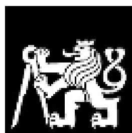


FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF MASTER'S THESIS

Title:	Speech development application
Student:	Bc. Vojtěch Pajer
Supervisor:	Ing. David Šenkýř
Study Programme:	Informatics
Study Branch:	Web and Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of summer semester 2019/20

Instructions

The goal of this thesis is to design, implement, and test a new application to help develop speech of hearing-impaired children using voice-controlled animations and simple games.

The application will be created in a collaboration with a special pedagogical center at SŠ, ZŠ a MŠ pro sluchově postižené Holečkova.

1. Analyze Speech Viewer 3 application used in the mentioned center.
2. Due to the incompatibility of Speech Viewer 3 application and current operating systems, design a new application for a platform based on your choice.
3. Analyze the existing voice-processing libraries for the selected platform and the selected technology.
4. Implement the application with at least 3 simple games recommended by the special pedagogical center.
5. Test the application in a co-operation with the users of the special pedagogical center.
6. Summarize and evaluate the results reached.

References

Will be provided by the supervisor.

Ing. Michal Valenta, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague October 24, 2018

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Speech development application

Bc. Vojtěch Pajer

Department of Software Engineering
Supervisor: Ing. David Šenkýř

May 9, 2019

Acknowledgements

Firstly, I want to thank my supervisor Ing. David Šenkýř for all of his valuable advice and the immediate help he provided me whenever I needed it.

Secondly, I'd like to thank Bc. Denisa Šleisová, Lucie Koháková and PhDr. Jarmila Roučková for their cooperation during my work on this project. Also, I am very grateful to my friends and colleagues for their words of support and motivation.

Finally, I want to thank my parents and my family for the amazing support they've provided me throughout my study years. Thank you.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on May 9, 2019 . . . .

Czech Technical University in Prague
Faculty of Information Technology
© 2019 Vojtěch Pajer. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Pajer, Vojtěch. *Speech development application.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2019.

Abstrakt

Tato diplomová práce popisuje návrh a implementaci mobilní aplikace pro operační systém iOS. Úkolem aplikace je podpořit rozvoj řeči u dětí se sluchovým postižením za pomoci čtyř animovaných her. Při vývoji aplikace byla použita architektura VIPER, framework na tvorbu her SpriteKit a framework na zpracování zvuku AudioKit.

Klíčová slova rozvoj řeči, dětské hry, Swift, iOS, SpriteKit, VIPER, AudioKit, reaktivní programování, RxSwift.

Abstract

This diploma thesis describes the design and implementation of a mobile application for operating system iOS. The goal of the application is to improve the ability of hearing-impaired children to control their voice. The application consists of four animated games. VIPER architecture, game framework SpriteKit and a sound processing framework AudioKit were used during the implementation of this application.

Keywords speech development, children's games, Swift, iOS, SpriteKit, VIPER, AudioKit, reactive programming, RxSwift.

Contents

Introduction **1**
Structure . . . 2

1 State-of-the-art **3**
1.1 Speech Viewer . . . 3
1.2 Krtek . . . 4
1.3 Mentio Hlas . . . 4

2 Analysis and Design **7**
2.1 Requirements Definition . . . 8
2.2 Use Cases . . . 9
2.3 Architecture . . . 10

3 Our Approach **17**
3.1 Cloud Game . . . 17
3.2 Airplane Game . . . 19
3.3 Dog Game . . . 21
3.4 Balloon Game . . . 22

4 Implementation **25**
4.1 Technologies . . . 25
4.2 Sound Processing . . . 32
4.3 Game Implementation . . . 34
4.4 Event Logging . . . 37
4.5 Other Libraries . . . 39
4.6 Overview . . . 40

5 Deployment and Distribution **43**
5.1 Deployment of iOS Applications . . . 43
5.2 Distribution . . . 44

6 Testing **47**
6.1 Testing Results 47
6.2 Testing Conclusion 49

Conclusion **51**
Future work 52

Bibliography **53**

A Acronyms **57**

B Contents of enclosed DVD **59**

C Profiles of Tested Children **61**

List of Figures

1.1 Examples of screens of *Speech Viewer* application 4
1.2 Layout of game *Krtek* 5
1.3 Examples of screens of *Mentio Hlas* application 5

2.1 Usecase diagram 11
2.2 *Model-View-Controller* architecture 12
2.3 *Model-View-ViewModel* architecture 13
2.4 *VIPER* architecture 15

3.1 Images of *Cloud Game* 18
3.2 Timeline of events happening in *Cloud Game* 19
3.3 Images of *Airplane Game* 19
3.4 Timeline of events happening in *Airplane Game* 20
3.5 Images of *Dog Game* 21
3.6 Timeline of events happening in *Dog Game* 22
3.7 Images of *Balloon Game* 23
3.8 Timeline of events happening in *Balloon Game* 24

4.1 Viper architecture enhanced by reactive programming 29
4.2 Application without dependency injection 30
4.3 Application with dependency injection 32
4.4 Layout of *SpriteKit Scene* editor 35
4.5 Sequence of images simulating dog jumping movement 37
4.6 Composition of classes in *Hlásek* 41

List of Tables

4.1 List of general events and games that evoke them 38
4.2 List of game events 39

Introduction

Communication for people with hearing impairment can be challenging. Learning how to speak and pronounce words correctly without the voice feedback is, without any external help, close to impossible. According to the statistics of the *Czech Union of the Deaf* from 2017 [1], only 7,300 people in Czech republic could use the sign language. For children, who want to play and communicate with their peers, and are unable to express themselves, is the situation even more alarming.

Kateřina Hádková classifies the hearing impairments in [2] based on the period of life when it happened. The hearing impairments of children can happen in the prenatal[1], perinatal[2] and postnatal[3] period of their life. The most common causes of hearing impairment from the prenatal and perinatal periods are infectious diseases, metabolic disorders, use of ototoxic drugs, meningitis, complicated childbirth and cerebral hemorrhage. During the postnatal period, the causes are, for instance, meningitis, parotitis, rubeola, and injuries.

Mobile devices have become an inseparable part of the majority of Czech households. According to the *Czech Statistical Office*, in 2015, 98% of Czech families owned a mobile phone, and every fifth family owned a tablet device. For families with young children, the number of tablet devices doubled [3]. As of the beginning of 2019, there were 2,100,000 applications on the *Google Play Store*[4], and 1,800,000 applications on the *AppStore*[5] [4]. Neither of these applications was designed to improve the hearing-impaired children's ability to speak.

[1] A period before the birth of the child (during the pregnancy).
[2] A period immediately before and after birth.
[3] A period shortly after the birth of a child.
[4] An online shop with applications for devices running operating system Android.
[5] An online shop with applications for devices running operating systems iOS and macOS.

This diploma thesis focuses on the development of such an application. The application will consist of animated games that will keep children engaged and motivate them to improve. Each of these games will be designed to focus on a different aspect of a children's voice control.

The work on this thesis will be done in cooperation with a master student from *Charles University*, who will help with a better understanding of hearing impairment problems. Also, the *Special Pedagogical Center at the High School, Elementary School and Kindergarten for Hearing Impaired Children in Holečkova* (*SPC*) will help with the testing on children that are visiting the center.

Structure

The work on this diploma thesis is separated into six segments, each of which is described in a separate chapter. The list of chapters follows.

Chapter 1 analyses applications that focus on the improvement of children's ability to speak.

Chapter 2 shows the requirements and use cases of the application and discusses the architecture that was used during the implementation.

Chapter 3 presents the design of each game in the application.

The technologies that were used during the implementation and the way the core parts of the application were implemented are described in **Chapter 4**.

Chapter 5 discusses the deployment and distribution of the application to designated testing devices.

Chapter 6 presents the results of the testing and the changes that resulted from it.

The final chapter summarises the results of this project and presents possible future improvements.

Chapter 1

State-of-the-art

There are limited applications addressing speaking problems of children with impaired hearing or speaking practice in general. The most notable ones are mentioned in this chapter.

1.1 Speech Viewer

Speech Viewer is an application developed by *IBM* in the end of the last century. It has changed the before-existing boring speech exercises into interactive games that keep children motivated and entertained. *Speech Viewer* offers numerous speech exercises, such as voice timing, pitch control, two-phoneme contrast[6] and spectra pattering[7] [5]. Each exercise offers various game themes. For example, the voice modulation exercises can be played with a car that drives uphill, a chicken that jumps up to the henhouse, a warrior ascending the stairs to the treasure, or a bucket of bricks that need to be pulled up over a pulley.

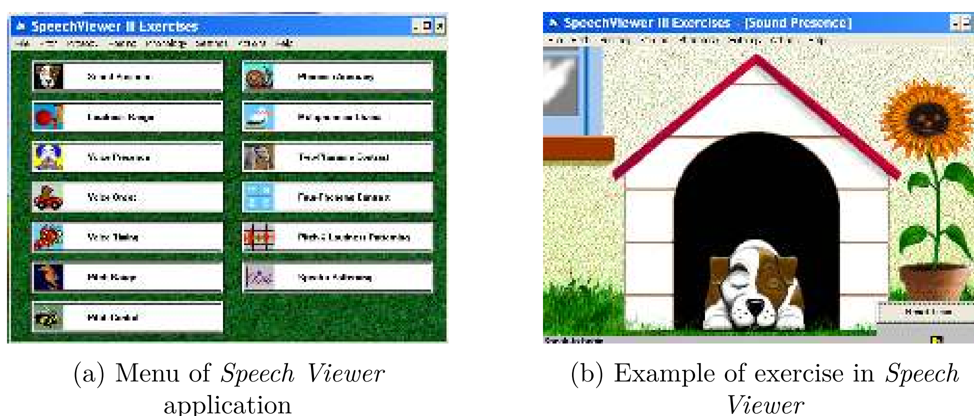
However, it poses a number of serious problems:

- It is no longer updated. Therefore, it cannot run on operating systems higher than Windows XP [6].
- It requires an external microphone.
- Some exercises need an extensive voice setup in order to calibrate the application.
- It reacts to non-vocal sounds, such as hitting the table or clapping.

Despite the problems mentioned above, *Speech*View*er* was, according to the *SPC*, a breakthrough application when it was published. Therefore, it is a main source of inspiration for the application developed in this master thesis.

[6]Exercise that improves accuracy in contrasting phonemes.

[7]Exercise that uses spectral analysis to improve accuracy of phoneme production.



(a) Menu of *Speech Viewer* application

(b) Example of exercise in *Speech Viewer*

Figure 1.1: Examples of screens of *Speech Viewer* application

1.2 Krtek

Krtek is a simple chasing game developed by *Josef Kufner* as a semestral project on the *Faculty of Electrical Engineering* at *Czech Technical University* in the semester 2005/2006 [7]. In the game, the players control a gardener and a mole. The gardener plants plants on his garden. He receives points for each planted plant and bonus points when he catches the mole. The mole destroys the plants, for which he receives points. The game loop is endless. Players control their characters by a pair of tones for the left and right movement. The mole is controlled by the lower frequencies and the gardener by the higher frequencies. In the bottom of the screen, there is an indicator that shows which frequencies are the players aiming for [8].

As shown in Figure 1.2, *Krtek* has a plain user interface written in C programming language library *ncurses* [9]. Even though it is an application controlled by a voice, it cannot be used for the development of speech of children with hearing impairment. The range of sounds required to control the application is limited. Also, it requires two players to participate in the game. Moreover, it only works on desktop computers, therefore its usage requires an external microphone.

1.3 Mentio Hlas

Mentio Hlas is a software for training of a phonation and a voice modulation. It provides a visual feedback on voice creation and develops the ability to change the intensity and the frequency of the voice. The exercises are focused on, for instance, the indication of a presence of the voice, the length of the expiratory flow, the timing of speech, and switching between the low and the high frequencies [10]. Each exercise offers various game themes.

The user interface, the composition of game themes and the functionality

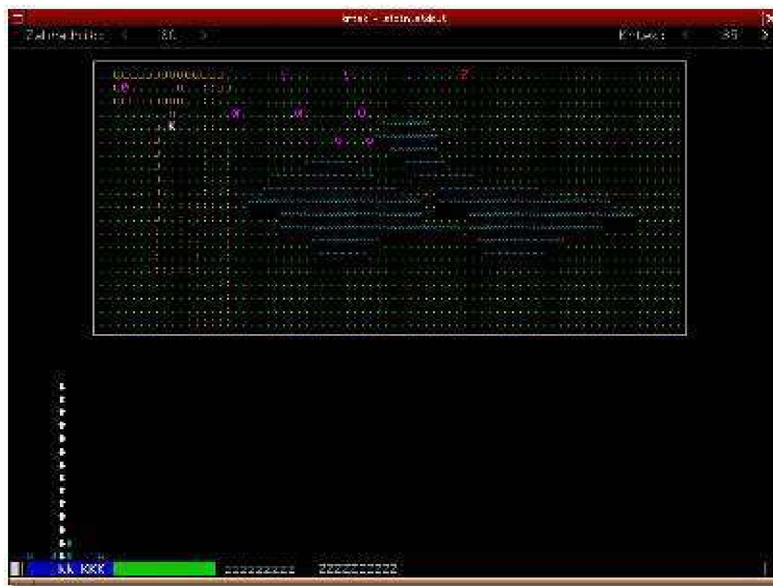
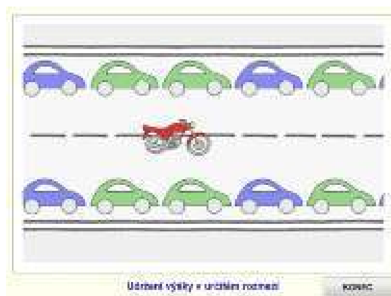


Figure 1.2: Layout of game *Krtek*

of *Mentio Hlas* (shown in Figure 1.3) are very similar to the ones of the *Speech Viewer* application. Its target audience are, among others, children with a hearing impairment. Despite its advantages, *Mentio Hlas* has not been found suitable for this master thesis. Its main problems are the fact that it is only available on desktop computers and its cost. The *Mentio Hlas* license is expensive [11]. Also, in order to use the application, users need to buy an external microphone.



(a) Exercise for holding of voice height in certain range

(b) Gradual voice height increase exercise

Figure 1.3: Examples of screens of *Mentio Hlas* application

Chapter 2

Analysis and Design

Starting from now on, the chapters are devoted to the design and implementation of an application called *Hlásek*. *Hlásek* is a mobile game application whose target audience is children with hearing impairment. Children's games are controlled by a sound coming from the microphone of the device and are meant to improve children's ability to control their voice.

Hlásek has been created as a solution that tries to solve the problems of the applications from the previous chapter, and presents a new, entertaining and intuitive prototype. The target age group of the application is mostly preadolescent children. Therefore, a great deal of effort had been put in designing the application in a way that holds children's attention for an extended time. Another challenge was adjusting the difficulty level of each game so that it was neither easy nor too difficult. From the nature of the application (games, children's design, demand on being attention-holding), it was decided that the application would be created for tablet devices.

As stated in the assignment of this diploma thesis, the application was meant to be developed in cooperation with the *SPC*. They provided the grounds for the application testing. After establishing the initial contact with the *SPC*, it was discovered that they only had one device available – iPad Air 2. Hence, it was decided that the *Hlásek* application would be developed for the operating system iOS.

Along with the *SPC*, a master student from *Charles University*, Bc. Denisa Šleisová, was significantly contributing to the designing process of the *Hlásek* application. Denisa studies *Special Pedagogics* at the *Pedagogical Faculty*. Currently writing her master thesis on *The use of modern technological aids for the development of spoken speech at children with severe hearing impairment*, she plans using the *Hlásek* application as the main testing tool for her observations. For that reason, she cooperated with the author of this thesis and provided valuable insights on the problems of children with hearing impairment.

2.1 Requirements Definition

An important step before the development of a new software is defining all requirements on the software. The requirements were divided into *functional* and *non-functional requirements*.

2.1.1 Functional Requirements

Functional requirements describe the behaviour of the application and its interaction with its environment. The list of functional requirements follows.

- **FR1: Account Creation**
  The application should allow users to create a new account.
- **FR2: Login**
  The application should require the user to be logged in.
- **FR3: Account Deletion**
  The application should allow the user to delete his/her account.
- **FR4: Game Selection**
  The application should allow the user to select a game.
- **FR5: Sound Processing**
  The application should process the sound from the microphone of the device.
- **FR6: Game Movement**
  The application should move the game entities according to the processed sound from the microphone.
- **FR7: Game Finish**
  The application should be able to finish the game when the user accomplishes the desired tasks.
- **FR8: Game Exit**
  The application should allow the user to exit the game before it finishes.

2.1.2 Non-Functional Requirements

Non-functional requirements complement the functional requirements. They describe the additional required features of the application. The list of non-functional requirements follows.

- **NFR1: Mobile Application**
  The application should be a native mobile application running on the operating system iOS.

- **NFR2: Programming Language**
  The application should be written in the programming language Swift.
- **NFR3: System Version**
  The application requires the operating system version 12 or higher.
- **NFR4: Orientation**
  The application should support a landscape orientation and forbid a portrait orientation.
- **NFR5: Internet Connection**
  The application should utilise an internet connection.
- **NFR6: Database**
  The application should use an internal database for saving of the user account information.
- **NFR7: Language**
  The application should be in the Czech language.
- **NFR8: Intuitiveness**
  The application should be easy to understand and use.
- **NFR9: Updating**
  The application installed on testing devices should be easily updatable.
- **NFR9: Deployment**
  The application should be deployable to desired devices.
- **NFR10: Event Logging**
  The application should log important voice events.
- **NFR11: Event Uploading**
  The application should upload logged events to online storage for further analysis.
- **NFR12: Crash Reporting**
  The application should report its crashes.

2.2 Use Cases

Use cases define the typical interaction of a user role and the system as a list of steps. The diagram in 2.1 shows an overview of all possible use cases. Even though children are likely to need assistance when using the *Hlásek* application, separating *Supervisor* and *Child* into two different roles seemed pointless. The use case diagram only distinguishes between two roles – the *Unsigned User* and the *Signed User*.

- **UC1: Account Creation**
  - The unsigned user selects the *Create Account* option on the landing page.
  - The unsigned user fills in the required information and confirms the creation.
- **UC2: Account Deletion**
  - The unsigned user selects the *Login* option on the landing page.
  - The unsigned user chooses the account that should be deleted from the account list.
  - The unsigned user taps the cross button on the selected account and confirms the deletion.
- **UC3: Gameplay**
  - The unsigned user selects the *Login* option on the landing page.
  - The unsigned user selects the account he wants to sign with.
  - The signed user selects the game he/she wants to play.
  - The signed user finishes or exits the game.

2.3 Architecture

The selection of a good architecture of the application is a crucial step that should be made before the very beginning of the implementation. According to the article at Medium [12] and a lecture by *Krzysztof Zabłocki* [13], there are several architectural patterns in the mobile application development society. The most popular ones are *Model-View-Controller*, *Model-View-ViewModel*, and *VIPER*.

2.3.1 Model-View-Controller

The *Model-View-Controller* pattern (*MVC*) for the iOS applications is slightly different from the traditional *MVC* pattern known in, for example, web development. Having all three entities tightly coupled dramatically reduces reusability for each of them.

The composition of entities in *MVC* that is used in the iOS applications is captured in Figure 2.2a. The *Controller* mediates between the *View* and the *Model*. It separates the *Model* and the *View*. As a result, the *Controller* becomes the least reusable part, which is not a problem.

In praxis, however, this composition is difficult to maintain. The *Controller* is tightly bound to the *View* life cycle and separating them is hard.

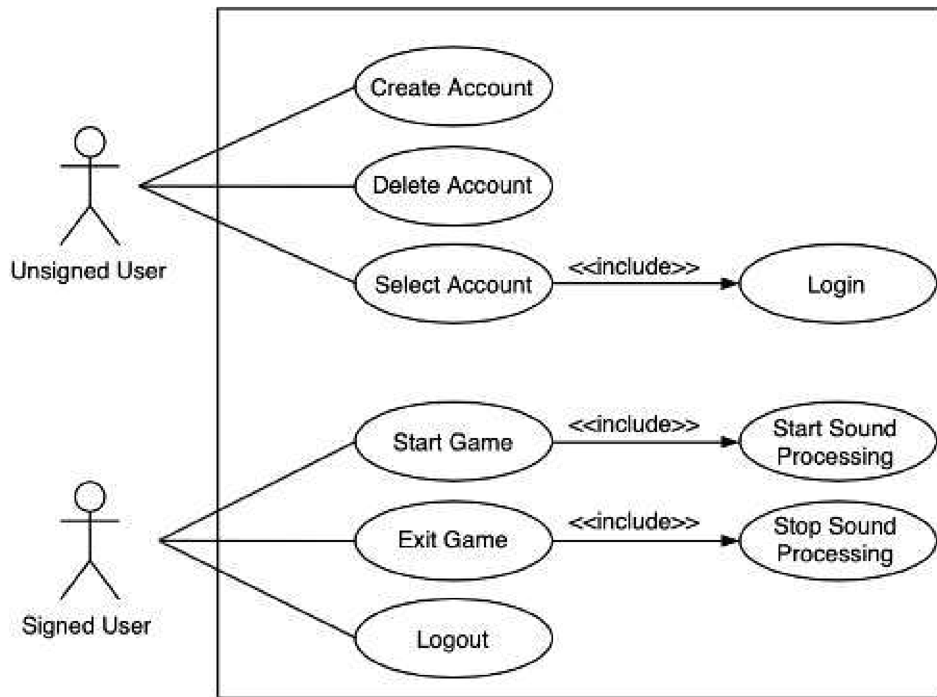


Figure 2.1: Usecase diagram

It results in a so-called *Massive View Controllers* (as shown in Figure 2.2b), *View-Controller* entities that are responsible for the majority of tasks in the application and are almost non-reusable. Therefore, the only reusable and testable part is *Model*.

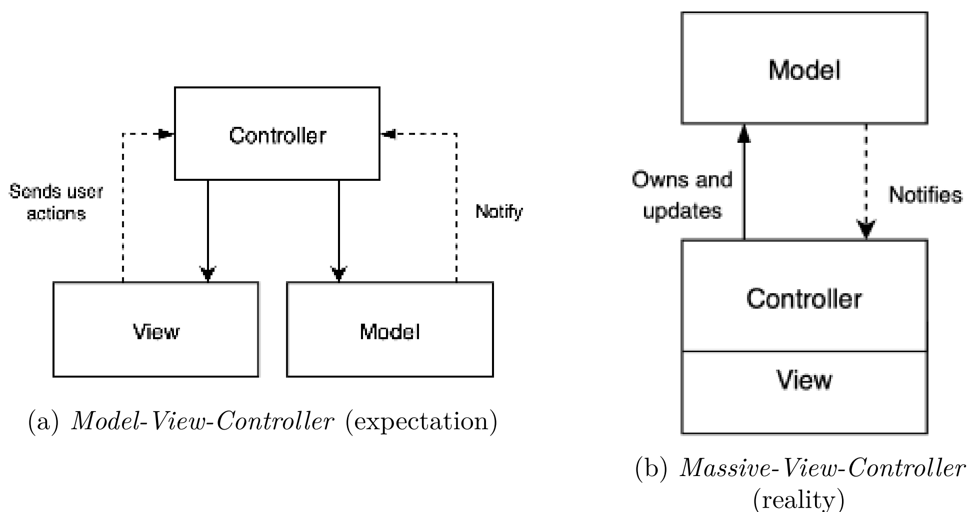
Despite all that was mentioned above, *MVC* is still a popular pattern. From all of the patterns mentioned in this section, it is the easiest to learn, maintain, and requires the least amount of code.

2.3.2 Model-View-ViewModel

The *Model-View-ViewModel* pattern (*MVVM*) takes into account the problems of the *MVC* pattern. It treats the *View-Controller* entity from *MVC* as the *View* and introduces a new entity – *ViewModel*. *ViewModel* is a *UIKit*[8] independent representation of the *View* and its state. It contains all of the business logic, which makes the application excellently testable.

The *View* transfers most of its logic to the *ViewModel*, and it binds to it. The binding is, in most cases, done by *reactive programming*. In short, the *View* observes events on the *ViewModel* and changes its elements accordingly.

[8]Apple library containing all UI elements.



(a) *Model-View-Controller* (expectation)

(b) *Massive-View-Controller* (reality)

Figure 2.2: *Model-View-Controller* architecture

The *reactive programming* is discussed in detail in Section 4.1.1 of the *Implementation* chapter. The structure of *MVVM* pattern can be seen in Figure 2.3.

The pattern falls short in two areas – screen routing and the separation of concerns for more complex modules. In the latter, the *ViewModel* grows enormously and takes responsibility for numerous unrelated tasks. The *VIPER* pattern addresses these problems.

In comparison to *MVC*, the *MVVM* pattern is more challenging to learn and understand. Also, the amount of the code is doubled. Nonetheless, the excellent testability and easier maintenance and extensibility make the pattern a better choice for every more complicated application.

2.3.3 VIPER

The *VIPER* pattern is the last pattern described in this section. Also, it is the pattern that was used in the implementation of *Hlásek*. Therefore, it will be discussed in more detail. *VIPER* focuses on separation of concerns even more than the previous patterns and introduces five layers. These layers construct the abbreviation *VIPER – View-Interactor-Presenter-Entity-Router*. It is also the only pattern of the ones mentioned above that address screen routing. In various implementations (including this one), it is extended by a *Builder*.

- The *Router* is the component responsible for the routing of the application. It controls the flow of the screens and handles events before each screen is displayed.

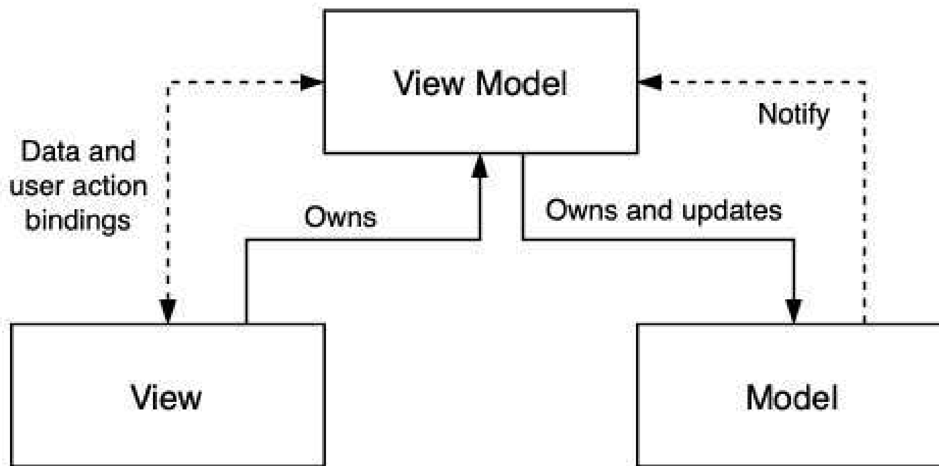


Figure 2.3: *Model-View-ViewModel* architecture

- *Entities* are the plain data objects. The data access responsibility is handled by the *Interactor*.
- The *Interactor* is responsible for the business logic related to the data (represented by the *Entities*), networking, database operations, and others. Also, it uses external services and managers that are not considered as a part of the *VIPER* module. *Interactor* receives non-UI related tasks from the *Presenter*.
- The *Presenter* is the component responsible for the UI related (but *UIKit* independent) business logic. It receives events from the *View* and either process them or passes them forward to the *Interactor*. It also holds an instance of the *Router* and invokes routing methods on it.
- The *View* is similar to the one in the *MVVM* pattern. It displays the user interface a sends user interactions to the *Presenter*. The only difference is that it does not use binding.
- *Builder* is an optional component of the *VIPER* architecture. It creates instances of all screens, connects them to create a working *VIPER* module and optionally injects dependencies.

2.3.3.1 Routing

The screen size of mobile devices allows less amount of content on each screen than on the computer and web applications. Therefore, mobile applications require more screens to display the same information. Moreover, the layout on iPhones and iPads often differs. For instance, selecting a product from the

product list on an iPhone opens a new screen with the product detail, whereas iPad, which provides more screen area, displays the product list on the left side of the screen and the product detail on the right side. Handling this logic in the *View* (from where the routing is handled in the previously mentioned patterns) results in more complex, less reusable *View*s. Even though routing between screens is very frequent and often handles some application logic, *VIPER* architectural pattern is one of the few patterns that address routing as a separate concern.

Navigation between screens in iOS applications is handled by the *Navigation Controller*. *Navigation Controller* has a navigation stack to which it pushes new screens and from which it pops them. *VIPER* approaches routing as follows:

- Each screen has its *Router* class. This class is responsible for pushing of the screen to the navigation stack.
- The *Router* is started with the *Finish* and the *Cancel* callbacks. These callbacks define where the application should navigate when the work of the screen is finished or canceled. They also abstract the flow of the screens from the screen itself.
- *Router* for more complex screens, from where there are many navigation paths, such as the application settings or the dashboard, also implements methods for navigation along these paths.
- Above all screen routers, there is the *Application Router* which handles the flow of the entire application.

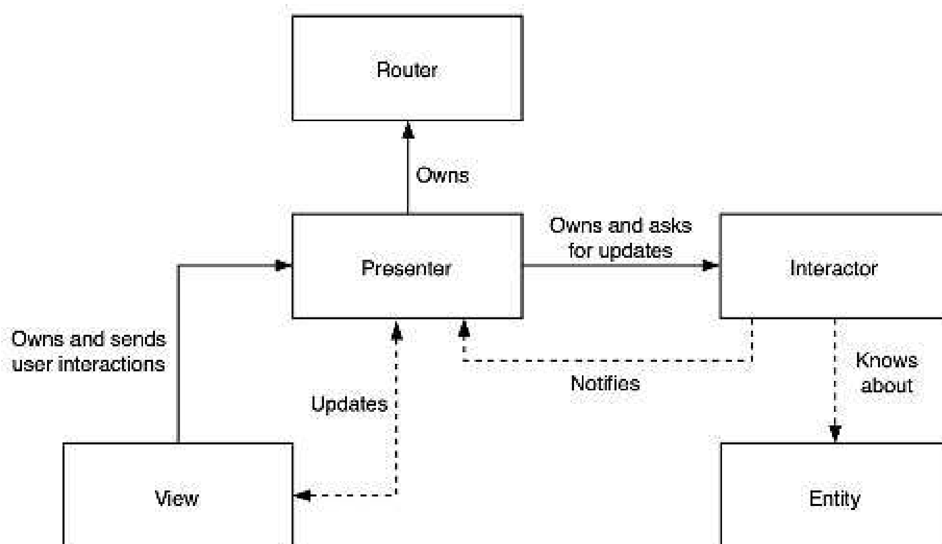


Figure 2.4: *VIPER* architecture

Chapter 3

Our Approach

This chapter elaborates on the games that compose *Hlásek*, and the purpose that each of these games serves. The application introduces four games – *Cloud Game*, *Airplane Game*, *Dog Game* and *Balloon Game*. Each game presents a different vocal exercise. Composition of these exercises should improve the children's ability to control their voice.

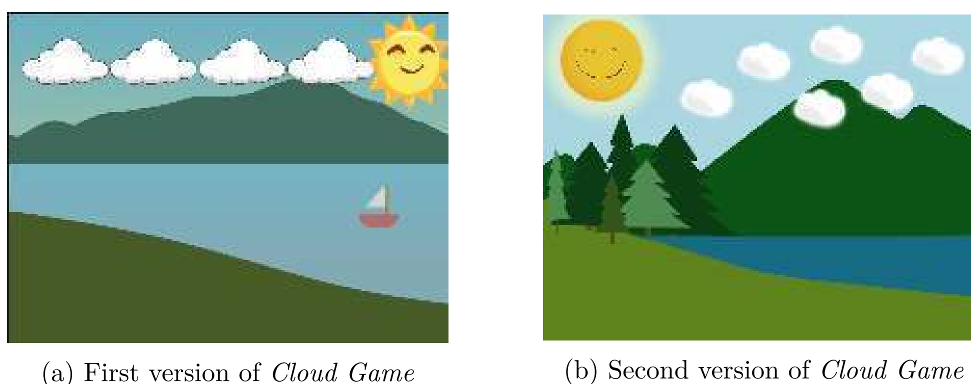
One of the main challenges was to make the games not only useful but also entertaining for the children, so that they enjoy coming back to playing it again. It was achieved by a friendly animated design of each game and a special visual reward when the game was finished.

The final design of each game displayed in their corresponding sections was created by *Lucie Koháková*, a second year student of *Industrial Design* on the *Faculty of Architecture* at *Czech Technical University in Prague*.

3.1 Cloud Game

The *Cloud Game* is the first game of the application. The main purpose of this game is to introduce the application to the children by showing them that they will control it with their voice.

The game presents an animated nature with a smiling sun and clouds on the sky. The layout of two currently used versions is shown in Figure 3.1. After a brief delay, the clouds start falling to the ground, where they stop. The sun stops smiling and an arrow indicating that the clouds should be raised back to the sky appears. At this point, user interaction begins. Whenever the child makes a sound, one of the clouds starts raising. When all clouds are back, the sun starts smiling again and the game ends.



(a) First version of *Cloud Game* (b) Second version of *Cloud Game*

Figure 3.1: Images of *Cloud Game*

3.1.1 Timeline of events

The example timeline that indicates events in *Cloud Game* is displayed in Figure 3.2:

- *t1*: The game is started. The sound is not processed yet because the initial animation has to be played first.
- *t2*: The initial animation finishes and the sound processing is initialised.
- *t3*: The processed sound amplitude was in a voice range; however it did not remain there for the required amount of time and was, therefore, classified as an invalid sound. This approach filters out a small sample of short vocal sounds. More importantly, it removes a majority of instant non-vocal sounds, such as clapping and hitting the table.
- *t4*: The processed sound amplitude was in a voice range for the required amount of time. The first cloud starts raising back to the sky.
- *t5*: The processed sound amplitude was in a voice range for the required amount of time. The second cloud starts raising back to the sky.
- *t6*: Even though the sound amplitude was in a voice range for an extended period of time, no more clouds started raising after the second one. This way the game requires children to go back to being silent before making another sound again.
- *t7*: The sound was too loud. Nothing happens.
- *t8*: Once all clouds are back on the sky (the process of rising each cloud is not shown in the timeline), the game is finished.

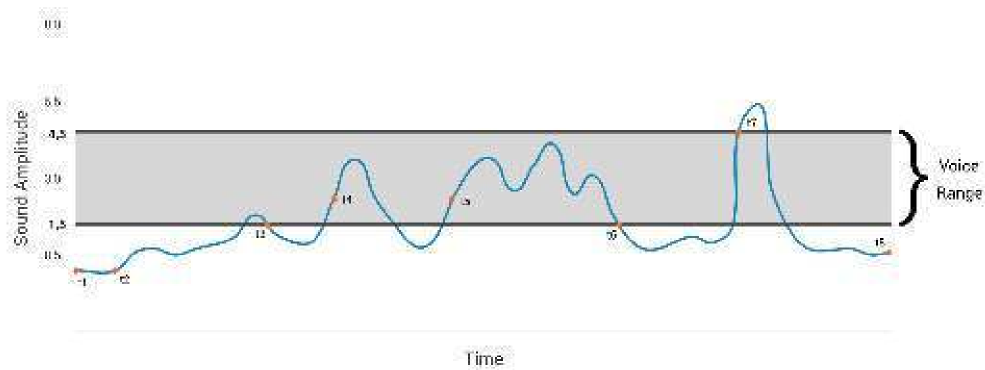


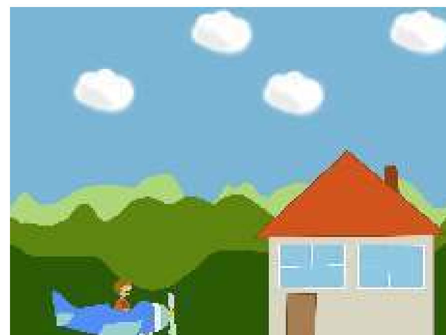
Figure 3.2: Timeline of events happening in *Cloud Game*

3.2 Airplane Game

The second game is already more specialised than the first one. It exercises lengthening of the expiratory flow and the ability to hold a constant vocal phonation.

The primary object in this game is an airplane flying through the landscape to a house. At the beginning of the game, the airplane is hidden behind the left side of the screen. An arrow indicates that something is hidden there. Once the child makes the first sound, the airplane appears and starts flying towards the house. The child then needs to speed the airplane up by long vocal sounds. When he or she becomes silent or makes short vocal sounds, the airplane starts slowing down, gradually descending to the ground. When it hits the ground, it stops entirely. The goal of the game is to fly with the airplane next to the house. Once it is there, it lands, and the game is finished. The graphical design of the game can be seen in Figure 3.3

(a) Airplane flying due to children's long expiratory flow



(b) Airplane landed next to house

Figure 3.3: Images of *Airplane Game*

3.2.1 Timeline of events

The example timeline that indicates events in *Airplane Game* is displayed in Figure 3.4:

- *t1*: The game is started and sound processing is initialised. Airplane is not visible on the screen yet. It requires an initial sound to appear from the left side of the screen.
- *t2*: The processed sound amplitude was not in a voice range for the required amount of time.
- *t3*: The processed sound amplitude was in a voice range for the required amount of time. The airplane accelerates and appears from the left side of the screen.
- *t4*: The initial animation finishes.
- *t5, t6, t7*: The airplane accelerates.
- *t8*: The airplane is at its maximum speed. It cannot accelerate more.
- *t9*: The sound amplitude dropped below the voice range. The airplane starts descending.
- *t10*: The airplane hits the ground and stops.
- *t11*: The airplane reaches its final destination and lands.

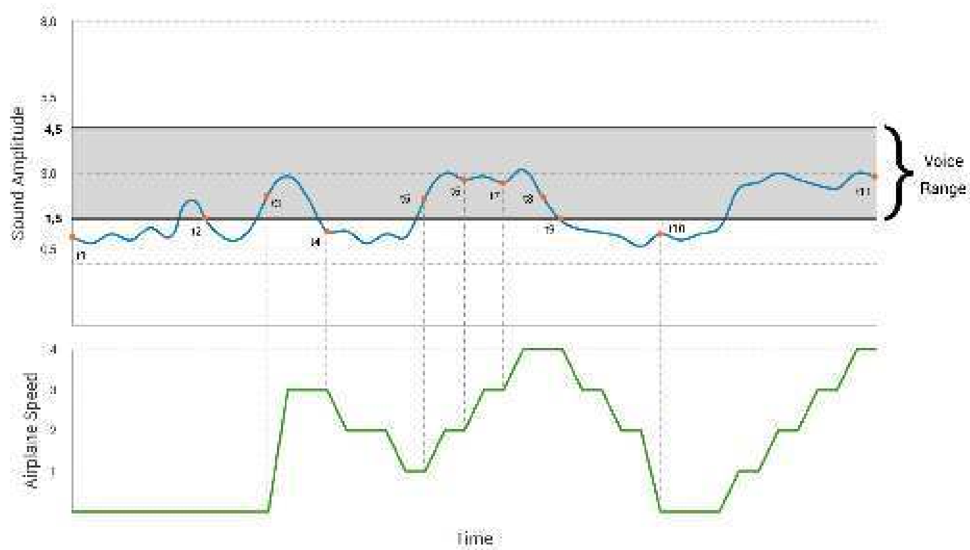


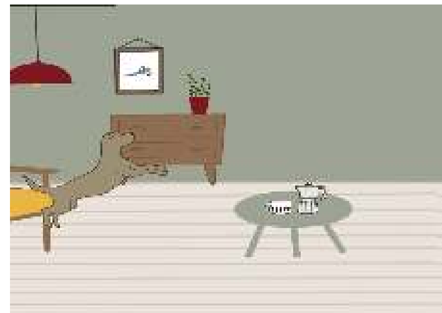
Figure 3.4: Timeline of events happening in *Airplane Game*

3.3 Dog Game

Within the third game, children practice an articulation of monosyllabic words.

The game starts with a view centred on a bone lying on a floor of a furnished living room. The view starts moving through the living room until it reaches a dog sitting on an armchair. The dog jumps down from the armchair and the game begins. The basic idea of this game is very simple – every time the child pronounces a monosyllabic word or makes a short sound, the dog jumps in the direction of the bone. Multi-syllabic words or longer sounds have no effect on the dog. When he gets close to the bone, he runs to it, picks it up and wiggles his tail, after which the game ends.

(a) Initial screen layout



(b) Dog jumping after successful pronunciation of monosyllabic word

Figure 3.5: Images of *Dog Game*

3.3.1 Timeline of events

The example timeline that indicates events in *Dog Game* is displayed in Figure 3.6:

- *t1*: The game is started. The sound is not processed yet, because the initial animation has to be played first.
- *t2*: The initial animation finishes and the sound processing is initialised.
- *t3, t6, t8, t10*: The flag indicating that the processed sound amplitude is in the voice range is switched on.
- *t4*: The sound is too long. Another flag indicating that is switched on. The dog remains still.
- *t5*: The too-long-sound-flag is switched off.
- *t7, t9*: The processed sound is classified as a monosyllabic word. The dog jumps towards the bone.

- *t11*: The sound is too loud. Flag indicating that is switched on.
- *t12*: The too-loud-sound-flag is switched off.
- *t13*: After several jumps, the dog reaches the bone and the game is finished.

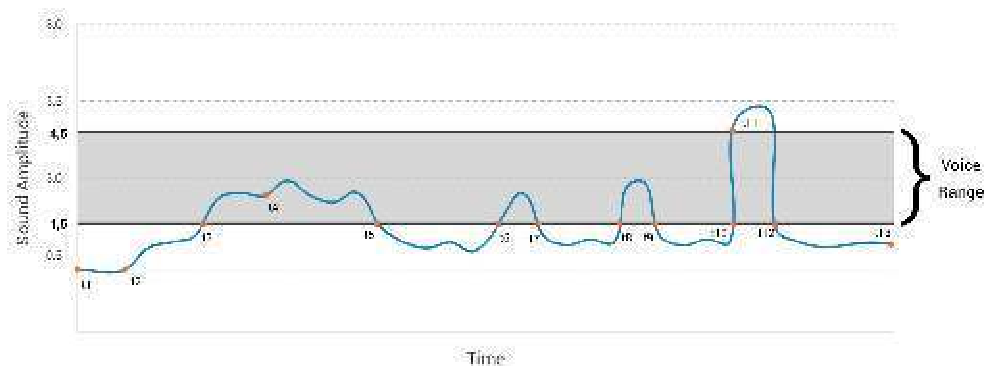


Figure 3.6: Timeline of events happening in *Dog Game*

3.4 Balloon Game

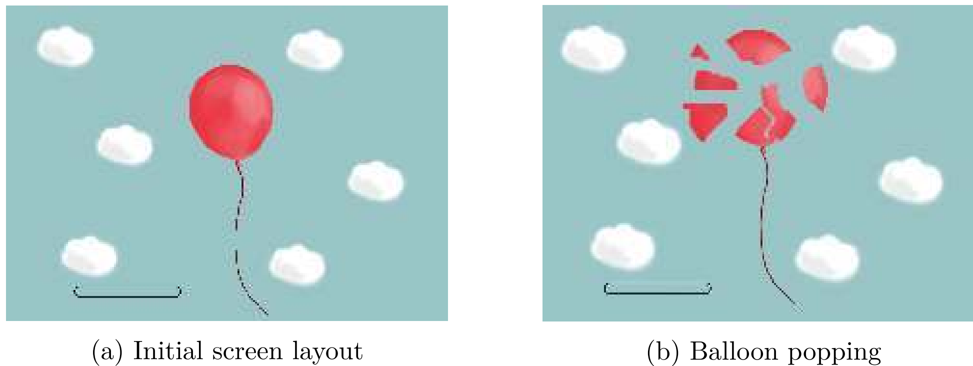
The lack of hearing feedback on speaking makes it difficult for people with a hearing impairment to control the volume of their spoken word. The last game focuses on this problem.

The game consists of a small balloon on a cloudy sky background. The balloon increases and decreases its size based on the volume of the child's voice. The goal of the game is to produce sounds in a *target volume range* for a certain period of time. In the bottom of the screen, there is an empty progress bar. When the child reaches the *target volume range*, the balloon starts moving up and down and the progress bar starts showing progress. Changing the volume to one that is outside of the *target volume range* resets the progress bar. Keeping the voice volume in the *target volume range* until the progress bar is full wins the game. The balloon turns and a smiley face appears on it. Screaming or loud speaking causes the balloon to pop, indicating that the sound was too loud.

The *target volume range* and the pop threshold are values that can be adjusted inside the settings of the application to fit the specific needs of every child.

3.4.1 Timeline of events

The example timeline that indicates events in *Balloon Game* is displayed in Figure 3.8:



(a) Initial screen layout

(b) Balloon popping

Figure 3.7: Images of *Balloon Game*

- *t1*: The game is started and the sound processing is initialised.
- *t2*: The processed sound amplitude reached the target voice range. The balloon starts slightly moving up and down to indicate that this is the correct volume.
- *t3*: The sound amplitude grew above the target voice range. The balloon stops moving.
- *t4*: The sound amplitude returned to the target voice range. The balloon starts moving again.
- *t5*: The sound amplitude dropped below the target voice range. The balloon stops moving.
- *t6*: The sound amplitude reached the pop threshold. The balloon pops and disappears.
- *t7*: After a short period of time, the balloon appears again and the game continues.
- *t8*: The sound amplitude remained in the target voice range for the required amount of time. The balloon stops moving, turns and a smiley face appears on it.
- *t1-t8*: The size of the balloon scales according to the sound amplitude.

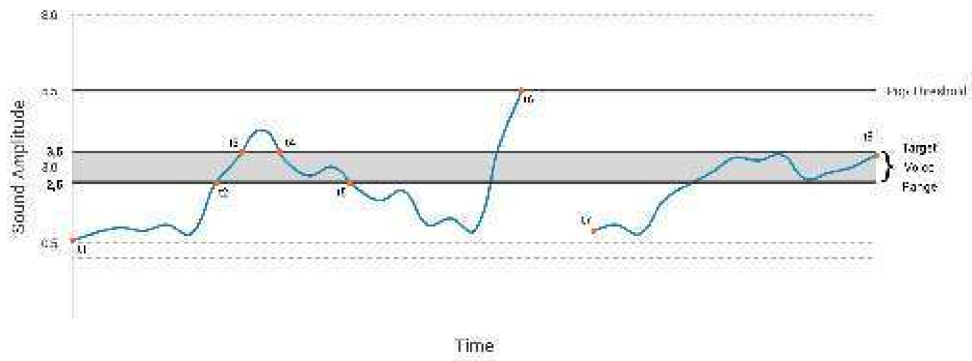


Figure 3.8: Timeline of events happening in *Balloon Game*

Chapter **4**

Implementation

This chapter provides a comprehensive description of the implementation of the *Hlásek* application. Firstly, the most significant technologies that were used are described. Secondly, the implementation of the core parts of the application, sound processing, game implementation, and event logging, are discussed. Finally, the third party libraries that were used are mentioned.

4.1 Technologies

4.1.1 Reactive Programming

According to the article written by *André Staltz*, *"Reactive programming is programming with asynchronous data streams"* [14]. The data stream in this respect can be anything, for instance, variables, events, caches and data structures. It is created asynchronously. This section will firstly explain the basics of reactive programming, and then describe how it was used in the implementation of *Hlásek*.

One of the implementations of reactive programming is *ReactiveX* (*Rx*), which exists for numerous programming languages. The following subsections describes reactive programming using the *Swift* library *RxSwift* [15]. The theoretical part is based on the articles from *Sebastian Boldt* [16] and *André Staltz* [14], and the *RxSwift* documentation [17].

4.1.1.1 Basics

The `Observable` object is an object which creates the elements of the data stream and emits events. It is, for example, the result of a database query request. The emitted events are:

- `next(value: T)`: Called when a value or a collection of values is added to the `Observable` data stream. The `value` parameter contains the actual value from the stream.

- `error(error: Error)`: Called when an error, such as download error, is encountered. It terminates the data stream.

- `completed`: Called when the data stream ends normally.

The `Observer` object is an object that `subscribe`s to the events emitted by the `Observable` object. The `Observable` object does not create any elements of the data stream until the first `Observer` `subscribe`s to it. Multiple `Observers` can `subscribe` to single `Observable` and receive the same events emitted by it. Each `Observer` receives events emitted by the `Observable` until the subscription is `dispose`d.

The `DisposeBag` object is a virtual bag of `Observer` objects. It `dispose`s inserted `Observers` when it is de-initialised. After every subscription, the `Observer` should be added to the `DisposeBag`.

The `Completable` object is a special type of the `Observable` object, which only emits the `completed` and the `error` events. It is used for asynchronous operations that does not return any data, such as data uploading.

The `Single` object is a special type of the `Observable` object. The `Single` object returns data only once. It is used for asynchronous operations, such as API calls, where only a single batch of data is expected to be received.

The `Subject` is a special type of the `Observable` object, to which it is possible to add elements dynamically.

4.1.1.2 Operators

The `Operator`s transform, filter or combine the elements emitted by the `Observable` data stream. The output of the `Operator` is another `Observable` data stream, so multiple `Operator`s can be chained together. Also, custom operators can be created. The following list contains some useful `Operator`s:

- `map`: transforms the elements emitted from the `Observable`,

- `skip`: skips a certain amount of elements emitted from the `Observable`,

- `filter`: filters the elements by a specific condition,

- `merge`: merges two `Observable`s,

- `concat`: concatenates the output of multiple `Observable`s so that the elements of the second one are appended to the end of the first one.

The comprehensive list of all *ReactiveX* operators can be found in [18].

4.1.1.3 Enhancing VIPER Architecture

The features of the reactive programming were used to enhance the *VIPER* architectural pattern. The granularity of responsibilities is one of the most significant advantages of *VIPER*. However, the coupling between some components can still be improved.

The *View* owns the instance of the *Presenter*, and the *Presenter* holds the instance of the *View*. The *View* sends user interactions to the *Presenter*, and the *Presenter* notifies the *View* about updates. Similarly, the *Presenter* owns the instance of the *Interactor*, and the *Interactor* holds the instance of the *Presenter*. *Presenter* sends non-UI related logic to the *Interactor*, and the *Interactor* returns the results (often asynchronously). The component interaction is captured in the snippet below.

```
class Interactor {
    func getNewUsers() {
        ...
        // The users are fetched from the database and
    filtered
        presenter.showNewUsers(users: downloadedUsers)
    }
}

class Presenter {
    func getNewUsers() {
        interactor.getNewUsers()
    }

    func showNewUsers(users: [UserDBEntity]) {
        // Convert UserDBEntity to the User model
        view.presentUsers(users: users)
    }
}

class View {
    func viewDidLoad() {
        presenter.getUsers()
    }

    func presentUsers(users: [User]) {
        // Presentation of the users
    }
}
```

Listing 4.1: *VIPER* component interaction without reactive programming

The code below captures the same interaction between modules once the reactive programming is integrated into the application.

```
class Interactor {
    func getUsers() -> Observable<[UserDBEntity]> {
        ...
        // The users are fetched from the database
        observable.on(.next(fetchedUsers))
        ...
    }
}

class Presenter {
    func getNewUsers() -> Observable<[User]> {
        return interactor.downloadUsers()
          .filter { user in
             user.isNew == true
          }
          .map { user in
             // Convert UserDBEntity to the User model
          }
    }
}

class View {
    func viewDidLoad() {
        presenter.getUsers()
         .subscribe(onNext: { users in
            // Presentation of the users
         })
         .disposed(by: disposeBag)
    }
}
```

Listing 4.2: *VIPER* component interaction with reactive programming

The sample code above features some of the advantages provided by the reactive programming. It removes the dependence of the *Interactor* on the *Presenter* and the dependence of the *Presenter* on the *View*. This allows the creation of more generic *Presenter*s and *Interactor*s. As a result, multiple *Presenter*s with the same logic can use the same *Interactor*, and multiple *View*s can use the same *Presenter*. A comparison of approaches can be seen in the examples above. In the first example, the *Interactor* has a function `getNewUsers` which fetches the users from the database and filters them by the `isNew` property. If a different *Presenter* wants to use this *Interactor* and

get users filtered by, for example, their age, a new function has to be created. Moreover, the *Interactor* needs to have the instance of the *Presenter* for returning of the results. In the second example, the *Interactor* is responsible only for fetching of users from the database. The *Presenter* does the filtering. Different *Presenter*s can `subscribe` to the same *Interactor* and filter the results according to their needs.

Additionally, it simplifies the control of the data flow and organises data transformations. The enhanced *VIPER* architecture is shown in Figure 4.1.

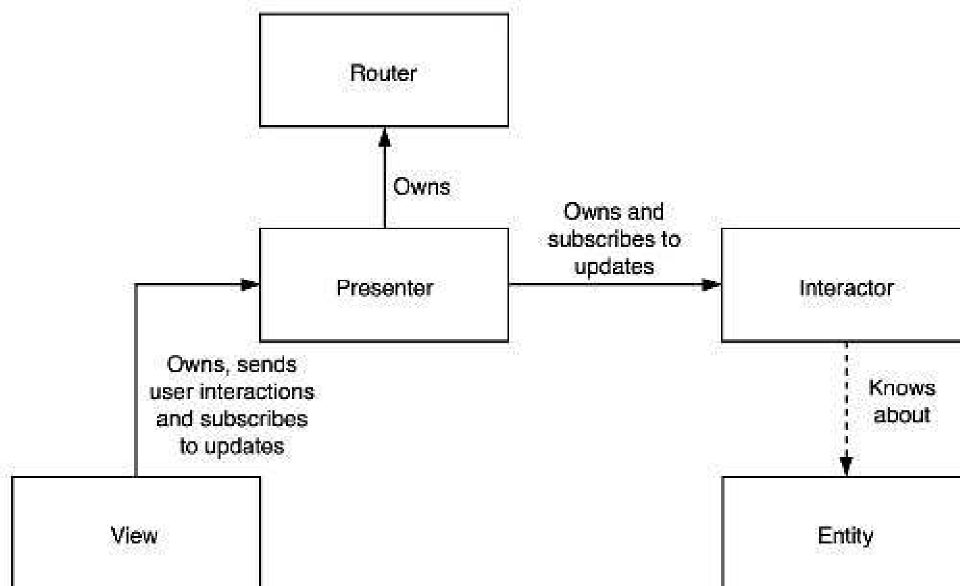


Figure 4.1: Viper architecture enhanced by reactive programming

4.1.1.4 Reactive Programming in Games

The reactive programming was also utilised in the games. Each game has a *Game Manager* class responsible for the sound processing, the event handling, and the analytics reporting, and a *Game Scene* class that renders the game. Games with an initial animation do not process the sound from the very beginning. In order to avoid hard-coding of the duration of the initial animation in *Game Manager*, the initial animation function in *Game Scene* returns `Completable`. Then, no matter the length of the initial animation, sound processing does not start until the `completed` event is received.

It is used in the same way for the game-finished event. The most significant application of the reactive programming is in the sound processing. It is described thoroughly in Section 4.2.2.

4.1.2 Dependency Injection

Dependency injection is a design pattern in which one object (dependency injector) provides (injects) services (dependencies) for other objects. Instead of initialising the object with each service, it is initialised with the dependency injector and extracts the services from it. This way, the objects are not responsible for the origin of the services. Furthermore, without dependency injection, the services that are used by multiple classes need to exist as *Singleton* classes. Dependency injection ensures that the services will be initialised only once – when the dependency injector is being created. The intent of dependency injection is to achieve a better separation of concerns.

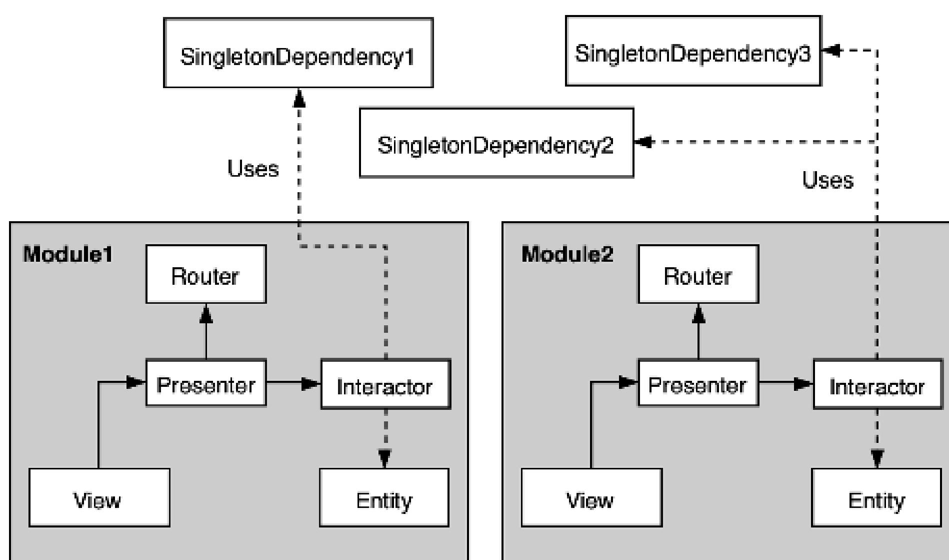


Figure 4.2: Application without dependency injection

The *Swift* programming language improves dependency injection with a feature called *protocol composition*. *Protocol composition* was added to *Swift* in version 3.0. It allows creating of properties whose type is a composition of multiple protocols (*Swift* alternative of interfaces). Only objects that conform to all of these protocols can be assigned to these properties.

Dependency injection smoothly combines with the *VIPER* architecture. The *Interactor* is the component responsible for interaction with external services. Dependencies need to be injected only there. The implementation of the dependency injection in the application follows these steps:

- A protocol is defined for every dependency in the application. The protocol has one property. The property type is equal to the dependency type.

```
protocol HasDatabaseInteractor {
    var databaseInteractor: DatabaseInteractor
}

class DatabaseInteractor {}
```

- The `AppDependencies` object is created. This object conforms to all dependency protocols and has a property for all dependencies.

```
struct AppDependencies: HasDatabaseInteractor,
    HasSoundProcessingInteractor,
    HasFirebaseAnalytics {
    var databaseInteractor: DatabaseInteractor
    var soundProcessingInteractor:
    SoundProcessingInteractor
    var firebaseAnalytics:
    FirebaseAnalyticsInteractor
}
```

- Each dependency is instantiated in the `prepare` function of the `AppDependencies` object.

- *Interactor*s define a dependencies property. Its type is a composition of protocols of services they need. Dependencies are passed to the init function of the *Interactor*.

```
class Interactor {
    typealias Dependencies =
    HasSoundProcessingInteractor &
    HasFirebaseAnalytics
    var dependencies: Dependencies

    init(dependencies: Dependencies) {
        self.dependencies = dependencies
    }
}
```

- The `AppDependencies` object conforms to all dependency protocols so that it can be passed to the init function of every *Interactor*. With the protocol composition, the *Interactor* extracts only the services that it needs and makes the rest of the dependencies inaccessible.

The comparison of application with and without dependency injection can be seen in Figures 4.2 and 4.3.

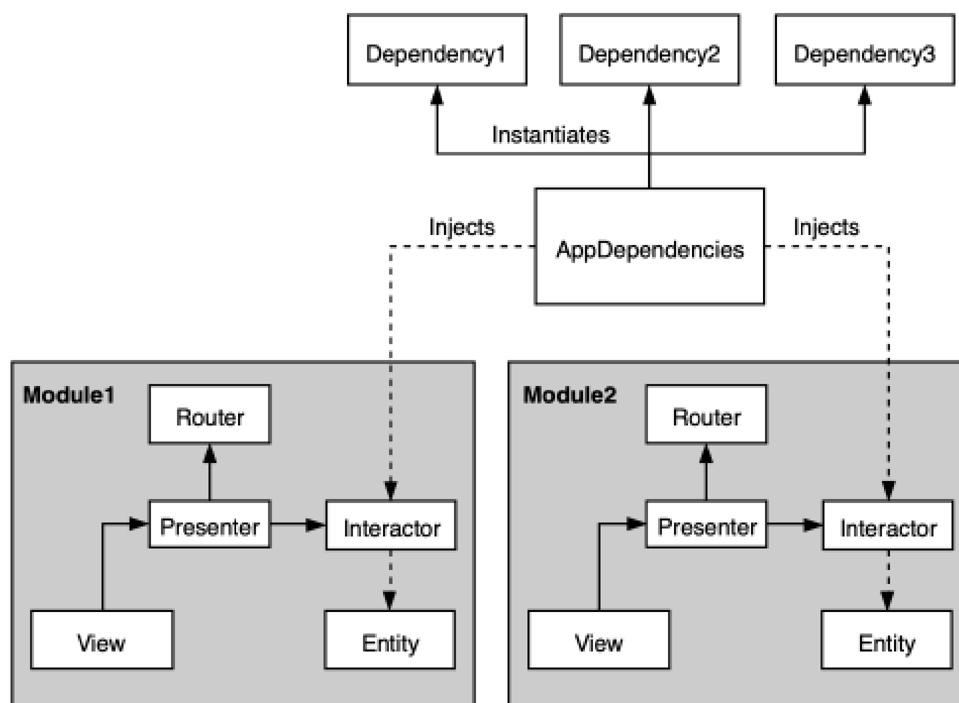


Figure 4.3: Application with dependency injection

4.2 Sound Processing

Unlike the majority of mobile application games that are controlled by touch gestures and device orientation, *Hlásek* application uses a sound as the primary controlling input. After research on libraries and frameworks that could be used for sound processing, two possible candidates were found – the *AudioKit* framework [19] and the *Speech* framework [20].

4.2.1 Speech

Speech is an *Apple* framework created for the recognition of a spoken word. It performs speech recognition on a live or a prerecorded audio and receives valuable information about it, such as transcriptions, alternative interpretation and confidence levels of the result. Although the framework is robust, it was not suitable for the *Hlásek* application. The main goal of *Hlásek* is to recognise the amplitude and the frequency of processed sound. *Speech* is not capable of doing that. Recognising the actual content of spoken word might be useful in the future extensions of the application; however, for purposes of this master thesis, it proved to be useless.

4.2.2 AudioKit

AudioKit is a powerful framework for audio synthesis, processing, and analysis. Among numerous other functions, it enables working with oscillators, sound envelops, and mixers, and allows additive sound synthesis, and sampling. *Hlásek* utilises only a small subset of these functions.

It needs to capture sounds from the microphone of the device and derive their frequency and amplitude. The process steps follow.

1. An instance of `AKMicrophone` is created which represents audio from the standard input.

2. An instance of `AKFrequencyTracker` is initialised with the `AKMicrophone` instance, making it track sounds from the standard input.

3. An instance of `AKBooster` is created. `AKBooster` receives a sound from its input and amplifies/quietens it by a specified `gain`. `AKFrequencyTracker` is passed as the input, and the `gain` is set to 0, turning sounds from the input to a silence.

4. The instance of `AKBooster` is set as an output node of the *AudioKit*. That way the framework does not produce any sounds.

5. *AudioKit* is started.

6. A new timer is created. Every tick of a timer the frequency and amplitude values are read from the `AKFrequencyTracker`.

Every game uses sound processing. Therefore, an *Interactor* class that manages sound processing was created and injected into every game. The class initialises the *AudioKit* framework. Also, it exposes a function that starts the timer with a designated tick frequency and three `Observable` properties. The list of these properties follows.

- `volume`: The amplitude of the sound from the standard input that is transformed to have values ranging from 0 to 200. Updated with every tick of the timer.

- `frequency`: The frequency of the sound from the standard input. Updated with every tick of the timer.

- `soundMade`: An enumeration of important sound events. Updated every time an interesting sound event occurs. The possible values of this property are in the following list (the `voice range` value mentioned in the list has been adjusted after several testing sessions to meet the needs of the children).

- `inVoiceRange`: The volume of the sound has been in a designated voice range for a required number of timer tics. The required number of timer tics is set in the function that starts the timer. It is a game-dependent value.
- `belowVoiceRange`: The volume of the sound is below a designated voice range.
- `aboveVoiceRange`: The volume of the sound is above a designated voice range.
- `inVoiceAllowedDuration`: The number of timer tics, during which the volume of the sound has been in a designated voice range, is neither shorter nor longer than a required duration range. The required duration range is set in the function that starts the timer. It is a game-dependent value.
- `aboveVoiceAllowedDuration`: The number of timer tics, during which the volume of the sound has been in a designated voice range, is longer than a required duration range.

Each game initialises the timer according to its needs and `subscribe`s to the observable properties that are important in that particular game. Afterward, it reacts to the changes of those properties without any knowledge of the source of the changes.

4.3 Game Implementation

When deciding which framework to use on the implementation of the games, an obvious choice was a *SpriteKit* framework [21]. *SpriteKit* is an *Apple* framework for creating a high-performance and battery-efficient 2D games. Integration with *SceneKit*[9] [22] is supported, and performing of physics effects (such as adding of force fields and collision detection) and their animation is greatly simplified. Also, *SpriteKit* is integrated into the *XCode IDE* and does not require additional configuration.

The key aspects of a game creation will be explained in the following sections.

4.3.1 Setting Up

Each game consists, at least, of two files – a *SpriteKit Scene* and a *Swift* source file associated with that particular scene. The *SpriteKit Scene* is used as a drag-and-drop editor for the game layout as displayed in Figure 4.4. Each node is positioned to its designed location. It receives a texture image, name, size, and other optional properties.

[9] A high-performance rendering engine with an API for working with 2D and 3D assets.

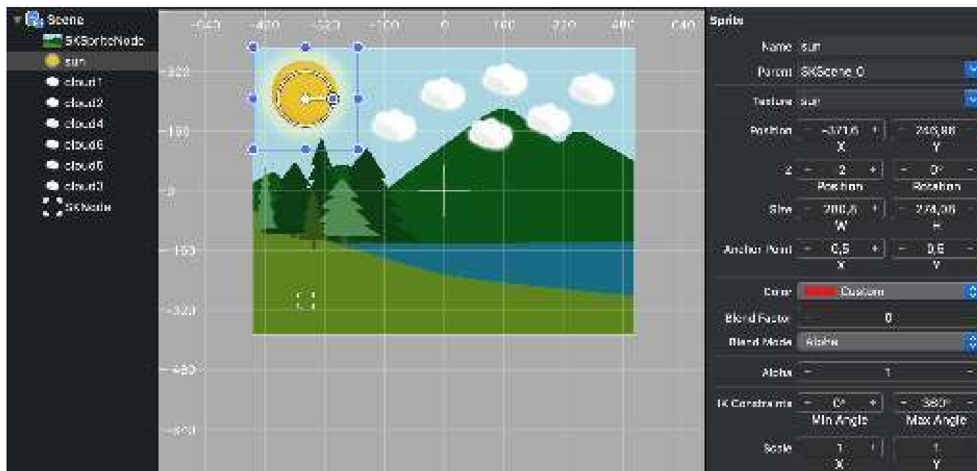


Figure 4.4: Layout of *SpriteKit Scene* editor

Afterward, nodes that are expected to be animated need to be connected with the *Swift* source file. This can be achieved by the following command (assuming the node is called *sun* in the *SpriteKit Scene*):

```
let sunNode = childNode(withName: "sun")
```

4.3.2 Animations

Nodes are animated by the `SKAction` class. `SKAction` is an object that, when executed on a node, changes its structure or content. Each action has a duration. Various operations can be performed with actions, for example, grouping, sequencing or delaying. There is an extensive set of predefined actions, some of which are:

- `moveTo(x: y: duration:)`: moves the node to a specified location over the duration,
- `rotate(byAngle: duration:)`: rotates the node by a specified angle over the duration,
- `applyForce(force: atPoint: duration:)`: applies a force to a specified point of the node over the duration (this action requires the node to have a `PhysicsBody` described in Section 4.3.3),
- `sequence(actions:)`: connects multiple actions, so that the next one starts executing when the previous one finishes.

Created action is run on a node in this way:

```swift
let scaleUpAndDownAction =
   SKAction.repeatForever(
        SKAction.sequence([
          SKAction.scale(to: 1.1, duration: 0.75),
          SKAction.scale(to: 1.0, duration: 0.75)
]))
sunNode.run(scaleUpAndDownAction)
```

Listing 4.3: Example of node animation

4.3.3 Physics Effects

Animating nodes is useful in various situations; however, when the games get more complicated, it would be problematic to keep track of all the animations that are happening at some point in time.

Within every *SpriteKit Scene*, there is a `PhysicsWorld` object. This object simulates the physics of the real world. In order to make a node react to the forces of the `PhysicsWorld`, it needs to have a `PhysicsBody`. Creating a `PhysicsBody` for a node tells the scene that it should add physics simulations to it.

A `PhysicsBody` provides various properties. Adjusting these properties can achieve behaviour that corresponds to the behaviour of a real object the node represents. These properties are, for instance, a mass, density, friction, and restitution. Furthermore, it allows applying of forces, angular forces, and torques to the node.

Once the node has a `PhysicalBody` it starts automatically interacting with other nodes with a `PhysicalBody`. Collisions can be controlled by settings of a `categoryBitMask` and a `collisionBitMask`. Following example demonstrates how to make an airplane ignore clouds but collide with the ground and trees.

```swift
let planeBitmask: UInt32 = 0x1 << 0
let cloudBitmask: UInt32 = 0x1 << 1
let groundBitmask: UInt32 = 0x1 << 2
let treeBitmask: UInt32 = 0x1 << 3

plane.physicsBody?.categoryBitmask = planeBitmask
cloud.physicsBody?.categoryBitmask = cloudBitmask
ground.physicsBody?.categoryBitmask = groundBitmask
tree.physicsBody?.categoryBitmask = treeBitmask

plane.physicsBody?.collisionBitmask = groundBitmask |
    treeBitmask
```

Listing 4.4: Example of collision control

4.3.4 Animating Textures

The texture animation is a process in which a sequence of successive images is played in a short period of time. It is used for a simulation of a movement of nodes. Animating, for instance, a sequence of images shown in Figure 4.5, simulates a dog jumping movement. Using texture animations in a game considerably improves the smoothness and overall user-friendliness of that particular game.

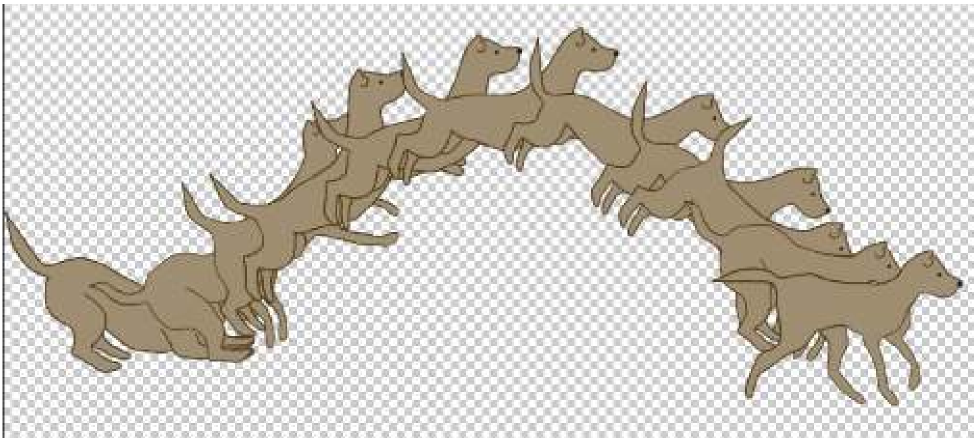


Figure 4.5: Sequence of images simulating dog jumping movement

4.3.5 Update Function

The `update` function is a function that is called every time a scene renders its content, which happens 60 times in a second. Therefore, the `update` function is a place where the game-over conditions are checked, the endless scrolling backgrounds properly positioned and more.

4.4 Event Logging

The main goal of the *Hlásek* application is to improve children's ability to speak. In order to be able to check, whether the game level is properly set and children are making progress, monitoring of children's performance is necessary. The supervisor can evaluate the reactions and understanding of games during the children's initial contact with the application. However, tracking of long-time progress by a supervisor can be difficult. For that reason, event logging was integrated into the application, so that long time progress of children can be easily tracked.

4.4.1 Firebase

Firebase is a mobile and web application development platform. It provides 18 products that are being used by hundreds of thousands of applications daily. Among these products are, for example, *Realtime Database*, *Authentication Kit*, *Machine Learning Kit*, *Google Analytics* and *Predications*. [23]

Hlásek utilises the *Google Analytics* product of *Firebase*. It allows applications to log custom events during the application run. Each event can have a custom name and numerous parameters assigned to it. Once logged, the event is sent to the *Firebase* console, where it is stored and can be used for further processing.

Similarly to the sound processing, event logging is used in every game of the application. Hence, a *FirebaseAnalyticsInteractor* was created and injected into every game with dependency injection. It exposes two functions for logging of general events and game events.

- General events are events happening in all games. They are evoked either by sound or the game itself. The list of events is shown in Table 4.1.

Event	**Game**			
	Cloud	**Airplane**	**Dog**	**Balloon**
Game Started	x	x	x	x
Game Finished	x	x	x	x
Game Exited	x	x	x	x
Long Sound			x	
Short Sound	x	x		
Quiet Sound	x	x	x	
Loud Sound			x	

Table 4.1: List of general events and games that evoke them

- Game events are events that only happen in some games. The list of game events is shown in Table 4.2.

Each event has parameters specifying the time it happened, the game it happened in, and the user who is currently logged in. *Firebase* is only capable of event analysis. For analysis of event parameters, the data needed to be sent to *BigQuery*.

4.4.2 BigQuery

BigQuery is a serverless cloud data warehouse [24]. When used to its full potential, it provides powerful Business Intelligence Engine and built-in machine learning. For the *Hlásek* application, it was used for the possibility to analyse

Game	**Event**
Cloud	cloudLifted
Airplane	initialSoundMade startedDescending startedAscending stopped
Dog	dogJumped
Balloon	balloonPopped soundBelowTargetRange soundInTargetRange soundAboveTargetRange

Table 4.2: List of game events

event parameters and for its simple integration to the *Firebase* console.

BigQuery has its query language, but it also supports SQL. Besides events logged by the *FirebaseAnalyticsInteractor*, numerous other events, regarding sessions, user interactions, screen opening, and more, are logged automatically. Each event is accompanied by numerous parameters, such as the geolocation, the device type and the application information. Queries that filter out the unwanted information and only leave the essential events and parameters were created. These queries are:

- `allDataQuery`: interactions of all users in all games in a specified period of time,
- `cloudGameQuery`: interactions of all users in *Cloud Game* in a specified period of time,
- `airplaneGameQuery`: interactions of all users in *Airplane Game* in a specified period of time,
- `dogGameQuery`: interactions of all users in *Dog Game* in a specified period of time,
- `balloonGameQuery`: interactions of all users in *Balloon Game* in a specified period of time,
- `userQuery`: interactions of specific user in all games in a specified period of time.

4.5 Other Libraries

Besides *AudioKit*, *SpriteKit*, *RxSwift*, and *Firebase*, the application uses third-party libraries in the following list.

- *Stevia Layout* [25]: A library simplifying the usage of *Autolayout*[10].
- *R.swift* [26]: A library changing the weak string-typed access to resources like images and fonts to the strongly typed access with available autocomplete. To demonstrate the usage of the R.swift library, let's assume there is an image in the project named *exampleImage*. Without *R.swift* the image would be accessed like this:

  ```
  let image = UIImage(named: "exampleImage")
  ```

  R.swift changes the image access into this:

  ```
  let image = R.image.exampleImage()
  ```

- *Realm* [27]: A mobile database that runs directly inside the mobile devices.
- *SYNBase* [28]: A multifunctional library with various handy extensions, such as *Realm Database* reactive extension and *float* extension for scaling its value according to the screen size.
- *RxDataSources* [29]: A reactive extension for table views and collection views.
- *Swiftlint* [30]: A tool that enforces *Swift* coding style and conventions.
- *Fabric* [31]: An online service for over-the-air application installation and testing.

4.6 Overview

The composition of the *Hlásek* application with technologies and core parts described in this chapter is shown in Figure 4.6.

[10]A system of constraints and rules that governs the layout of components for different screen sizes

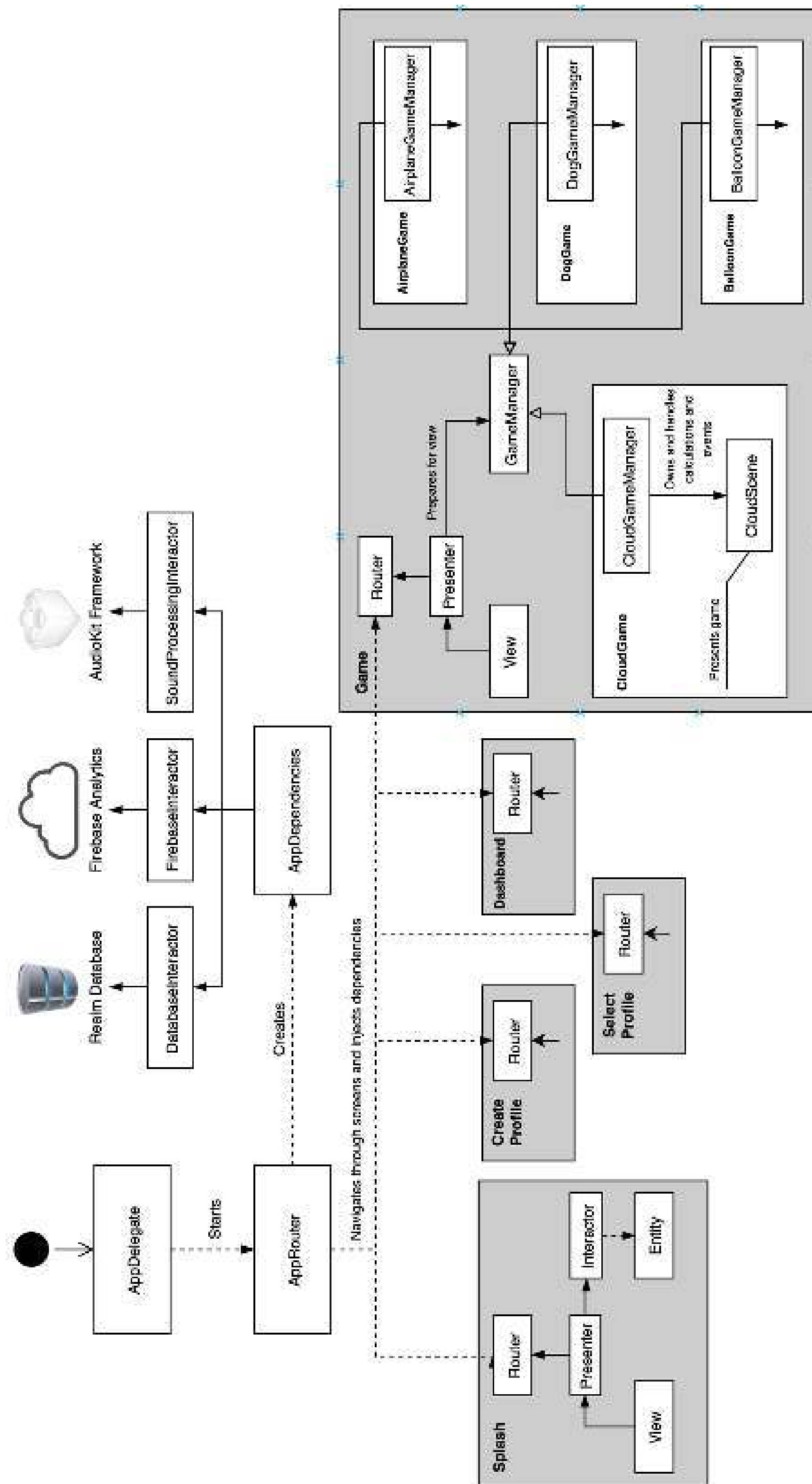


Figure 4.6: Composition of classes in *Hlásek*

Chapter **5**

Deployment and Distribution

Once the MVP of the *Hlásek* application was finished, it had to be deployed and then distributed to the mobile devices of the *SPC*, where it was supposed to be tested. It was expected that everyday testing would provide rich feedback with numerous change requests. To achieve meaningful testing results, quick reaction to these change requests had to be assured. The key aspects of the deployment process and the distribution to a limited number of devices will be discussed in this chapter.

5.1 Deployment of iOS Applications

Deploying an application to physical devices requires it to be digitally signed and linked with an adequate provisioning profile.

5.1.1 Code Signing

Code signing is a security technology that the developers use to certify, that they created the application. Once the application has been signed, the system can detect when it changes [32]. The code is signed with a code signing identity, which consists of a private key and a digital certificate. The digital certificate contains the public key that complements the private key. Besides, it also has a usage extension that enables it to sign code [33] .

In order to generate a certificate for a signing identity, the developer needs to create a *Certificate Signing Request* (*CSR*) from the *Keychain Access* application on the Mac. When the *CSR* is created, public and private keys are automatically generated. The private key is automatically stored in the *Keychain* on the Mac. The requested certificate is the public half of the key pair. After the developer submits the request to the *Apple Developer Portal*, a new certificate is generated.

There are two different types of certificates – Development certificates and Production certificates. Development certificates are used for code signing of

the versions of the application that are run on the testing devices. Production certificates can either be *Ad Hoc* or *App Store* certificates. *Ad Hoc* certificates sign versions of the application that are distributed to a limited number of registered devices. *App Store* certificates are used for the versions that are sent to the *AppStore* [34].

5.1.2 Provisioning Profiles

Besides code signing, running the application on a real device also requires a particular type of a system profile called *Provisioning Profile*. *Provisioning Profiles* are used to launch applications on devices and allows them to use certain services [35]. With the provisioning profile, developers can choose which devices will be able to run the application and which services can the application access [36]. Specifically, these services are *In-App Purchase*[11], *Game Center*[12], *Push Notifications* and others.

Apple distinguishes between two different provisioning profile types – *Development* and *Distribution Provisioning Profiles*. *Development Provisioning Profiles* are used for installing on the test devices. *Distribution Provisioning Profiles* are separated into four categories – *In House*, *Ad Hoc*, *tvOS* and *App Store* [37].

- *In House*: A *Provisioning Profile* for any company device. It has no limitation on the number of devices that can install the application.
- *Ad Hoc*: A *Provisioning Profile* for distribution outside of the company. Each device to which the application is installed, has to be registered. The limit of devices is 100. Unlike *In House* provisioning profile, *Ad Hoc* provisioning profile allows developers to control the target devices.
- *tvOS*: A *Provisioning Profile* for devices with *tvOS*[13].
- *AppStore*: A *Provisioning Profile* for upload to the *AppStore*.

5.2 Distribution

In the scope of this diploma thesis, the application was to be distributed only to devices of the *SPC*. The devices were registered under the *Apple* developer account of company *Synetech s.r.o.* The author of this thesis is employed in this company and was allowed to use their developer account for the purposes of this application. An appropriate certificate and provisioning profiles were created.

XCode IDE archives the source code of the application and creates an

[11] A possibility to buy additional products inside the application.
[12] An online multiplayer social gaming network.
[13] An operating system that runs on the Apple televisions.

`.ipa` file (*iPhone Application Archive*). The created archive is afterward either uploaded to the *AppStore* for worldwide publication or to one of the online services for over-the-air installation and testing. Examples of these services are *TestFlight* service [38] and *Fabric* service [31]. *Hlásek* was distributed via *Fabric* service.

5.2.1 Fabric

Fabric is a mobile platform with modular kits that can be combined and matched to satisfy application needs [39]. A list of these kits follows.

- *Answers* enables real-time analytics with insights into core goals, such as retention, growth, and engagement.
- *Beta* simplifies sending of the beta builds to the user devices.
- *Crashlytics* records all application crashes with a detailed description regarding the device type, a software version, the line of the source code, and so forth.

Crashlytics and *Beta* kits were integrated into the *Hlásek* application.

Chapter 6

Testing

Testing is an essential part of a software development process. *Hlásek* non-game screens have basic user interface with a minimum responsibilities. Therefore, the whole chapter is dedicated to the testing of each game.

Testing was performed in the *SPC* on an *iPad Air 2* device. Children visited the center for their usual therapist sessions and *Hlásek* was tested on each of them. The sessions were held under supervision of *PhDr. Jarmila Roučková*, who was the kid's speech therapist, and *Bc. Denisa Šleisová*, who was already mentioned in the *Analysis and Design* chapter. The application was tested by 10 children. The profiles of tested children are listed in Apendix C.

6.1 Testing Results

The short-term testing was divided in the following three iterations.

6.1.1 First Testing Iteration

The first testing iteration was performed on the very first released version of the application. In the beginning, the children did not understand that they need to control the application with their voice. They used their hands in the way they were used to control other tablet applications. It was necessary to show them that they had to use their voice. Firstly, they did not manage to complete a single game. The proper way to control the game was demonstrated. Then, the therapist helped the children by a tactile control of the vocal cords. Afterward, they were able to start using their voice. Once they saw that the game moved with sounds, they became very engaged in it. After the first session, children were only able to finish the *Cloud Game* and the *Airplane Game*. They appreciated the end game animation in the *Cloud Game*.

The first testing iteration resulted in the following change requests:

- the distance that airplane travels in the *Airplane Game* should be longer,
- the visibility of the "back" button on screens is bad,
- two more clouds should be added to the *Cloud Game* (it only had 4 clouds during the first iteration),
- the reaction time of the airplane (descending and ascending) is too slow,
- the jumping animation of the dog is too static and does not look good (it was not done by texture animation in the first iteration),
- *Airplane Game* should have a reward in the end (the airplane was not landing next to the house),
- the exit button in the *Dog Game* is not visible.

All of these change requests were resolved.

Overall, the first testing iteration was successful. It proved that the games are entertaining and not too simple for children to complete. With some help and longer practice, children should slowly improve and complete the games without a therapist's help.

6.1.2 Second Testing Iteration

The second testing iteration happened two weeks later. Some children still needed help by a tactile control of the vocal cords and a demonstration by the therapist. Others were able to complete the games by themselves. They were able to complete every game except the *Dog Game*. Only one child was able to complete the *Dog Game*. In the *Balloon Game*, majority of children enjoyed popping of the balloon, rather than keeping the volume in the *target volume range*.

The second testing iteration resulted in the following change requests:

- the distance that airplane travels should be slightly shorter,
- the settings of the balloon *target volume range* and the pop threshold should be removed and the values should be hardcoded,
- the reaction speed of the dog should be increased,
- *Dog Game* should have a reward in the end (the dog was not jumping to the bone),
- in the *Balloon Game*, the time for which the children need to hold the volume in the *target volume range* should be decreased.

It was decided that the settings in the *Balloon Game* would remain in the game for future testing. The increase of the reaction time in the *Dog Game* was not achieved. The way to distinguish the monosyllabic sounds from the multisyllabic sounds is to check whether there is a silence for a few moments after the monosyllabic sound was made. This extra check accounts for the delay in reaction time. Besides these two problems, all other change requests were resolved.

Overall, the second testing iteration was also successful. There was a visible progress in the children's ability to control the game.

6.1.3 Final Testing Iteration

The final testing iteration happened two weeks after the second testing iteration – one month after the first contact with the game. Children were most successful in the *Cloud Game*. They managed to lift the clouds easily. In the *Airplane Game*, the duration for which they were able to keep the airplane in the air has increased. The game was achieving the designated prolonging of their expiratory flow. The most problematic game was the *Dog Game*, where, due to the slow reaction time of the dog, children did not know how to control the game properly. Therapists have trained children to hold the correct volume for the *Balloon Game*, instead of popping the balloon with too loud voice, and children were succeeding in that.

The final testing iteration did not result in any change requests. It provided suggestions on possible future extensions of the game. These future extensions are discussed in Section 6.2 in the Conclusion chapter.

6.2 Testing Conclusion

The testing results were very positive. The games proved to be entertaining and their difficulty level set correctly. In the beginning, children struggled with each game. After one month, they were able to finish all games, except the *Dog Game* that would need to be modified. The testing also created numerous change requests. Majority of the change requests was resolved and the final form of the game seems to be valid.

The testing is an ongoing process. Denisa is regularly visiting the *SPC* and continues testing of the application. Every change request reported by her will be resolved even after submitting of this diploma thesis.

The application would need to be tested for at least six months in order to provide long-term testing results based on the event logging described in Section 4.4 of the Implementation chapter.

Conclusion

The primary goal of this diploma thesis was to create a mobile application for operating system iOS that would help children with hearing impairment improve their ability to use their voice. As a result of these requirements, the *Hlásek* application was created. *Hlásek* introduces four animated games, each of which is designed to improve a different aspect of children's voice control. *Cloud Game* shows them that they have to use their voice to control the application. *Airplane Game* exercises lengthening of the expiratory flow and the ability to hold a constant vocal phonation. *Dog Game* focuses on children's articulation of monosyllabic words. *Balloon Game* helps them to find the correct volume of their voice.

A part of this thesis was an analysis of existing applications that help children with speech development. Applications *Speech Viewer*, *Krtek* and *Mentio Hlas* were analysed. Application *Speech Viewer* inspired the composition of games in *Hlásek*.

The application has been deployed to a limited number of testing devices in the *SPC*. The application can be updated remotely.

The application was tested by ten children in three two-week-separate iterations. Testing resulted in several game change requests, out of which the majority was resolved. During the one-month testing period, children's ability to use their voice had improved. Also, they enjoyed the interaction with the game more than with the usual speech development tools. The long-time progress of each child can be analysed due to the event logging feature of the application. Every voice interaction with the game is reported to the online data storage. It allows an analysis of understanding and progress of each child.

During the implementation of *Hlásek*, we learned how to use the *VIPER* architecture, the *Apple* framework *SpriteKit*, and the sound processing framework *AudioKit*. I also tried working with reactive programming, dependency injection, and *Firebase Analytics*.

Future work

Currently, the application focuses on the speech and breathing development. In the future, exercises for the articulation of vowels and other vocal exercises could be added. Also, the problem discovered during the infield testing with the slow reaction of the dog in the *Dog Game* should be figured.

Moreover, current games could be extended by alternative versions based on the same principle (the same way it was done in the *Speech Viewer* application). Children would still complete the same vocal tasks but would be more engaged because of the variety of the visual representation of the task. Examples of such alternative versions are listed below.

- The principle of the *Dog Game* could be used to create a Cat game, where a cat chases a ball, or a Horse game, where a horse jumps over obstacles.
- The principle of the *Airplane Game* could be used to create a Boat game, where a boat sails over the ocean, or a hot air balloon that flies over the trees.
- The principle of the *Balloon Game* could be used to create a Fire game, where a fire is increasing its power based on the volume of the voice.
- The principle of the *Cloud Game* could be used to create a Bird game, where little birds fall from their nest and fly back to it with sounds.

It is expected that the application will be accessible for the public. At this moment, it is only used by a limited number of devices in the *SPC*. In order to make it available to the public, a few changes would need to be made in the design of the application.

Bibliography

[1] Česká unie neslyšících. *Statistiky počtu osob se sluchovým postižením*. [online]. [accessed: May 7, 2019]. Available from: https://www.cun.cz/blog/2017/05/17/statistiky-poctu-osob-se-sluchovym-postizenim/

[2] HÁDKOVÁ Kateřina. *Člověk se sluchovým postižením*. Praha: Univerzita Karlova, Pedagogická fakulta, 2016, ISBN 9788072906192.

[3] Český statistický úřad. *Informační společnost v číslech 2017*. [online]. [accessed: May 7, 2019]. Available from: https://www.czso.cz/documents/10180/46014808/061004-17_S.pdf/b9a0a83e-7a6f-4613-b1df-33fe8b5d1a8e?version=1.1

[4] Statista. *Number of apps available in leading app stores as of 1st quarter 2019*. [online]. [accessed: May 7, 2019]. Available from: https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/

[5] IBM. *Speech Viewer*. [software]. [accessed: Apr 26, 2019]. Available from: http://www.spectronics.com.au/article/3073

[6] CNet. *SpeechViewer III for Windows*. [online]. [accessed: May 7, 2019]. Available from: https://www.cnet.com/products/speechviewer-iii-for-windows-v-1-0-box-pack-1-user/

[7] KUFNER Josef. *Krtek Documentation*. [software]. [accessed: Apr 28, 2019]. Available from: https://josef.kufner.cz/programy/krtek/documentation.html

[8] KUFNER Josef. *Krtek*. [software]. [accessed: Apr 26, 2019]. Available from: https://josef.kufner.cz/programy/krtek/

[9] Ncurses. [software]. [accessed: Apr 26, 2019]. Available from: https://www.sallyx.org/sally/c/linux/ncurses

[10] PETRŽÍLKA Jan. *Mentio Hlas*. [software]. [accessed: Apr 26, 2019]. Available from: https://www.mentio.cz/Hlas

[11] PETRŽÍLKA Jan. *Ceník produktů Mentio Hlas*. [software]. [accessed: Apr 28, 2019]. Available from: https://www.mentio.cz/Cenik

[12] ORLOV Bohdan. *iOS Architecture Patterns*. [online]. [accessed: May 1, 2019]. Available from: https://medium.com/ios-os-x-development/ios-architecture-patterns-ecba4c38de52

[13] ZABŁOCKI Krzysztof. *Good iOS Application Architecture: MVVM vs. MVC vs. VIPER*. [online]. [accessed: May 1, 2019]. Available from: https://academy.realm.io/posts/krzysztof-zablocki-mDevCamp-ios-architecture-mvvm-mvc-viper/

[14] STALTZ André. *The introduction to Reactive Programming you've been missing*. [online]. [accessed: May 6, 2019]. Available from: https://gist.github.com/staltz/868e7e9bc2a7b8c1f754

[15] ReactiveX. *RxSwift*. [software]. [accessed: Apr 21, 2019]. Available from: https://github.com/ReactiveX/RxSwift

[16] BOLDT Sebastian. *Learn and Master the Basics of RxSwift in 10 Minutes*. [online]. [accessed: May 6, 2019]. Available from: https://medium.com/ios-os-x-development/learn-and-master-%EF%B8%8F-the-basics-of-rxswift-in-10-minutes-818ea6e0a05b

[17] ReactiveX. *ReactiveX for Swift*. [online]. [accessed: May 6, 2019]. Available from: https://github.com/ReactiveX/RxSwift

[18] ReactiveX. *The Operators of ReactiveX*. [online]. [accessed: May 6, 2019]. Available from: http://reactivex.io/documentation/operators.html

[19] AudioKit. [software]. [accessed: Apr 21, 2019]. Available from: https://github.com/AudioKit/AudioKit

[20] Apple Inc. *Speech*. [online]. [accessed: Apr 21, 2019]. Available from: https://developer.apple.com/documentation/speech

[21] Apple Inc. *SpriteKit*. [software]. [accessed: Apr 21, 2019]. Available from: https://developer.apple.com/spritekit/

[22] Apple Inc. *SceneKit*. [software]. [accessed: Apr 21, 2019]. Available from: https://developer.apple.com/documentation/scenekit

[23] Google Inc. *Firebase*. [software]. [accessed: Apr 21, 2019]. Available from: https://firebase.google.com

[24] Google Inc. *BigQuery*. [software]. [accessed: Apr 22, 2019]. Available from: https://cloud.google.com/bigquery/

[25] FreshOS. *Stevia Layout*. [software]. [accessed: Apr 21, 2019]. Available from: https://github.com/freshOS/Stevia

[26] KADIJK Mathijs. *R.swift*. [software]. [accessed: Apr 21, 2019]. Available from: https://github.com/mac-cain13/R.swift

[27] Realm. *Realm-Cocoa*. [software]. [accessed: Apr 21, 2019]. Available from: https://github.com/realm/realm-cocoa

[28] Synetech s.r.o. *SYNBase*. [software]. [accessed: Apr 21, 2019]. Available from: https://bitbucket.org/synetech/sycocoapods-specs/src/master/SYNBase/

[29] RxSwiftCommunity. *RxDataSources*. [software]. [accessed: Apr 21, 2019]. Available from: https://github.com/RxSwiftCommunity/RxDataSources

[30] Realm. *Swiftlint*. [software]. [accessed: Apr 21, 2019]. Available from: https://github.com/realm/SwiftLint

[31] Fabric. [software]. [accessed: Apr 17, 2019]. Available from: https://fabric.io

[32] Apple Inc. *About Code Signing*. [online]. [accessed: Apr 15, 2019]. Available from: https://developer.apple.com/library/archive/documentation/Security/Conceptual/CodeSigningGuide/Introduction/Introduction.html#//apple_ref/doc/uid/TP40005929-CH1-SW1

[33] Apple Inc. *Code Signing Tasks*. [online]. [accessed: Apr 12, 2019]. Available from: https://developer.apple.com/library/archive/documentation/Security/Conceptual/CodeSigningGuide/Procedures/Procedures.html#//apple_ref/doc/uid/TP40005929-CH4-SW1

[34] Apple Inc. *About Creating a Certificate Signing Request*. [online]. [accessed: Apr 15, 2019]. Available from: https://developer.apple.com/account/ios/certificate/

[35] Apple Inc. *Manage profiles on devices*. [online]. [accessed: Apr 15, 2019]. Available from: https://help.apple.com/xcode/mac/current/#/devaafd622d2

[36] Medium. *What is a provisioning profile and code signing in iOS?* [online]. [accessed: Apr 15, 2019]. Available from: https://medium.com/@abhimuralidharan/what-is-a-provisioning-profile-in-ios-77987a7c54c2

[37] Apple Inc. *iOS Provisioning Profiles*. [online]. [accessed: Apr 15, 2019]. Available from: https://developer.apple.com/account/ios/profile/create

[38] SATTERFIELD Benjamin; KOSMYNKA Trystan. *Testflight*. [software]. [accessed: Apr 17, 2019]. Available from: https://developer.apple.com/testflight/

[39] Fabric. *Many powerful tools, one easy platform*. [online]. [accessed: Apr 17, 2019]. Available from: https://docs.fabric.io/apple/fabric/overview.html

Appendix A

Acronyms

MVP Minimum Viable Product

XML Extensible markup language

SPC Special Pedagogical Center

IBM International Business Machines

SQL Structured Query Language

CSR Certificate Signing Request

Appendix B

Contents of enclosed DVD

```
readme.txt ....................... the file with CD contents description
src ....................................the directory of source codes
  impl ..................................implementation sources
  thesis ..............the directory of LaTeX source codes of the thesis
text ........................................the thesis text directory
  thesis.pdf .........................the thesis text in PDF format
```

Appendix C

Profiles of Tested Children

- **Child 1**
  - *Date of Birth*: 2013
  - *Occurrence of Hearing Impairment in Family*: Deaf parents and grandparents
  - *Type of Impairment*: Congenital total deafness
  - *Additional Impairments*: None
  - *Compensatory Tools*: Hearing aid from six months, inadequate for speech development
  - *Speech Treatment*: Visits speech therapist
  - *Speech Development*: None
  - *Overall*: Reacts to the loud sounds, does not react to his/her name, does not use his/her voice, makes inarticulate sounds
- **Child 2**
  - *Date of Birth*: 2015
  - *Occurrence of Hearing Impairment in Family*: Deaf parents
  - *Type of Impairment*: Congenital total deafness
  - *Additional Impairments*: None
  - *Compensatory Tools*: Has hearing aid, does not use it
  - *Speech Treatment*: Visits speech therapist
  - *Speech Development*: None
  - *Overall*: Reacts to the loud sounds, does not react to his/her name, does not use his/her voice, makes inarticulate sounds
- **Child 3**

- *Date of Birth*: 2014
- *Occurrence of Hearing Impairment in Family*: Deaf parents
- *Type of Impairment*: Congenital total deafness
- *Additional Impairments*: None
- *Compensatory Tools*: Has hearing aid, does not use it
- *Speech Treatment*: Visits speech therapist
- *Speech Development*: None
- *Overall*: Reacts to the loud sounds, does not react to his/her name, does not use his/her voice, makes inarticulate sounds

- **Child 4**
  - *Date of Birth*: 2013
  - *Occurrence of Hearing Impairment in Family*: Deaf parents
  - *Type of Impairment*: Congenital total deafness
  - *Additional Impairments*: None
  - *Compensatory Tools*: Has hearing aid, does not use it
  - *Speech Treatment*: Visits speech therapist
  - *Speech Development*: None
  - *Overall*: Reacts to the loud sounds, does not react to his/her name, does not use his/her voice, makes inarticulate sounds
- **Child 5**
  - *Date of Birth*: 2011
  - *Occurrence of Hearing Impairment in Family*: Mother without hearing problems, father unknown
  - *Type of Impairment*: Congenital total deafness
  - *Additional Impairments*: Cleft palate surgery
  - *Compensatory Tools*: One-sided cochlear implant on the right, after 5 years of having the implant is starting to react on sounds
  - *Speech Treatment*: Visits speech therapist
  - *Speech Development*: None
  - *Overall*: Last year unable to hearing, since September starting to understand words, able to pronounce some vowels and consonants (only with a help of a speech therapist – does not speak by itself)
- **Child 6**

- *Date of Birth*: 2011
- *Occurrence of Hearing Impairment in Family*: Parents and grandparents without hearing problems
- *Type of Impairment*: Congenital total deafness
- *Additional Impairments*: Childhood autism
- *Compensatory Tools*: One-sided cochlear implant on the right
- *Speech Treatment*: Visits speech therapist
- *Speech Development*: None
- *Overall*: Good use of hearing, able to repeat sounds and syllables, needs help from the speech therapist

- **Child 7**
  - *Date of Birth*: 2012
  - *Occurrence of Hearing Impairment in Family*: Parents and siblings without hearing problems
  - *Type of Impairment*: Congenital severe hearing defect
  - *Additional Impairments*: None
  - *Compensatory Tools*: Has hearing aid, can partially orient with hearing
  - *Speech Treatment*: Visits speech therapist
  - *Speech Development*: None
  - *Overall*: Prefers sign language to speaking, without hearing aid does not control his/her voice, has unnatural voice

- **Child 8**
  - *Date of Birth*: 2012
  - *Occurrence of Hearing Impairment in Family*: Parents and siblings without hearing problems
  - *Type of Impairment*: Congenital severe hearing defect
  - *Additional Impairments*: None
  - *Compensatory Tools*: Cochlear implant on both sides
  - *Speech Treatment*: Visits speech therapist
  - *Speech Development*: None
  - *Overall*: Does not react to sounds, does not speak, has unnatural voice, makes inarticulate sounds, struggles with vowel pronunciation

- **Child 9**
  - *Date of Birth*: 2017
  - *Occurrence of Hearing Impairment in Family*: Parents without hearing problems
  - *Type of Impairment*: Congenital total deafness
  - *Additional Impairments*: None
  - *Compensatory Tools*: Cochlear implant on both sides
  - *Speech Treatment*: Visits speech therapist
  - *Speech Development*: None
  - *Overall*: Reacts to the loud sounds, does not react to his/her name, does not use his/her voice, makes inarticulate sounds
- **Child 10**
  - *Date of Birth*: 2016
  - *Occurrence of Hearing Impairment in Family*: Parents without hearing problems
  - *Type of Impairment*: Moderate hearing defect
  - *Additional Impairments*: None
  - *Compensatory Tools*: Has hearing aid
  - *Speech Treatment*: Visits speech therapist
  - *Speech Development*: None
  - *Overall*: Reacts to the loud sounds, does not react to his/her name, does not use his/her voice, makes inarticulate sounds