A method for writing a recording into memory.

loadSong(): A method for loading a song from the library or saved recordings .

getSound(): A method for fetching the sound of a particular instrument part from the memory.

3.4. Instrument Layer

class *Instrument*			
A model class for the general instrument structure in *Coda.Id* and *name* of the instruments are commonly contained in this structure.			
Attributes			
private	*int*	*id*	
private	*String*	*name*	
Methods			
public	*int*	*getId*	*none*
public	*void*	*setId*	*id:int*
public	*String*	*getName*	*none*
public	*void*	*setName*	*name: String*

Table 10: Instrument Class Description

getId(): A method for getting the ID of the instrument.

setId(): A method for changing the ID of the instrument.

getName(): A method for getting the name of the instrument.

setName(): A method for setting the name of the instrument.

class *Part*			
A model class for constructing instruments by parts in *Coda.* Each *Part* of an *Instrument* has a unique functionality.			
Attributes			
private	*AudioClip*	*sound*	
private	*int*	*instrumentId*	
private	*String*	*name*	
private	*String*	*partSoundPath*	
Methods			
public	*AudioClip*	*getSound*	*none*

public	*void*	*setSound*	*sound:AudioClip*
public	*int*	*getInstrumentId*	*none*
public	*void*	*setInstrumentId*	*id: int*
public	*String*	*getName*	*none*
public	*void*	*setName*	*name: String*
public	*String*	*getPartSoundPath*	*none*
public	*void*	*setPartSoundPath*	*partSoundPath: String*

Table 11: Part Class Description

getSound(): A method for getting the sound of the instrument part.

setSound(): A method for changing the sound of the instrument part.

getInstrumentID(): A method for getting the ID of the instrument that the part belongs to.

setInstrumentID(): A method for changing the ID of the instrument that the part belongs to.

getName(): A method for getting the name of the instrument part.

setName(): A method for setting the name of the instrument part.

getPartSoundPath(): A method for getting the filepath of the sound of the instrument part.

setPartSoundPath(): A method for changing the filepath of the sound of the instrument part.

3.5. Song Layer

class *Song*			
A model class for the library Songs that are stored by default in *Coda.* In addition to the *noteSequence* that depends on the *Instrument* played, this class contains sounds of the other instruments in the Song.			
Attributes			
private	*AudioClip*	*otherSound*	
Methods			
public	*AudioClip*	*getOtherSound*	*none*
public	*void*	*setOtherSound*	*otherSound: AudioClip*

Table 12: Song Class Description

getOtherSound(): A method for getting the other sounds in the Song.

setOtherSound(): A method for changing the other sounds in the Song.

class *Piece*			
A model class for the general Piece structure in *Coda.* Attributes that are common to all pieces in Coda, either recorded by the user or in the library by default, are contained in this class.			
Attributes			
private	*String*	*filepath*	
private	*List<Node>*	*noteSequence*	
public	*String*	*pieceName*	
Methods			
public	*String*	*getFilePath*	*none*
public	*void*	*setFilePath*	*filepath: String*
public	*List<Node>*	*getNoteSequence*	*none*
public	*void*	*setNoteSequence*	*nodeSequence: List<Node>*

Table 13: Piece Class Description

getFilePath(): A method for getting the filepath of the Piece.

setFilePath(): A method for changing the filepath of the Piece.

getNoteSequence(): A method for getting the note sequence of the Piece.

setNoteSequence(): A method for changing the note sequence of the Piece.

class *Recording*
A model class for the *Recordings* made by the user. *Recordings* are stored as a sequence of *Notes* played by the user during the game.

Table 14: Recording Class Description

class Note			
A model class for the *Note*s for constructing all *Pieces* in Coda. Each *Note* has a *duration* that defines how long this note will play and an *instrumentPart* that defines which part of the instrument makes the sound of this *Note*.			
Attributes			
private	*double*	*duration*	
private	*Part*	*instrumentPart*	
Methods			
public	*double*	*getDuration*	*none*
public	*void*	*setDuration*	*duration: double*
public	*Part*	*getPart*	*none*
public	*void*	*setPart*	*part: Part*

Table 15: Note Class Description

getDuration(): A method for getting the duration of the Note.
setDuration(): A method for changing the duration of the Note.
getPart(): A method for getting the part required of the Note.
setPart(): A method for changing the part required of the Note.

3.6. HandGesture Layer

class *HandGestureManager*			
A controller class for interacting with the MediaPipe library to get the location of the hands continuously for detecting hits on the instrument and stimulating the relevant layers that handle hits on Parts.			
Methods			
public	*void*	*hitOnLocation*	*coord: Vector3*
public	*List<Vector3>*	*getHandLocation*	*none*

Table 16: HandGestureManager Class Description

hitOnLocation(): A method for detecting the hit on a location and invoking the relevant layers to handle the hit accordingly.

getHandLocation(): A method for getting the coordinates of the hand currently from MediaPipe.

class *MediapipeToUnityCoordinateManager*			
A mapper class for mapping the hand coordinates received from the camera to the coordinates of Unity according to the relative coordinate of the camera object in Unity.			
Methods			
public	*Vector3*	*translateCoordinate*	*coord:Landmark*
public	*List<NormalizedLandmarkList>*	*getMultiHandLandMark*	*none*
public	*List<Vector3>*	*translateCoordList*	*coordList: List<NormalizedLandmarkList>*

Table 17: MediapipeToUnityCoordinateManager Class Description

translateCoordinateList(): A method for translating a List of Normalized LandMarks relative to the camera into 3-dimensional coordinates relative to the camera object in Unity.
translateCoordinate(): A helper method for translating a single of Normalized LandMark relative to the camera into 3-dimensional coordinates relative to the camera object in Unity.
getMultiHandMark(): A method for getting the coordinates of the hand currently from MediaPipe as a Normalized HandMark List.

3.7. CameraControl Layer

class *CameraManager*			
A controller class for the camera operations in the application. This class invokes the camera to get the camera feed as a WebCamTexture to be analyzed by other layers which use camera data to arrange interactions (i.e. HandGestureManager).			
Methods			
public	*WebCamTexture*	*getCameraFeed*	*none*

Table 18: CameraManager Class Description

getCameraFeed(): A method for invoking the camera to get the camera feed as a WebCamTexture.

3.8. CardboardView Layer

class *NoteVisualizer*			
A class for rendering notes on the instruments. Notes will be visualized for two different purposes: for guiding the user to play the instrument and for indicating whether the user hit the correct note or not.			
Methods			
public	*void*	*visualizeNote*	*note: Part, double: duration*
public	*void*	*visualizeSuccess*	*note: Part*
public	*void*	*visualizeFailure*	*note: Part*

Table 19: NoteVisualizer Class Description

visualizeNote(): A method for visualizing a note to guide the user on a specific instrument part and for a specific duration.
visualizeSuccess(): A method for visualizing a successful interaction by the user on the relevant part.
visualizeFailure(): A method for visualizing a failed interaction by the user on the relevant part.

4 Glossary

UI: User Interface
UML: Unified Modelling Language
FOV: Field of View
6DOF: Six Degrees of Freedom
Client: Part of the system that the user interacts with.
Server: Part of the system that handles the access to a centralized resource or service.
VR : Virtual Reality. A computer generated world that is completely virtual in terms of environment and the objects in it which provides a virtual interactive environment to the user.
Computer Vision: Combining/Using advanced Image Processing, Machine Learning and Deep Learning techniques to enable computers to see as a human does.

5 References

[1] Smith. B, “*New study demonstrates link between music and statistical learning*,” The Sydney Morning Herald, 2019. [Online]. Available: https://www.smh.com.au/technology/new-study-demonstrates-link-between-music-and-statistical-learning-20170514-gw4eec.html. [Accessed: Oct. 10, 2019].

[2] ABRSM, *ABRSM:*. [Online]. Available: https://es.abrsm.org/en/making-music/4-the-statistics/. [Accessed: Oct. 10, 2019].

[3] IBM, "UML - Basics," June 2003. [Online]. Available: http://www.ibm.com/developerworks/rational/library/769.html.

[4] IEEE, "IEEE Citation Reference," September 2009. [Online]. Available: https://m.ieee.org/documents/ieeecitationref.pdf. [Accessed 9-Feb-2018].