ASSIGNMENT OF BACHELOR’S THESIS

Title:	Personal Finance Management Mobile App for iOS
Student:	Hoang Anh Ngo
Supervisor:	Ing. Marek Suchánek
Study Programme:	Informatics
Study Branch:	Web and Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of winter semester 2021/22

Instructions

The goal of the thesis is to design and implement a mobile app for iOS that would help users with personal finance and budget management. Overall work must compliant to the common software engineering practices:
- Review existing relevant applications, focus on features and UI/UX, and identify their advantages and disadvantages. Set functional and non-functional requirements for the app.
- Analyze possibilities in terms of technologies for iOS mobile app development, including recommended architectures and best practices.
- Design the app (both architecture and UI) based on the review and analysis using a software engineering approach.
- Implement and test the app according to the design.
- Compare the final app with other solutions. Describe distribution and further development possibilities.

References

Will be provided by the supervisor.

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

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

Prague March 31, 2020

Bachelor's thesis

Personal Finance Management Mobile App for iOS

Hoang Anh Ngo

Department of Software Engineering
Supervisor: Ing. Marek Suchánek

January 7, 2021

Acknowledgements

I would like to express my gratitude to the supervisor Ing. Marek Suchánek for his invaluable practical support during writing and refining of this thesis. I am deeply grateful to my family and friends for their presence and guidance through my studies. My special thanks are to my partner, for her immense mental support throughout the thesis development.

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 a school work under the provisions of Article 60 (1) of the Act.

In Prague on January 7, 2021 . . . .

Czech Technical University in Prague
Faculty of Information Technology
© 2021 Hoang Anh Ngo. 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

Ngo, Hoang Anh. *Personal Finance Management Mobile App for iOS*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2021.

Abstract

The bachelor thesis addresses an analysis, design, and implementation of a mobile application for finance management and budgeting on the iOS platform. The thesis analyzes other competitive applications with emphasis on their user interface, advantages and disadvantages. In particular, their main drawbacks, which this work solves, are described in more detail. The following chapters provide a description of the software engineering processes (design, implementation, and testing). The result of the thesis is a functional mobile application, thanks to which the users will be able to manage their finances and budgets easily. Realizing their financial habits, they will be ready to manage them better.

Keywords mobile application, iOS, management, finance, budget, MVVM, Swift

Abstrakt

Tato bakalářská práce se zabývá analýzou, návrhem a implementací mobilní aplikace pro platformu iOS pro správu financí a rozpočtů. V práci jsou analyzovány konkurenční aplikace s důrazem na jejich uživatelské rozhraní, výhody a nevýhody. Podrobněji jsou popsány především hlavní nedostatky konkurečních řešení, které výsledná aplikace řeší. Následující kapitoly pojednávají o procesech dle klasického softwarového inženýrství (návrh, implementace a testování). Výsledkem práce je funkční mobilní aplikace, díky níž budou uživatelé schopni jednoduše spravovat své finance a rozpočty, čímž zjistí své finanční zvyky a návyky a naučí se lépe hospodařit se svými penězi.

Klíčová slova mobilní aplikace, iOS, správa, finance, rozpočet, MVVM, Swift

Contents

Introduction **1**

1 Goals and Methodology **3**

2 State of the Art **5**
2.1 Spendee . . . 6
2.2 Wallet . . . 8
2.3 Pocket Expense 6 . . . 10
2.4 Summary . . . 12

3 Requirements Analysis **13**
3.1 Functional Requirements . . . 13
3.2 Non-functional Requirements . . . 14
3.3 Use Cases . . . 15

4 Design **21**
4.1 iOS Development . . . 21
4.1.1 Objective-C and Swift . . . 21
4.1.2 UIKit and SwiftUI . . . 22
4.1.3 Architectural Patterns . . . 22
4.1.4 Supported OS . . . 24
4.2 Application's Domain . . . 24
4.3 User Interface . . . 25
4.3.1 iOS Human Interface Guidelines . . . 25
4.3.2 Nielsen's Design Heuristics . . . 28
4.3.3 Scenes Design . . . 28

5 Implementation **35**
5.1 Firebase Backend . . . 35
5.1.1 Firebase Authentication . . . 35
5.1.2 Cloud Firestore . . . 36
5.1.3 Cloud Functions . . . 38
5.2 iOS Application . . . 38
5.2.1 Reactive Programming . . . 39

5.2.2 Clean Architecture and MVVM 40
5.2.3 Dependency Management 43
5.2.4 Navigation 45
5.2.5 UI Components 45

6 Testing **49**
6.1 Unit Tests 49
6.2 UI Tests 51
6.3 Usability Tests 51
6.3.1 Test Scenarios 52
6.3.2 Results 53

7 Results and Future Development **55**
7.1 Results 55
7.2 Future Development 55
7.3 Distribution 56

Conclusion **59**

Bibliography **61**

A Acronyms **67**

B Contents of Enclosed SD Card **69**

List of Figures

2.1 *Spendee* app preview . . . 6
2.2 *Wallet* app preview . . . 8
2.3 *Pocket Expense* app preview . . . 10

4.1 Interaction of *MVC* components according to [29] . . . 23
4.2 Interaction of *MVVM* components according to [34] . . . 24
4.3 Application domain's conceptual model . . . 25
4.4 Hierarchical and flat navigation style combined based on styles mentioned in [10] . . . 26
4.5 Modality presentation types [10] . . . 27
4.6 Core UI components used in the app [10] . . . 28
4.7 Authentication scene navigation flow . . . 29
4.8 Dashboard scene navigation flow . . . 30
4.9 Insight scene navigation flow . . . 30
4.10 Create Form scene navigation flow . . . 31
4.11 Budgets scene navigation flow . . . 31
4.12 Profile scene navigation flow . . . 31
4.13 Main scenes UI . . . 32
4.14 Additional screens UI . . . 33

5.1 *Cloud Firestore* NoSQL data model [46] . . . 37
5.2 Data binding diagram example . . . 39
5.3 *Clean Architecture* scheme [53, 54] . . . 40
5.4 Main layers of the app based on [54] . . . 41
5.5 Data flow diagram for creating a wallet based on [54] . . . 43

List of Tables

2.1 Features comparison 11

3.1 Requirements coverage 20

List of Listings

5.1 An authentication example using *Firebase Authentication* . . . 36
5.2 Persisting data on *Cloud Firestore* 37
5.3 An example of persisted transaction in JSON 37
5.4 A *Cloud Functions* example in TypeScript 38
5.5 Data binding code example 39
5.6 Domain layer components example implementation 41
5.7 Data Layer components example implementation 42
5.8 Presentation Layer components example implementation 42
5.9 *Dependency Injection* example implementation 44
5.10 Coordinator Pattern using XCoordinator [59] library 46
5.11 Creating and laying out UI components example 47
6.1 A unit test example 50
6.2 A UI test example 51

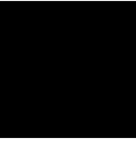
Introduction

The first smartphone appearance was in the late 90s; however, it was no later than in 2007, when Apple has introduced its first iPhone, that smartphones have gained on popularity [1]. In today's digital world, a smartphone has become one of the necessities in daily life thanks to its well-tailored features. Owing to this technological jewelry, modern technology has permeated mobile applications, affected, and advanced every aspect of our lives. [2] Smartphones solve various day-to-day problems using a great number of applications. One of the problems we might have or encounter in the future is keeping finances under control.

Even though being organized is not an inborn skill, spending money on nonessentials is. As a consequence, it is simple to lose track of expenses and end up living from hand to mouth. With budgeting, we can prioritize spending, manage finances better, or build new habits. Therefore, it allows us to reach our long-term goals while maintaining financial health—whence the motivation of creating an application further presented in the thesis.

Numerous applications already solve the described problem; however, either they are inadvertently complicated for a regular end user, visually obsolete, or lacking apparent features. The following chapters describe the aim of the thesis and analyze existing solutions. Other chapters are devoted to the application's design, implementation, and testing. Finally, distribution and further development possibilities are specified.

CHAPTER 1



Goals and Methodology

The main objective of this bachelor thesis is to develop a mobile application for iOS devices. The application will allow users to track, manage finances, and create budgets, which will be of substantial benefit to their financial health. Users will be able to take full control over their expenses and incomes, using the application's provided features.

Another goal is to address existing solutions that are popular in the mobile market. An analysis will be conducted on their primary assets and drawbacks, based on which the application's functional as well as non-functional requirements will be identified. Regardless of the recency of the finance tracking applications, the author strongly believes that tastes differ. For this reason, there will always be room for improvement and to target a different group of users.

Furthermore, to support the thesis' goal, possible technologies and best practices in iOS mobile development will be discussed. Last but not least, the application's implementation will conform to the discussed subjects and Software Engineering development processes as much as possible, with the consideration of the author's experience and knowledge.

The application will be tested using various testing methods that the author found most suitable for the solution. Finally, future development and distribution will be described.

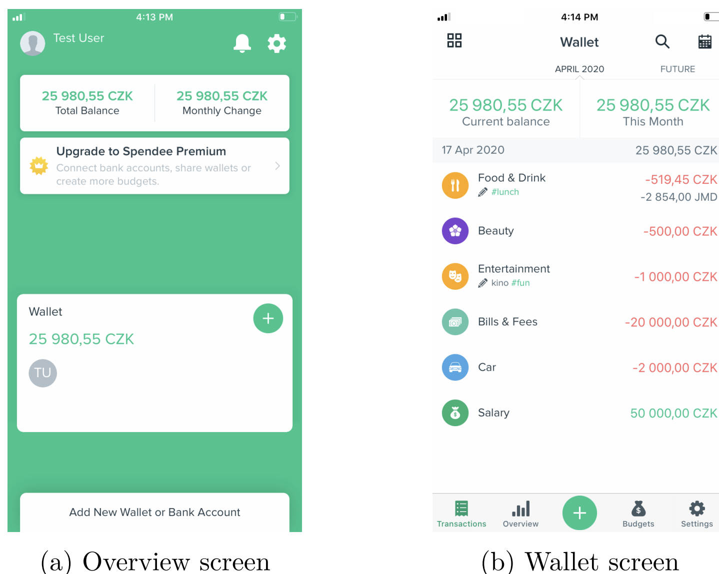
CHAPTER 2

State of the Art

Despite the fact that the worldwide App Store [3] market is congested with mobile applications (apps) for finance management and budgeting, not every solution is perfect or available to every user. Such solutions are, for the most part, impractically sophisticated. Other solutions, on the other hand, offer few primary features; however, the essential ones are locked out from users by a paid subscription plan. Whichever the case, it might discourage users from using such apps.

Hence, the following chapter analyzes the Czech App Store's current solutions, i.e., *Spendee*, *Wallet*, *Pocket Expense 6*. Each app was chosen based on its relevance, ratings, or popularity. It is no surprise that they have several features in common. Therefore, the aim is to mainly focus on the below listed fundamental features, which the current solutions have in common, and identify their advantages and disadvantages. A conclusion and comparison of features of all analyzed apps (Table 2.1) is available at the end of the chapter. In addition, in this chapter, wallets and accounts, or transactions and records are used more or less interchangeably since the terms are referred to differently in the analyzed apps.

- User Profile
- Wallets
- Budgets
- Categorization
- Filter and Search
- User Interface (UI) / User Experience (UX)



(a) Overview screen

(b) Wallet screen

Figure 2.1: *Spendee* app preview

2.1 Spendee

Spendee [4] is a personal multi-platform budgeting app, which is available on the web, iOS, and Android Operating System (OS). Originating from the Czech Republic, and with over three million downloads, it is also one of the country's most popular budgeting apps. Apart from the local mobile market, it is also available in more than 150 countries. This section addresses the iOS version only. [5]

Features

The app alone is free; however, its main features (as stated in the app's description [4]) are noticeably limited by a subscription plan users have—free or premium. Some of them are discussed in the paragraphs below.

User profile While some apps do not require having an account, *Spendee* does. It is due to a sharing feature, which allows users to share their wallets with others. For that, users are forced to create a profile using either an e-mail, Facebook, or Google account. Users then are able to change their first name, last name, or add their birthdate.

Nonetheless, it is convenient that the registration is swift and does not involve e-mail address verification. On the flip side, the app immediately shows a pop-up dialog after finishing the registration offering its premium subscription plan, which is an awful way to state that the app's functionalities are beforehand limited.

Wallets A wallet represents a place, i.e., a bank, a saving, or a cash account, where users track their transactions. The feature simulates the real-world scenario, in which each user has a various number of accounts. As a result, users are able to track various types of finances separately.

While premium users have an unlimited number of wallets, a free user is only provided with one single wallet. Even though it is the only drawback, it has, at the same time, devalued its only advantage. Without multiple wallets, it is just a bucket of bills crowded in one place.

Budgets This feature sets a certain limitation on the user's wallet for one or a few categories. It then handles and shows all transactions in given categories and displays a percentage of money spent, which provides users with some understanding of their financial habits and might help them spend less, thus save more. Similar to the previous feature, the ability to create multiple budgets is only accessible to premium users.

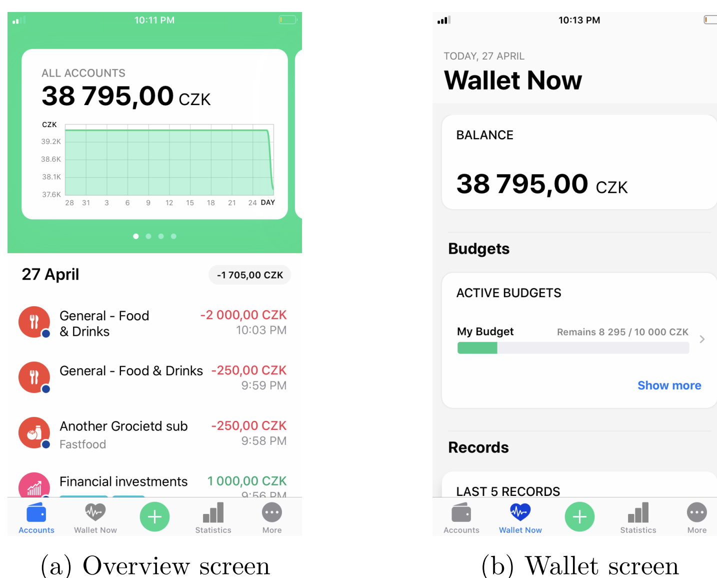
Categorization This feature allows users to maintain their finances organized. Users can categorize their transactions either by categories, notes, or hashtags[1]. The app comes with a pre-defined list of categories, which users may easily change or add. What is more, it is possible to create as many of them as needed. Later in the app, each transaction may be categorized.

Filter and search The app only supports searching, which outputs transactions matching a queried phrase—yet only those that are categorized. The result unfolds depending on what was searched.

UI/UX

The app's user interface is modern, clean, and yet very simple. Its UI colors depend on the user's mobile color theme, i.e., light or dark. After launching the app, users are presented with a wallet's primary screen, which contains a list of transactions, where they can quickly check the current balance and expenses, see Figure 2.1b. In-app navigation is intuitive. To navigate through screens such as a list of transactions, a wallet's overview, or settings, one can use a tab bar situated at the bottom of the mobile screen. Even though a few bugs were found during the analysis, the overall impression matches the app's rating.

[1] Hashtag is a metadata tag often used on social sites [6].



(a) Overview screen (b) Wallet screen

Figure 2.2: *Wallet* app preview

2.2 Wallet

Wallet [7], or *WalletApp*, is another favored personal finance management app from a Czech startup, Budgetbakers [8]. Apart from three million downloads, an excellent rating of 4.7 out of 5 stars on the App Store, and features described below, it is worth mentioning that the Budgetbakers boasts the newest functionality, which, at the time of writing the thesis, was being tested in the beta versions of the app. This functionality allows users to make actual finance transactions within the app, making *Wallet* the first-ever budgeting app providing such a feature. [9]

Features

In contrast to *Spendee*, *Wallet* is equipped with a greater number of (free) features and statistic tools, which would be a perfect match for demanding users, yet, on the contrary, complicated for simple or new users in a budgeting app. Conceivably, not all of them are free of charge. Besides, what is worth mentioning is the app's gaming feature that tracks the user's satisfaction with their records.

User profile Much the same as the previous app, users can sign-up via the same means as well as are not required to verify the e-mail address. Following the sign-up, the app offers an introductory account base currency and an initial balance setting. Users can also personalize their profile by filling out additional information, adjust their data and pri-

vacy consents (geolocation tracking, advertisement e-mails, or customer segmentation), and even delete the profile.

Wallets The app provides users with up to three accounts; any other amount is unfortunately available for premium users only. Regardless, account creation is straightforward and accompanies users with relevant account settings, such as a current balance, type of account, currency, color, or a possibility to exclude the account from overall statistics. Such a process is a feature *Spendee* does not offer even with its only wallet.

Budgets Surprisingly this feature is limitless, which means there is no end to creating and tracking budgets in the app. The feature clearly shows how much money users spent and the amount left in a specific budget. Each of the budgets then stores related records, which are only at the paid users' disposal, with detailed statistical information.

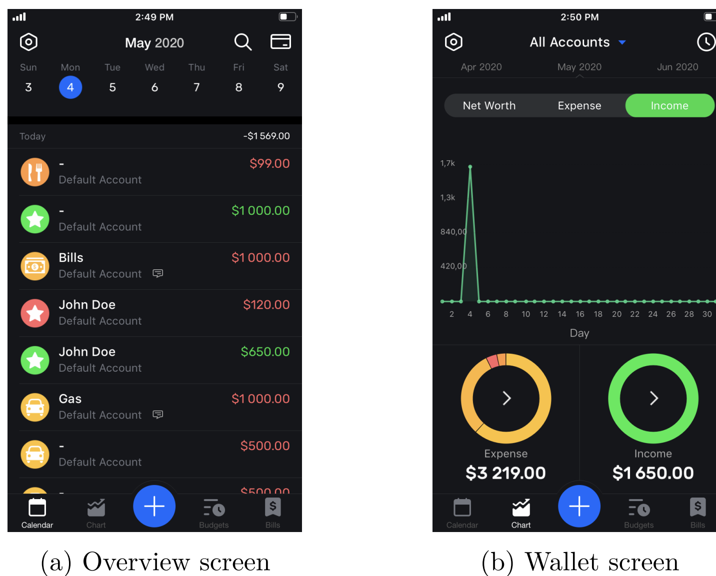
Categorization There are two means of categorization—categories and labels. Categories have more or less the same purpose as the previous app and differ in their customization. There are eleven categories prepared in advance, which can only be edited (users cannot create nor delete them). Additionally, each category includes several subcategories, which are only editable as well. Nonetheless, it is possible to add other subcategories.

A label is a characterizing word that is given to an object. In essence, it should be a self-explanatory word/phrase that describes the object. In the app, labels carry the exact purpose. For clarity, users are able to differentiate them by defining colors. Simple as it is, colors are another paid feature.

Filter and search The app meticulously tracks the user's records and displays them detailly in statistic graphs. Users then may alter their content by filtering. Primary criteria on the filter are accounts and a period that it ought to be applied. Furthermore, users can set custom filters, for instance, categories, labels, currencies, and many more.

UI/UX

The whole app is made in a basic UI that conforms to the design standards [10] of iOS development, which allows for natural navigation and usage of the app. The main downside is the fact that some screens are overloaded with information, which might seem, to a certain extent, chaotic. Another aspect that is displeasing to the user's experience is the stacking of screens on one another. A real-life example would be accessing a bottom card in a stack of cards. To get that card, one would need to remove all of the above cards. A preview of the app is shown in Figure 2.2.



(a) Overview screen (b) Wallet screen

Figure 2.3: *Pocket Expense* app preview

2.3 Pocket Expense 6

Pocket Expense [11] is in comparison with the previous apps less popular, and yet offers equally competitive design and services, as shown in Figure 2.3. The app is preferably suited for users who do not expect sophisticated insights or graphs and make do with fundamental budgeting tools.

Features

Unlike the previous apps, *Pocket Expense* has a little bit of every analyzed feature. Although it seems to be perfect at first sight, the features share many both positive and negative aspects.

User profile As opposed to the previous app, *Pocket Expense* only provides a sign-up using an e-mail. However, it is absorbing that it also gives users an option to sign-up "password-less", which requires e-mail verification. Unfortunately, due to an application-side error, this feature could not be tested. Regarding the profile itself, it serves as a way of exporting or restoring the user's data (while the export feature is paid) as there is no further profile personalization except for changing a profile's photo.

Wallets As a free user, there is only one account available with no initial setup—a premium profile is required to create multiple accounts with user customization. The account is by default set with US currency, which seems to be bound to the profile, not an account.

Budgets Users of the app are predestined to save money utilizing a single budget with one non-changeable category once the budget is created. The app then displays the amount left in the budget as users add their expenses in the tracked category tied to actual expenditure.

Categorization Users can organize their expenses in categories, of which they may have an unlimited number. Similarly to the above apps, each tracked transaction is tied to one category, or in this case, split to multiple categories, for instance, a payment regarding two separate items.

Filter and search The filter feature changes the content of insights of either all accounts or one specific account. With the filter usage, users can display transaction statistics from a chosen filtered period, such as a week, month, year, or custom time.

The search is an elaborate and fast feature that finds transactions satisfying the search phrase, which can be a category, note, payee, or amount. Furthermore, it is possible to search for related transactions of the same category within the list of transactions, see Figure 2.3a.

UI/UX

Despite being unpopular, the app is styled with a trending UI, unambiguous elements, and icons, making the app accessible even for users without any previous experience with this type of application. The app presents a list of transactions on launch that appears to be showing everything the user needs; however, elements are missing in the term of UX. It is ambiguous on what category the expenditure was spent. The app also provides an option to create a transaction with split categories, which means splitting the transaction's amount into several categories. Such transactions are displayed ambiguously as well. Above that, the app has video advertisements, which cannot be dismissed until users have watched them.

Feature \ App	Spendee	Wallet	Pocket Expense 6
User Profile	✓	✓	✓
Unlimited wallets			
Unlimited budgets		✓	
Categorization	✓	✓	✓
Filter		✓	✓
Search	✓		✓

Table 2.1: Features comparison

2.4 Summary

The purpose of this chapter was to analyze the relevant existing solutions based on criteria/features that were regarded as essential for budgeting applications and describe their advantages and disadvantages. The following Table 2.1 shows a comparison of the provided features of analyzed apps.

The outcome of the analysis results in two main supportable conclusions. Firstly, each app allows users to create a user profile, which stores their data for possible backup, and secondly comes with monetization that limits the fundamental features of the app. It is crucial to provide users with as many of those features as possible, and potentially monetize only additional features, for instance, detailed insights or wallet sharing. Therefore, the future design and implementation of the thesis will be conducted in regards to the mentioned conclusions.

CHAPTER 3

Requirements Analysis

Requirements analysis is an integral process and critical to the success of software development that must be exhibited to constrain an environment in which the app will exist [12, p. 2–1]. Therefore, this chapter's objective is to provide a definition of such requirements that are generally of two types: functional and non-functional. Both requirements were set on the ground of the analysis in the previous Chapter 2 and are essential to defining the app's functionalities and scope. What is more, the chapter describes basic use cases that are derived from the functional requirements.

3.1 Functional Requirements

This section introduces essential functional requirements of the thesis app, which describe functions or components the app must offer as well as their intended behavior—sometimes known as capabilities [12, p. 2–2].

F1 User account Each user must have an account to use the app, which can be created during the registration process. After registration or sign-in with an existing account, the user will not be required to do so again. The signed-in user session will be active until the user does not explicitly request to sign out.

F2 Wallets Users will be able to create an unlimited number of wallets. As the name suggests, a wallet will have a current balance and provide additional information such as the sum of the current month's expenses and incomes and a list of transactions that were made in the wallet.

F3 Budgets Users will be able to set up an unlimited number of budgets. Each budget must specify a budgeting amount and list of categories that are to be considered within the budget (budgeted categories).

F4 Transactions Users will be able to create, edit, or delete transactions at any point. Each transaction is bound to a wallet and must contain an amount and category. Apart from these attributes, it may include a title or a note for better categorization.

F5 Categories The app will provide users with a small number of predefined categories; however, the users will also be able to create categories of their own at any time. The main purpose of categories will be for transaction categorization.

F6 List of transactions The app will have a list of transactions that will contain records of one or all wallets. Each such list will only display the ones inside the range of a specific period—i.e., an active filter.

F7 Filtering Users will be capable of filtering their transactions in the app by a period of time to gain a better grasp of their finances.

F8 Sorting Users will be capable of sorting their transactions by creation date, category, or amount.

F9 Searching Users will be able to search for their transactions by title, description, amount, or category; however, only within the active filter (a period of time).

F10 Insight The app will offer visualized statistical insights of the user's finances for one or all wallets (wallet's spendings over time, incomes and expenses, and the most spent categories). Within the insight, users will be able to filter various periods and manage the filtered transactions.

F11 Localization The app will be localized in two languages—English and Czech.

3.2 Non-functional Requirements

Apart from the functional requirements (see Section 3.1) that specify an app's behaviors, non-functional requirements specify criteria, constraints, and qualities of an app, which means elaborating various characteristics: performance, security, compatibility, and so on [12, 13]. These characteristics will have a significant impact on architecture design.

N1 iOS App The app will be implemented for iOS devices supporting the operating system iOS 13 and newer.

N2 Security The app will provide users with a secure way to authenticate so that their personal data will be safe.

N3 Clean UI/UX The app will wear a simple and cleanly tailored design by using recommended guidelines.

N4 Data persistence The app will store user data on an online server.

N5 Testability The app will be implemented in such a manner that the app's core functionalities will be easy to test. What is more, those functionalities must be covered by appropriate tests.

N6 Scalability The app must be implemented so that the app can be easily scaled in future development.

3.3 Use Cases

This section describes the necessary use cases that should clarify the behaviors of functional requirements—defined in Section 3.1—under various conditions. It is basically a conversation between the user, called the *primary actor*, and the *system*. The user interacts with the system that responds accordingly [14]. Understanding the use cases significantly facilitates the UI design discussed in Section 4.3 and gives well-laid documentation for acceptance tests in Chapter 6. To check the coverage of requirements see Table 3.1.

Additional information

- For all use cases below, the primary actor is *User*.
- Notation *<UC6 Display current transactions>* means that the use case uses the same flow the use case in the notation.

UC1 Sign-up

The user must have an Internet connection and an existing e-mail address, which has not been registered in the system, to create an account successfully. If these conditions are met, the system will create a new account under the given credentials and automatically sign the user in.

- **Pre-condition:** System is connected to an Internet connection; User has an e-mail address.
- **Post-condition:** System will have created a new account with the given credentials and an initial wallet.
- **Flow:**
  1. User selects to sign up.
  2. System presents a sign-up screen.
  3. User provides required credentials.

4. System processes the credentials.
5. System creates a new account.
6. System signs User in.
7. System presents a wallet creation screen, in which User can set up an initial wallet.

- **Extensions:**

4a. System fails to process the credentials.

 4a1. System reports the failure to User.

 4a2. User either cancels the flow or tries again.

4b. User provides an already registered e-mail.

 4b1. System notifies User.

UC2 Sign-in

To sign in, the user must have an existing account and an Internet connection to authenticate his credentials. Given that valid credentials were provided, the system will sign the user in. In case the user was already signed in, the whole flow is skipped and will continue to the system's main screen.

- **Pre-condition:** System is connected to an Internet connection; User has an account.
- **Post-condition:** User will be signed in.
- **Flow:**

1. User selects to sign in.
2. System presents a sign-in screen.
3. User provides required credentials.
4. System processes the credentials.
5. System signs User in.

- **Extensions:**

1a. User is already signed in.

 1a1. User proceeds to the main screen.

4a. System fails to process the credentials.

 4a1. System reports the failure to User.

 4a2. User either cancels the flow or tries again.

UC3 Create a wallet

When creating a wallet, the user must provide the wallet's necessary information, such as the wallet's name and initial balance.

- **Pre-condition:** User is signed in.
- **Post-condition:** System will have created a new wallet.
- **Flow:**

  1. User selects to create a wallet.
  2. System presents a wallet creation screen.
  3. User provides required information.
  4. User confirms the creation process.
  5. System creates the wallet.

UC4 Create a transaction

The user must have at least one wallet to create a transaction that will be bound to it.

- **Pre-condition:** User as at least one wallet.
- **Post-condition:** System will have created the transaction.
- **Flow:**

  1. User selects to create a transaction.
  2. System presents a transaction creation screen.
  3. User fills in the necessary information about the transaction, including a category and a wallet to which the transaction will be bound.
  4. User confirms the creation process.
  5. System creates the transaction.

UC5 Display current transactions

- **Flow:**

  1. User selects a wallet or all wallets for which to display the transactions.
  2. System presents the selected wallet(s).
  3. System presents the list of transactions.

UC6 Display transaction detail and edit it

The user will be able to edit any existing transaction if he has at least one.

- **Pre-condition:** User has at least one transaction.
- **Post-condition:** System will have changed and stored changes of the transaction.
- **Flow:**
  1. *<UC5 Display current transactions>*
  2. User selects a transaction to edit.
  3. System presents a transaction detail screen.
  4. User selects to edit the transaction.
  5. System presents a transaction edit screen.
  6. User edits the transaction as needed.
  7. User confirms the changes.
  8. System stores the changes.

UC7 Filter out transactions

Given a list of transactions, the user will be able to filter out transactions based on a given criterium, which will leave only transactions that satisfy it.

- **Flow:**
  1. *<UC5 Display current transactions>*
  2. User selects to filter transactions.
  3. System presents a filtering form, in which User can set it up.
  4. User chooses to filter transactions by period “Month”.
  5. System alters the list of transactions leaving only the ones that were created in a given month.

UC8 Sort transactions

Given a list of transactions, the user will be able to sort transactions based on creation date, amount, title, or category.

- **Flow:**
  1. *<UC5 Display current transactions>*
  2. User selects to sort transactions.
  3. System presents a sorting form, in which User can set it up.

4. User chooses to sort transations by category.
5. System alters the list of transactions so that they are ascendingly/descendingly sorted by category.

UC9 Search transactions

Given a list of transactions presented by the system based on the active filter, the user will be able to search transactions within it. The search will depend on a searched keyword—that is, date created, title, note, category, or amount.

- **Flow:**
    1. *<UC5 Display current transactions>*
    2. User selects to search transactions.
    3. User enters a keyword "Gro".
    4. System presents a list of transactions that contain the searched keyword.
- **Extensions:**
    4a. System cannot find any transaction satisfying the given keyword.
        4a1. System presents an empty list of transactions.

UC10 Edit transaction after searching

At any time anywhere transactions are available (through a list of transactions), the user will be able to select a transaction to perform an action—for instance, editing.

- **Pre-condition:** User has at least one transaction.
- **Post-condition:** System will have changed and stored changes of the transaction.
- **Flow:**
    1. *<UC9 Search transactions>*
    2. *<UC6 Display transaction detail and edit it starting from step two>*

UC11 Display wallet's insights

- **Pre-condition:** User is signed in; User has at least one wallet.
- **Flow:**
    1. User selects to view insight of a wallet.

2. User either selects a specific wallet or all wallets to view the insight for.
3. System presents an insight screen that includes statistical information for the selected wallet within a certain period of time.
4. User performs desired actions, such as filtering or editing available transactions.

UC12 Change app’s language

- **Post-condition:** System will have changed its language from Czech to English, and vice versa.
- **Flow:**
  1. User selects to change the app’s language from the device settings.
  2. User changes the system language from Czech to English.
  3. User confirms the change.
  4. System changes the app’s language accordingly to the set language in the device.

UC13 Create a transaction that affects a budget

- **Pre-condition:** User has an active budget for “Groceries” category.
- **Post-condition:** System will have created a transaction; System will have changed the budget’s balance accordingly.
- **Flow:**
  1. *<UC4 Create a transaction> with “Groceries” category*
  2. System increases the budget’s spent amount by the given amount.

Use Case \ Requirement	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11
UC1	✓	✓									
UC2	✓										
UC3		✓									
UC4				✓							
UC5						✓	✓				
UC6				✓		✓	✓				
UC7							✓				
UC8					✓	✓		✓			
UC9						✓			✓		
UC10				✓		✓			✓		
UC11							✓			✓	
UC12											✓
UC13			✓	✓							

Table 3.1: Requirements coverage

CHAPTER 4

Design

The following chapter conceptualizes the work's solution and forms a detailed architecture meeting software requirements mentioned in Chapter 3. Encompasses the solution design that presents iOS development fundamentals, the domain model of the work, final screen prototypes, and creates a clear structure for the upcoming stage of the software development process.

4.1 iOS Development

This section provides an insight into the iOS development environment. However, the aim is not to describe aspects of the environment in detail yet instead give brief information on key parts of iOS development used in this work later on.

4.1.1 Objective-C and Swift

Objective-C is an object-oriented programming language used by Apple since the early 1980s that inherits the advantages from the first programming languages C and Smalltalk [15]. Since *Objective-C* is over 30 years old, it became harder for new developers to learn and understand as the language aged. It was not until June of 2014, during the annual Worldwide Developer Conference (WWDC) event, that Apple unveiled a new programming language called *Swift* [16].

Despite both languages being native languages to code Apple products and have corresponding App Programming Interfaces (API) and frameworks (Cocoa, Cocoa Touch), they are not alike. *Swift* offers more powerful tools and modern coding solutions for safer, faster, and more expressive code. Thanks to a simpler syntax, *Swift* is more human-friendly and has a fast learning curve,

making it more suitable for new iOS developers [16, 17]. That said, it was chosen as the programming language for the following development.

4.1.2 UIKit and SwiftUI

The current development on iOS provides developers with two design frameworks—*UIKit* and *SwiftUI*. Both frameworks allow developers to design their app's interfaces from scratch using the provided toolkit.

UIKit It is an event-driven framework built on imperative programming paradigms [18, 19] that has been around for more than ten years [20], making it as robust as possible. *UIKit* has gained comprehensive support over the years and provides a broad API coverage that supplies developers with various library adaptations [21] to develop their apps on the iOS platform.

SwiftUI This framework, released in 2019 at WWDC [22], brings developers a new modern way to design their app's interfaces not only on the iOS platform but also on all Apple platforms (macOS, tvOS, and watchOS) [20, 23]. It uses declarative syntax, which might require changing one's perspective when transitioning from *UIKit*. Since *SwiftUI* is a relatively young framework that is still evolving, it is unsurprising that it is not feature-complete, which means it does not offer advanced UI components that might be in other frameworks. Despite that, some believe that *SwiftUI* is to replace the *UIKit* in years to come [21].

Given that *SwiftUI* does not provide broad API coverage as *UIKit*, is not detaily supported and well-documented, the upcoming development phase implements the work using the first framework. Regardless of what framework is chosen, the implementation in Chapter 5 provides a generic solution for both frameworks.

4.1.3 Architectural Patterns

If the decades-long software development has proven anything to be practical, it is an architectural pattern. It has been giving developers solution blocks that deliver effective solutions to structure the data flow, data storage, control flow, and many more in any application [24, 25]. Therefore, it is vital at this early stage of development to decide what architecture the app will be built in to constrain the app's quality attributes and meet the non-functional requirements in Section 3.2.

Model-View-Controller

Model-View-Controller (MVC) is one of the widely used architecture patterns in iOS development [26]. It has been around since the early days of Smalltalk

and is recommended by Apple that builds its frameworks on the same architecture pattern [27, 28].

It consists of three main components—i.e., *model*, *view*, and *controller* objects. *Model* objects encapsulate the application's domain data and should not be explicitly connected to *view* objects that present them. The *view* objects' purpose is to display the *model's* data and communicate user interactions to *controller* objects. Thus the *controller* objects are mediators between *models* and *views*, handle the application business logic, and manage other objects' life cycle [29]. For better visualization, see Figure 4.1.

Even though the *MVC* pattern is relatively simple in theory, and works perfectly for smaller-sized projects [30], in practice, *view* and *controller* objects tend to become much more tightly coupled [31], which breaks the pattern and leads to a so-called *Massive View Controller* [30]. As the project grows in size, the *controller's* responsibility gradually increases and makes it handle responsibilities that it ought not, for instance, navigating or networking. Such structure leads to an error-prone and hard to test codebase [30].

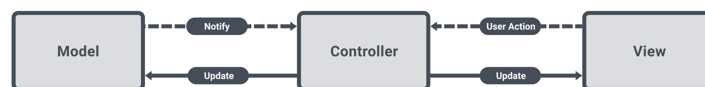


Figure 4.1: Interaction of *MVC* components according to [29]

Model-View-ViewModel

Model-View-ViewModel (MVVM) is another popular pattern [26], which is very similar to *MVC*. Both of them assert the concept of Separation of Concerns (SoC). Nevertheless, there are two main differences.

Firstly, it combines *controller* and *view* objects into one *view* object with minimal responsibility compared to *MVC controller* objects [32]. Its primary role is to manage UI components, handling user interactions, and binding user data to *a view model* (called data binding, which is explained in Subsection 5.2.1). As a consequence, the pattern avoids *Massive View Controller*.

Secondly, it introduces a new object called *view model* specifically designed for a view object responsible for handling any presentational logic, such as transforming data into a human-readable form, which was initially the *controller's* responsibility in *MVC* [33]. Thus it plays a similar role as the controller object; however, it separates concerns more.

View objects no longer own *model* objects but directly ask *view model* objects that they own for the data necessary to update the UI. Such SoC reduces the complexity of the *view* (controller) objects as the business logic is moved out from it. More importantly, it increases the code's testability as the business logic does not depend on the view implementations. [34] Moreover, as shown in Figure 4.2, the data flow is unidirectional, which helps any codebase to be scalable in the future [35].

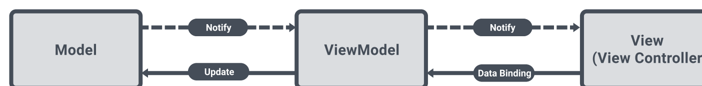


Figure 4.2: Interaction of *MVVM* components according to [34]

4.1.4 Supported OS

When developing an iOS application, it is important to decide which minimum iOS version the app will support. This decision will impact not only the app's availability but also the accessibility of the newest OS API. A "current minus one" rule is often applied, which narrows the versions down to the latest and previous iOS version. [36]

At the beginning of 2020, iOS13—the newest iOS version at the time—usage was around 70% while iOS12 at round 23% [37]; however, with the release of iOS14 in September 2020 [38], iOS13's usage has currently decreased down to a mere 18% of all iPhone devices, making iOS14 the most used iOS with 72% usage [39]. Given the statistics, the vast majority of iPhone users update their devices to the newest iOS each year; thereby, iOS13 will be the minimum supported version.

4.2 Application's Domain

A domain model helps to identify the relevant concepts and ideas of a domain. It decomposes the domain into individual conceptual classes or objects—that reflect the real world—and visualizes the associations between them using Unified Modeling Language (UML) notation. [40] See Figure 4.3.

The central object of the domain is a transaction that partakes in every event in the app. For a transaction to exist, it has to belong to a category and a wallet. While transactions are existentially dependent, categories and wallets are not. Users will be able to create such transactions and provide detailed attributes to characterize and categorize their spendings. During the transaction's lifetime, it can be regarded in one or more budgets if its category is budgeted. The domain identifies various types of categories, wallets, and filters. Moreover, the domain identifies four core scenes of the application:

- **Dashboard**: presents a list of transactions recorded in the current month that belong to an active wallet. The user can change the active wallet in the filter.
- **Insight**: shows insights of user's spendings within the selected filter.
- **Budgets**: users can create budgets for the current month for one or multiple categories.
- **Profile**: manages the user's information.

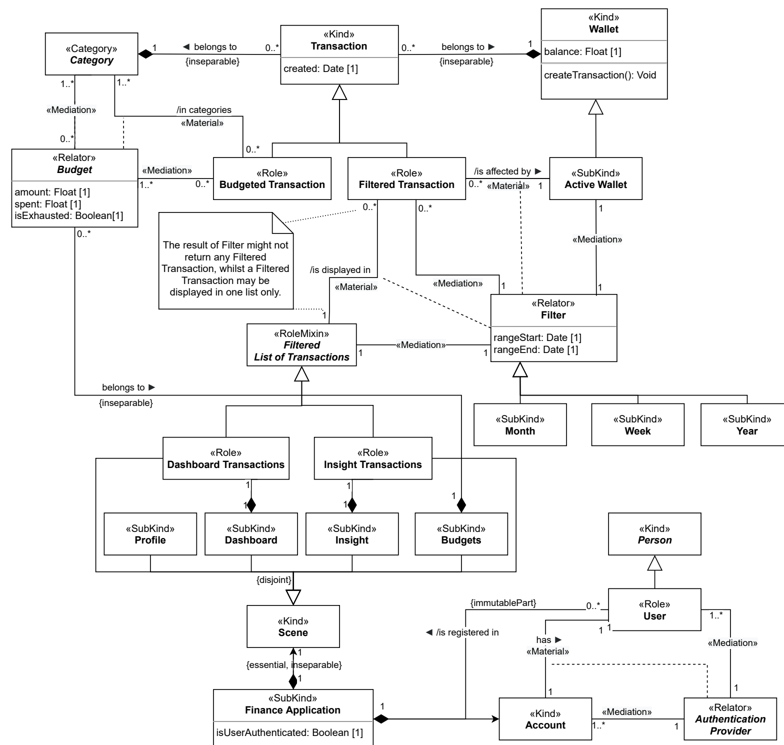


Figure 4.3: Application domain's conceptual model

4.3 User Interface

Designing the UI is only a small yet vital part of developing any software application [41]. Therefore, for the UI design to succeed, it is necessary to gain a clearer picture of the future users' needs. The design is created based on the previous existing solutions, requirements analysis, and the use cases mentioned in Chapter 3.

The following section focuses on the designing process that follows Apple's *iOS Human Interface Guidelines* [10] designing principles, as well as Jakob Nielsen's ten heuristics for interaction design [42]. However, the objective is not to follow the guidelines step by step—following them is not as straightforward as following a cooking recipe as they are purposely general to make them broadly applicable [43]—but instead, use them as a ruler to stay on the best practices track.

4.3.1 iOS Human Interface Guidelines

The *iOS Human Interface Guidelines* provide in-depth information and UI resources for achieving a consistent appearance across iOS applications. The

following are some of the UI components and techniques from the guideline that this work's app uses to meet the platform's standards [10]:

App Architecture

Launch screen A launch screen appears immediately after an app starts. Its purpose is not artistic, but to enhance the app's perception as quick to launch and ready for use. Every iOS app must have a launch screen.

Navigation Navigation is an important part of an app's design. The UI ought to be implemented so that the navigation feels natural and familiar. The app uses a combination of flat and hierarchical navigation style, which is the root of navigation flow.

On the top level, there are content categories (called "Scenes" in Section 4.3.3) in the form of a single-level tree that allows users to switch from one scene to another. Each lower level of flat navigation represents hierarchical navigation that lets users navigate in a one-directional way (down or up). An illustration of the combined navigation style is depicted in Figure 4.4; or in Section 4.3.3 that describes scene flow in more detail.

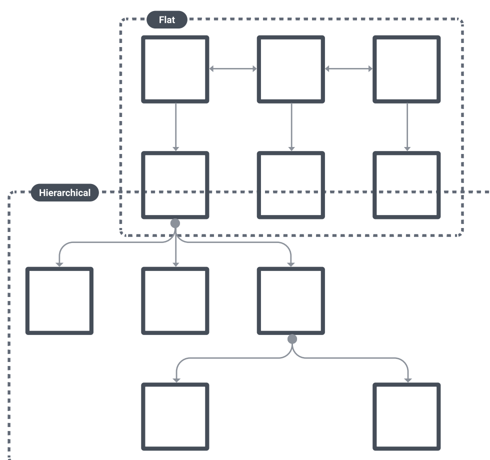
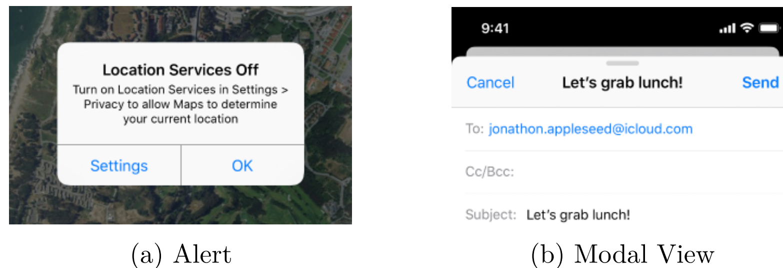


Figure 4.4: Hierarchical and flat navigation style combined based on styles mentioned in [10]

Modality Modality is a design technique that presents content—an alert or a modal view—in a temporary mode that is separate from the user's current context and requires an explicit action to exit. See Figure 4.6.

The app uses this technique to separate the primary navigation flow from secondary flows. For instance, displaying a budget's detail screen, as Section 4.3.3 describes, is still in the Budgets scene's main flow, while

the secondary flow is navigating to a transaction's detail screen from within the budget's detail screen.



(a) Alert (b) Modal View

Figure 4.5: Modality presentation types [10]

UI Components

Tab bars 4.6a A tab bar is used as a top-level component in the flat navigation style that has the ability to navigate between different scenes of an app swiftly. It appears at the bottom of an app screen and consists of buttons with a label and icon. The tab bar uses buttons strictly for navigation instead of performing actions.

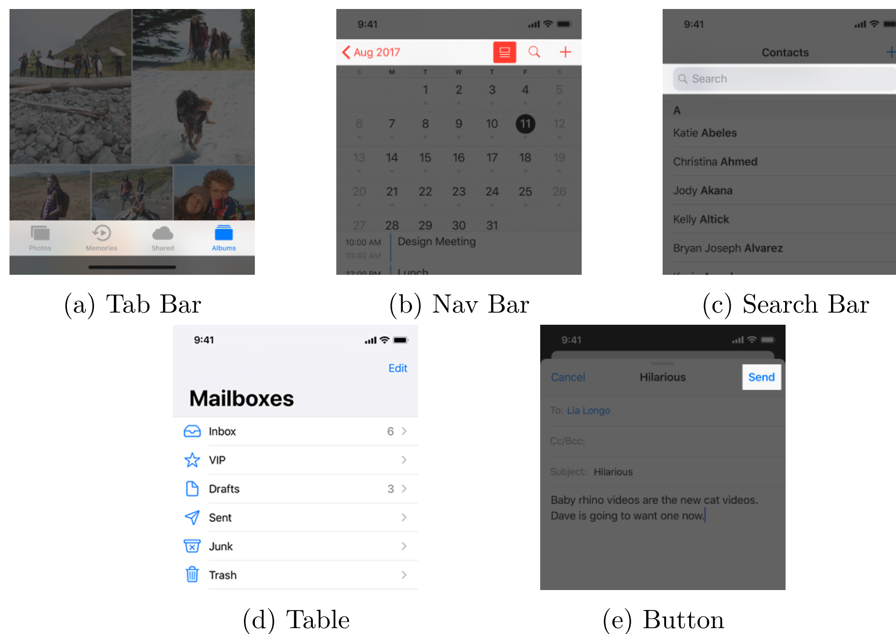
Navigation bars 4.6b A navigation bar allows users to navigate through a series of hierarchical screens by stacking them on one another. When this happens, the navigation bar displays a back button indicating possible navigation backward. It may also have additional control items for managing the active view.

Search bars 4.6c As the name suggests, a search bar allows users to type text into a field, which leads to searching for results in some collection of data. It should be noted that the search bar does not perform the actual searching—that the developer must implement himself—but instead provides native appearance and animations.

Tables 4.6d A table displays large or small amounts of information cleanly in a single-column list of rows. It can be used to present data or as an entry point to means of navigation.

Buttons 4.6e A button performs an app-specific action, for instance, to create or cancel editing a transaction, and appears either in navigation bars or at the bottom of the app screen.

As can be seen in Figure 4.13, owing to the *UIKit* framework all of the above mentioned components can be greatly customized to match the app's style.



(a) Tab Bar (b) Nav Bar (c) Search Bar

(d) Table (e) Button

Figure 4.6: Core UI components used in the app [10]

4.3.2 Nielsen's Design Heuristics

The ten general heuristics for interaction design by Jakob Nielsen is a usability engineering method for finding drawbacks of a designed UI to be addressed and fixed during the design process. [42]

Some ambiguities were resolved in time owing to the heuristics. For instance, one of the heuristics—the visibility of system status, which says the app should always keep users informed about its current status—has led to adding/fixing a missing empty state of table components. Therefore, the final screen presents an illustration with a message stating that the user does not have any data in the table, delivering much clearer experience.

4.3.3 Scenes Design

The analysis in the previous chapters led the UI design to break into six main scenes [2]—Dashboard, Insight, Create Form, Budgets, and Profile. The following sections describe each scene including their navigation flow illustration. All designs are visible in Figure 4.13.

[2]The work defines a scene as a group of multiple screens, which form one functional unit for which it makes sense to create a unified navigation flow.

Authentication

The scene is quite simple. It consists of two text fields to fill in credentials and "a confirm button" to perform the authentication action. An additional button is placed below this button to redirect users to a different authentication flow (see Figure 4.7), where Google Account can be used to authenticate. If the user does not have an account, he can switch to a sign-up screen to create one. The screens are almost identical except for an extra text field to enter the user's name. See Figure 4.13a.

The scene is error preventive (fifth heuristic of [42]), so the confirm button is initially disabled to prevent the user from triggering the action without providing the required credentials. Furthermore, the scene presents alerts in the event of unsuccessful authentication.

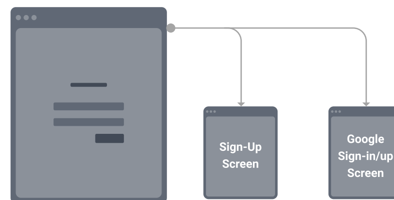


Figure 4.7: Authentication scene navigation flow

Dashboard

A Dashboard scene is pertained to display the most critical and valuable information in an app in one place. The analysis has identified three essential pieces of information to display to the users.

The scene displays an active wallet's [3] name, current balance, and cashflow indicating expenditures and earnings at the top of the app screen, as illustrated in Figure 4.13b. Under the wallet's name, users can see the current date. Beneath this section, users can find the current month's list of transactions belonging to the active wallet.

Moreover, the scene provides users with search and sort functionalities to find and organize their spendings by tapping on dedicated icons/labels; and are able to view a transaction's detail screen by tapping on the transaction or change the active wallet by tapping on the wallet's name. For the scene's navigation flow, see Figure 4.8.

Insight

The Insight scene provides users with a visual illustration of their wallet's spendings in some period, which the users may alter in provided filter func-

[3] An active wallet is either one or all of the user's wallets set to be displayed. In other words, it is a default wallet on which users can perform various actions the app provides.

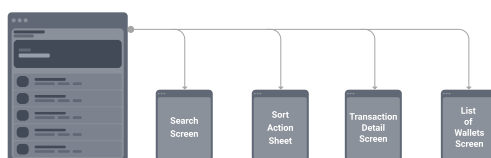


Figure 4.8: Dashboard scene navigation flow

tionality. The UI reflects the set filter with clear labels of the selected wallet's name, a textual description of the period, and a table of transactions. Further, the scene presents the spendings in the form of a bar and pie chart—one visible at a time—each showing different sets of data. The bar chart shows spendings aggregated by time, while the pie chart shows data by four most spent categories. Within the Insight, users are able to view a transaction's detail screen as well.

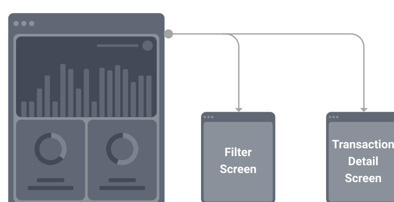


Figure 4.9: Insight scene navigation flow

Create Form

The *F4 Transactions* requirement defined in Section 3.1 states a transaction must be createable at any point. To meet the requirement, the creation of transactions cannot be within some scene; instead, it has to be a scene itself.

Besides, adding new transactions will probably be the most performed action, which adds to the importance of having this functionality easily accessible. As presented in Figure 4.13b, the scene's button is designed as glowing to diverge from others to emphasize on its importance. However, doing so, the design breaks the guidelines [10] recommendations not to execute actions from tab bar buttons. This is an example when it is not straightforward to follow the guidelines step by step, and sometimes it is necessary to design the UI as the domain requires.

The form (Figure 4.13d) allows users to create a transaction by filling in the amount, transaction's title, note, and date. Further, users can change the type of the transaction by tapping on "Expense" or "Income", change to which wallet, and to which category the transaction is going to belong.

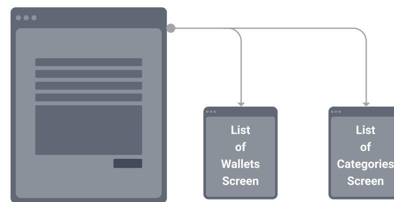


Figure 4.10: Create Form scene navigation flow

Budgets

The scene's main screen (Figure 4.13e) is a table of budgets. Each budget is displayed as a cell containing the budget's name, the actual budget, and both visual and percentage indicator of the amount spent. To create a new budget a button with "plus" icon is provided on the right side of the navigation bar. A budget's detail screen (see Figure 4.14), where users can see additional information such as budgeted categories and transactions, is presented by tapping on a cell.

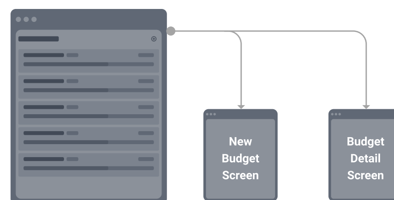


Figure 4.11: Budgets scene navigation flow

Profile

The Profile scene (Figure 4.13f) shows the user's name and e-mail address. The current scope only allows users to manage their wallets and categories from the scene or log out.

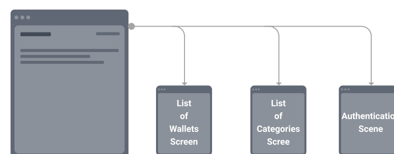
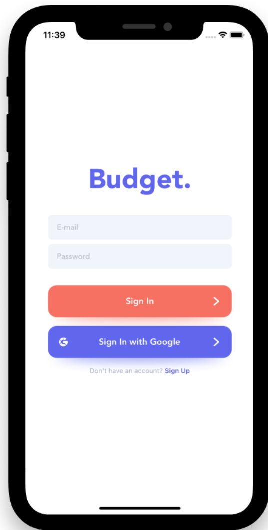
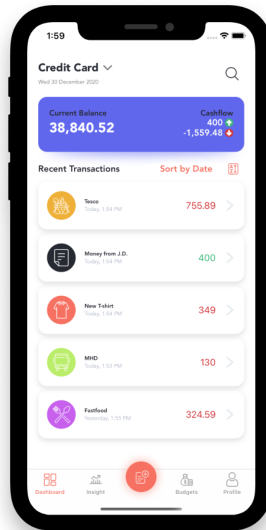


Figure 4.12: Profile scene navigation flow

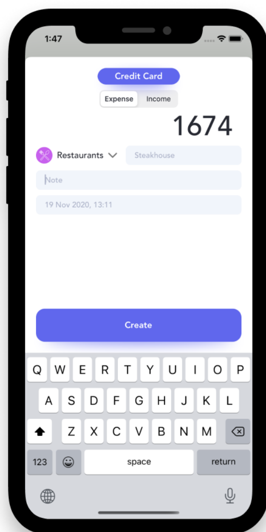


(a) Authentication

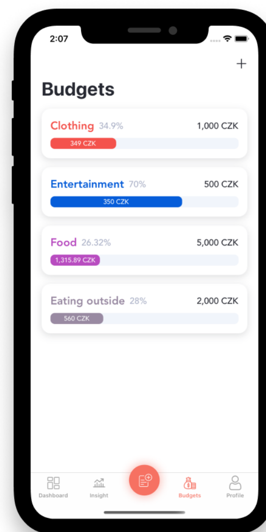


(b) Dashboard

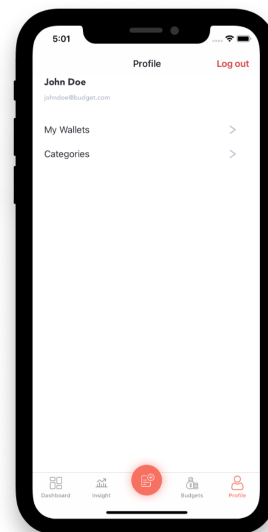
(c) Insight



(d) Create Form

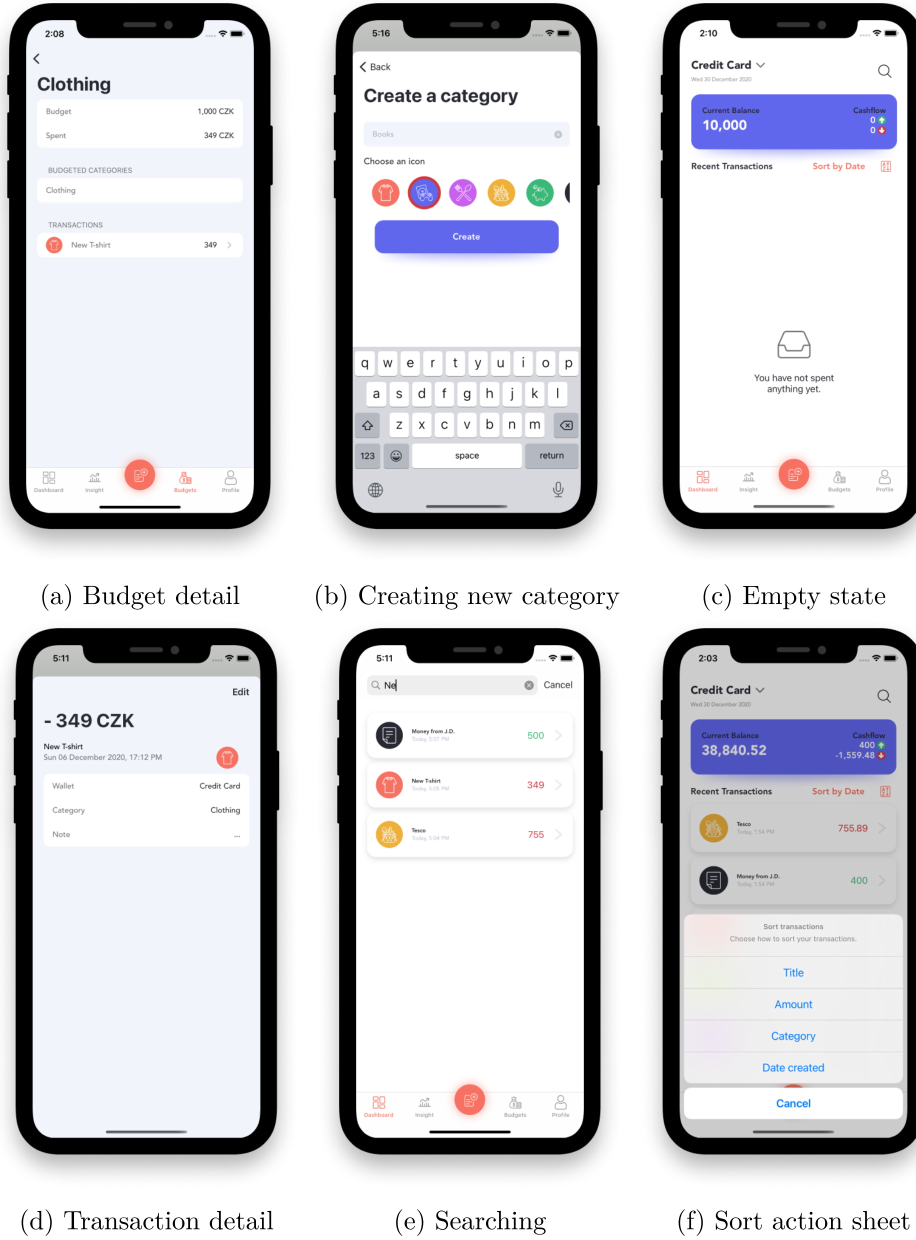


(e) Budgets



(f) Profile

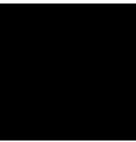
Figure 4.13: Main scenes UI



(a) Budget detail (b) Creating new category (c) Empty state

(d) Transaction detail (e) Searching (f) Sort action sheet

Figure 4.14: Additional screens UI



CHAPTER 5

Implementation

In light of the work's objective and the previous analysis, the following chapter describes the work's most important process, separated into two main sections. The first section gives a brief overview of the technologies used for the server-side implementation of the app. The second section explains how most parts of the app are implemented, starting from the codebase structure to navigation to UI components layout creation.

5.1 Firebase Backend

Firebase is Google's application development platform that helps developers build scalable mobile and web applications using the infrastructure built on Google services. The platform provides functionalities like analytics, databases, messaging, user authentication, and crash reporting. [44]

In other words, the *Firebase* platform as the backend is an additional application that is accessed through the Internet via provided API from frontend clients—the iOS app. This section is concerned about *Firebase* technologies used in the implementation process to disburden the client-side app business logic from the backend-side logic, which will be of great benefit in the long-run as both codebases will be insulated.

5.1.1 Firebase Authentication

Firebase Authentication is one of *Firebase's* services that provide end-to-end identity solutions, supporting e-mail and password accounts, phone authentication, and authentication using social network accounts like Google, Facebook, GitHub, and many more. The service removes the burden of implementing a secure authentication system from scratch, which is incredibly difficult

without any prior knowledge. In addition to that, it also provides a customizable drop-in authentication API that handles the UI flow for signing users. [45]

The current implementation leverages the service's advantages of letting users easily sign-in or sign-up using an e-mail and password account or a Google account. Listing 5.1 shows how the *Firebase Authentication* API takes care of the hard part of authenticating users. The API performs an asynchronous method `signIn(email:password:completion)` in the background leaving only the responsibility of handling its callback. In the event of an error, an alert is presented to the user; otherwise, the application navigates to the app's main screen.

```
1  typealias Handler = (Result<Void, Error>) -> Void
2
3  /// Authenticate user using an e-mail password account.
4  func signIn(email: String, password: String, handle: @escaping Handler) {
5      Auth.auth().signIn(withEmail: email, password: password) {
6          result, error in
7              if let error = error {
8                  handle(.failure(error))
9                  return
10             }
11
12             // handle callback - i.e. navigate to Dashboard Scene
13             handle(.success(()))
14     }
15 }
```

Listing 5.1: An authentication example using *Firebase Authentication*

5.1.2 Cloud Firestore

Cloud Firestore is a serverless NoSQL document-oriented database that allows developers to build hierarchies to store, synchronize, and query data structured in the form of collections and documents [46]. As Figure 5.1 shows, a collection is the root element of the data model containing multiple documents. Documents are objects containing the stored data as key-value pairs and may even contain subcollections [47].

Using *Firestore*, the app is able to store user's data online, allowing data sharing across multiple devices in case the user is signed-in on more devices. Additionally, *Firestore* offers seamless integration with other *Firebase* services, including *Firebase Authentication* and *Cloud Functions* (read Subsection 5.1.3) [46]. That said, the database is chosen to be the only persistence storage for the app due to its provided services.

The data model of the database consists of one root `users` collection containing documents for each registered user. A user's document, identified by the registered e-mail address, holds four subcollections—`categories`, `wallets`, `transactions`, and `budgets`—persisting precisely what the name suggests.

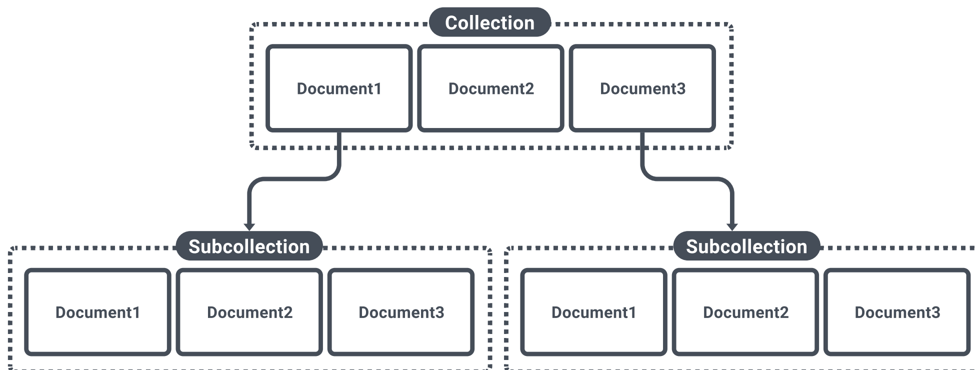


Figure 5.1: *Cloud Firestore* NoSQL data model [46]

Persisting users' data is made simple using *Firestore* API and Swift's `Codable` protocol [48]. The `Codable` protocol provides easy data serialization to and from an external representation, such as JavaScript Object Notation (JSON), which is perfect since *Firestore* represents the data in JSON format [49]. Listing 5.2 demonstrates how a transaction is created and persisted in the database without the need for explicit serialization, whereas Listing 5.3 shows the transaction's representantion in JSON.

```
1  struct Transaction: Codable {
2      let title: String
3      let amount: Double
4  }
5
6  let database = Firestore.firestore()
7
8  /// Create a transaction for a specific user
9  func create(_ transaction: Transaction) {
10     do {
11         // the serialization is done internally thanks to Codable protocol
12         _ = database.collection("users")
13                 .document("userId") // an example user identifier
14                 .collection("transactions")
15                 .addDocument(from: transaction)
16     } catch let error { /* handle error state */ }
17 }
```

Listing 5.2: Persisting data on *Cloud Firestore*

```
1  {
2      "title": "Shopping",
3      "note": "Bought food for dinner",
4      "amount": 634.78,
5      "category": { "name": "Grocery" },
6      "walletRef": { /* FIRDocumentReference */ }
7  }
```

Listing 5.3: An example of persisted transaction in JSON

5.1.3 Cloud Functions

Cloud Functions is a serverless service that lets developers implement and deploy JavaScript or TypeScript backend code in the cloud. The code is deployed to Google's Cloud infrastructure and gets executed in response to events triggered by *Firebase*; for instance, *Firestore* read and write events. The service is fully insulated from both the database and the iOS client application, so it guarantees a certain level of separation and scalability of both codebases as one is not dependant on the other. It is absorbing to note that using *Cloud Functions*, a part of the codebase is not reverse-engineerable. [50]

This solution is suitable for performing additional operations next to the app's primary operations. For instance, creating a default list of categories on every user's registration or performing a side-effect when creating a transaction (e.g., altering a wallet's balance). The example code in Figure 5.4 illustrates the implementation of `onCreate` cloud function that is triggered whenever the user creates a transaction, or more precisely, whenever a write event occurs in the user's `Transactions` collection. Alternatively, a similar function may be triggered upon read or update events as well.

```
1  export const updateWalletOnTransactionCreate = functions.firestore
2      .document("users/{userId}/transactions/{transactionId}")
3      .onCreate(async (change, context) => {
4          const transaction = change.data() // the created transaction
5          const userId = context.params.userId
6          const db = admin.firestore()
7
8          // async. retrieve wallet's snapshot to alter its balance
9          const walletSnap = await db
10             .collection("users")
11             .doc(userId)
12             .collection("wallets")
13             .doc(change.data().walletId)
14             .get()
15
16         // calculate new wallet balance and income/expense
17         // based on the transaction type
18         await walletSnap.ref.update({
19             /* new wallet state with updated balance */
20         })
21     })
```

Listing 5.4: A *Cloud Functions* example in TypeScript

5.2 iOS Application

The following section outlines the iOS client's implementation process for personal finance management and budgeting. The text proceeds chronologically similarly to how the implementation does, i.e., starting from the project's structure planning and dependency management to navigation between screens to the realization. The mentioned process has a significant impact on the codebase's final internal structure, considering how it will be

testable, scalable, and reusable in the future. Finally, the section describes the process of creating the UI and the steps during the implementation of several important features.

5.2.1 Reactive Programming

Mobile applications expect to be interacted with by users, and the personal finance app is no exception. The interaction is represented as an event such as tapping on a button to create a transaction. Such an event changes the UI and the state of the application. A list of transactions gets refreshed and updated with the newly created transaction, a title displaying the current wallet's balance changes by an amount, or a budget's spent amount changes due to the transaction being in a budgeted category.

Hence, the reactive programming paradigm, which provides a way of working with asynchronous data and propagation of change [51], is used in the implementation to handle these events in a clean and maintainable manner. More specifically, the implementation utilizes the *RxSwift* library, which lets developers write asynchronous code reacting to new data and events in a sequential, isolated manner [52]. Moreover, *RxSwift* goes well with *view model* objects as it allows simple data model binding. As Figure 5.2 and Listing 5.5 show, a *view model* object exposes `Observable<T>` properties that are bind directly to *view* object's UI components [52].

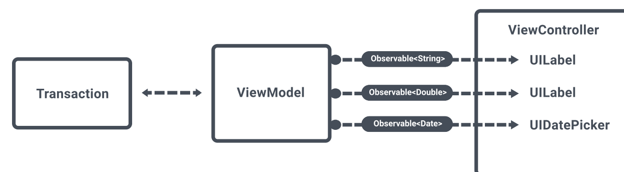


Figure 5.2: Data binding diagram example

```
1 // view model object with exposed observables
2 final class TransactionsListViewModel {
3     let title: Observable<String>
4     let amount: Observable<String>
5 }
6
7 // view object
8 final class TransactionsListViewController: UIViewController {
9     var viewModel: TransactionsListViewModel!
10
11     /// Data binding the view model's exposed observables to UI components
12     func bindViewModel() {
13         viewModel.title.bind(to: titleLabel.rx.text).disposed(by: bag)
14         viewModel.amount.bind(to: amountLabel.rx.text).disposed(by: bag)
15     }
16 }
```

Listing 5.5: Data binding code example

5.2.2 Clean Architecture and MVVM

Clean Architecture is an architectural pattern introduced by Robert C. Martin, which main role is to provide the Separation of Concerns (SoC). The separation is achieved by dividing the architecture into at least two layers—one for business rules and one for interfaces—with specific responsibilities. Amongst the main advantages of this pattern, and the reasons the architecture has been chosen for the implementation of this work, are testability, scalability, independent UI, and independent database. [53]

The primary rule of *Clean Architecture* is the *Dependency Rule*. The rule says that the code's dependency flow can only go from the outer layer inward, as depicted in Figure 5.3. The code in the inner layers has no knowledge or whatsoever about functions, classes, or variables on the outer layers, e.g., the code cannot have a use case implementation that is dependant on the implementation of a database. [53]

Although Figure 5.3 shows that the architecture is structured into four layers, according to [53], the number of circles is a rule of thumb and should reflect the software's needs. Nevertheless, the *Dependency Rule* should always be applied. Therefore, after generalizing all layers, the code's structure is divided into three main layers—domain, data, and presentation—which suits the needs of iOS development with *MVVM* pattern, see Figure 5.4.

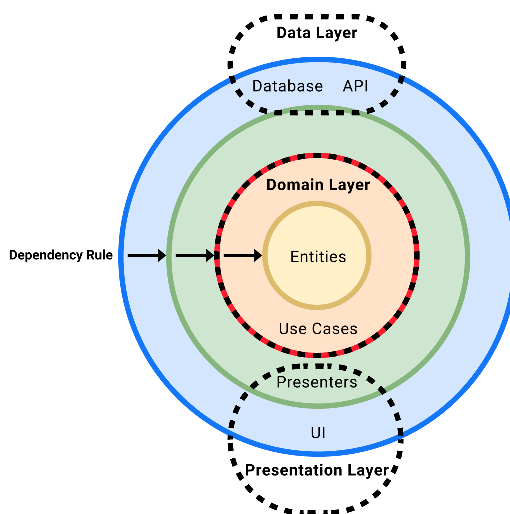


Figure 5.3: *Clean Architecture* scheme [53, 54]

Domain layer

The domain layer is the inner-most layer of the architecture containing *entities*, *use cases*, and *repository interfaces*. It is the core of the application, which is fully isolated from other layers, making the layer fully testable [54].

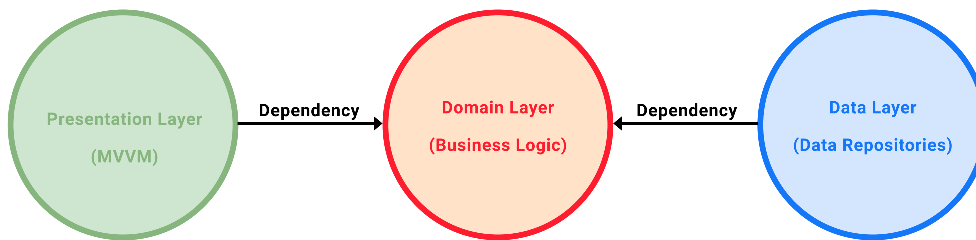


Figure 5.4: Main layers of the app based on [54]

```
 1 // Entity
 2 struct Wallet {
 3     let id: String
 4     let type: WalletType
 5     let name: String
 6     let balance: Double
 7     let income: Double
 8     let expense: Double
 9 }
10
11 // Repository interface
12 protocol WalletsRepository {
13     func create(_ wallet: Wallet) -> Observable<Void>
14 }
15
16 // (3) Use Case interface and implementation
17 protocol CreateWalletUseCase {
18     func execute(_ wallet: Wallet) -> Observable<Void>
19 }
20
21 final class CreateWalletUseCaseImpl: CreateWalletUseCase {
22     private let walletsRepository: WalletsRepository
23
24     init(walletsRepository: WalletsRepository) {
25         self.walletsRepository = walletsRepository
26     }
27
28     func execute(_ wallet: Wallet) -> Observable<Void> {
29         return walletsRepository.create(wallet)
30     }
31 }
```

Listing 5.6: Domain layer components example implementation

Data layer

The data layer contains concrete implementations of repositories defined in the domain layer. This layer manages any communication with the *Firestore* database and takes care of the data's serialization to the domain *entities*, which means it has to be dependent on the domain layer.

Presentation layer

The presentation layer is where the application's interface logic is handled. It contains UI components, *view model* objects (presenters), and *view* objects from the *MVVM* design pattern. In order to present data to users, the layer needs to be dependent on the domain layer only.

```swift
// Repository implementation
final class WalletsRepositoryImpl: WalletsRepository {
    private let firestoreService: FirestoreService

    init(firestoreService: FirestoreService) {
        self.firestoreService = firestoreService
    }

    // dependency on the domain layer's entity
    func create(_ wallet: Wallet) -> Observable<Void> {
        // firestoreService creates and persists a new wallet on Firestore
        // returns an observable sequence that emits Void
        // as the response once the process has finished
        return firestoreService.createWallet(wallet)
    }
}
```

Listing 5.7: Data Layer components example implementation

```swift
// (1) View object
final class WalletsViewController: UIViewController {
    var viewModel: WalletsViewModelType!

    private let createWalletButton = UIButton()

    func bindViewModel() { // data binding to the view model
        createWalletButton.rx.tap
            .bind(to: viewModel.onCreateWalletButtonTapped)
            .disposed(by: bag)
    }
}

// (2) View Model object
final class WalletsViewModel: WalletsViewModelType {
    // exposing observable for data binding
    let onCreateWalletsButtonTapped = Observable<Void>.just(())

    // dependency on the domain layer's use case
    init(createWalletUseCase: CreateWalletUseCase) {
        onCreateWalletButtonTapped
            .map { Wallet() }
            .flatMap { createWalletUseCase.execute($0) } // executing use case
            .subscribe(onNext: { // handling response
                // successfully created - refresh list of wallets
            }, onError: {
                // an error occured - show an alert
            })
            .disposed(by: bag)
    }
}
```

Listing 5.8: Presentation Layer components example implementation

Example

Listings 5.6, 5.7, and 5.8 and Figure 5.5 presents a diagram exemplifying the data flow of the *UC3 Create Wallet* use case (see Section 3.3) and demonstrates how each layer's component is glued together to implement the functionality. Per *MVVM's* definition in Subsection 4.1.3, `WalletsViewController` (the view object) handles the user's request to create a wallet and unidirectionally forwards the request to `WalletsViewModelType`, which executes the `CreateWalletUseCase`. After receiving an asynchronous response from the

`WalletsRepository`, the use case passes the response back to the *view model* that updates the UI with a newly created wallet or presents an alert if an error occurs. Also, note that the separation of the architecture into the three layers does not break the *Dependency Rule* since:

1. `WalletsViewController` is a UI component from the outer-most blue layer,
2. `WalletsViewModelType` is a Presenter from the green layer,
3. `CreateWalletUseCase` is a Use Case from the red layer,
4. `WalletsRepository` is the Database component from the outer-most blue layer, as Figure 5.3 shows.

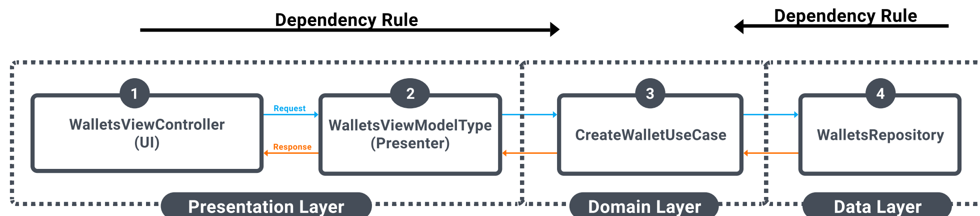


Figure 5.5: Data flow diagram for creating a wallet based on [54]

5.2.3 Dependency Management

With the given structure defined in the previous Subsection 5.2.2, it is clear that the implementation involves a considerable amount of layers and components, creating a certain relationship between one another—a dependency. Dependencies in the implementation are managed by a principle called *Dependency Injection* (DI).

According to [55], the *Dependency Injection*, which uses the fifth principle of SOLID, Inversion of Control (IoC), provides:

- better and easier application Unit testing and scalability,
- less boilerplate for the initialization of dependencies is done by the injector component,
- and enabling loose coupling.

Although numerous libraries and implementations exist providing DI, the application's implementation uses protocol composition [56] with an initializer-based injection method [57] to manage dependencies in the project

without the need for a third-party library. Using this method, all dependencies can be declared in one singleton class `AppDependency`, which is injected in other component's initializators. Owing to the protocol composition, the injected component does not have access to every dependency declared in `AppDependency`, but only the one's it conforms to with protocol conformance. The following example in Listing 5.9 shows how DI of `CreateWalletUseCase`'s dependency on `WalletsRepository` and `TransactionsRepository` is achieved. Note that the `WalletsRepository` and `WalletsRepositoryImpl` are taken from Listing 5.6 and Listing 5.7. Also, note how the IoC is applied.

```
1 // Dependency protocols
2 protocol HasWalletsRepository {
3     var walletsRepository: WalletsRepository { get }
4 }
5
6 protocol HasTransactionsRepository {
7     var transactionsRepository: TransactionsRepository { get }
8 }
9
10 /// A container storing all dependencies required in the app
11 final class AppDependency {
12     // to achieve only one instance of AppDependency in the app
13     static let shared = AppDependency()
14     private init() { }
15
16     // Implementing conformance to dependency protocols - stored dependencies
17     lazy var walletsRepository: WalletsRepository = {
18         WalletsRepositoryImpl()
19     }()
20     lazy var transactionsRepository: TransactionsRepository = {
21         TransactionsRepositoryImpl()
22     }()
23     lazy var anotherRepository: AnotherRepository = {
24         AnotherRepositoryImpl()
25     }()
26 }
27
28 // AppDependecy's conformance to dependency protocols
29 extension AppDependency: HasWalletsRepository { }
30 extension AppDependency: HasTransactionsRepository { }
31 extension AppDependency: HasAnotherRepository { }
32
33 /// A Use Case component that is dependent on WalletsRepository
34 final class CreateWalletUseCaseImpl: CreateWalletUseCase {
35     // Protocol composition - composes two protocols together (or more...)
36     typealias Dependencies = HasWalletsRepository & HasTransactionsRepository
37
38     init(dependencies: Dependencies) {
39         // can access the WalletsRepository and TransactionsRepository only
40         // the AnotherRepository is inaccessible as the Dependencies type
41         // does not conform to HasAnotherRepository protocol
42         dependencies.walletsRepository.create() // accessing the dependency
43         dependencies.transactionsRepository.doSomething()
44     }
45 }
46
47 /// Initialization of a component with its dependency
48 let createWalletUseCase: CreateWalletUseCase =
49     CreateWalletUseCaseImpl(dependencies: AppDependency.shared)
```

Listing 5.9: *Dependency Injection* example implementation

5.2.4 Navigation

As the root container for the main scenes, the combined navigation style, mentioned in Subsection 4.3.1, is implemented using `UITabBarController`, and `UINavigationController`, as the root component for the underlying hierarchical navigations. Each scene has its own `UINavigationController` consisting of a root view controller. The `UITabBarController` manages each `UINavigationController` and controls which navigation controller is displayed.

The navigation within each scene is implemented using `UINavigationController` that defines a stack-based scheme. Whenever the user navigates deeper in a navigation flow, a view controller is pushed on the navigation controller's stack. On the contrary, a view controller is popped from the stack whenever the user navigates back in a navigation flow. Since navigation controllers manage view controllers on their stacks, only the top-most view controller is displayed. In some cases, a view controller is modally presented over the top-most view controller.

Coordinator Pattern

Most of the business and presentational logic has been moved out of the view controller objects using the MVVM design pattern, which greatly solves the *Massive View Controller* problem. However, the responsibility to control the navigation in the application is still encoded in a view controller's code. This responsibility is separated from view controllers using a *Coordinator Pattern* introduced by Soroush Khanlou [58].

The navigation code is decoupled from the view controller and moved to a *coordinator* object. Each scene has its `Coordinator` accountable for creating the navigation controller's root view controller, view controllers, and the navigation between them, i.e., pushing, popping, or presenting view controllers. Practically, the `Coordinator` starts the navigation flow of a scene. See Listing 5.10 demonstrating the implementation of coordinator pattern using XCoordinator [59] library.

5.2.5 UI Components

Every UI component in the app is created entirely programmatically, without using Storyboards and XIB (XML Interface Builder) files. A programmatic UI provides better performance making the app build faster since it does not have to load XIB files from disk and convert them to NIB (NeXTSTEP Interface Builder) files [60, 61]. Further, components in code are more familiar for developers, better to debug, provide more control, and especially easier to resolve merge conflicts [61].

Nonetheless, the programmatic approach's downside is worse visualization of the UI and app's flow as the component's creation is not maintained in one

```
import XCoordinator

// each route represents a possible step in a navigation flow
enum InsightRoute: Route {
    case insight
    case filter
}

/// A coordinator managing a navigation controller
final class InsightCoordinator: NavigationCoordinator<InsightRoute> {
    init() {
        // initialize the coordinator with an initial route
        super.init(initialRoute: .insight)
    }

    /// Decide how to transition to the given route
    /// - pushing, popping (or dismissing if a view controller is presented
     modally)
    override func prepareTransition(for route: InsightRoute) ->
     NavigationTransition {
        switch route {
        case .insight:
            // injecting a weak reference of the InsightCoordinator instance
            // into view controller's view model to trigger different
            // routes in the navigation flow
            let viewmodel = InsightViewModel(router: unownedRouter)
            let viewController = InsightViewController(with: viewmodel)
            // push on the navigation stack
            return .push(viewController)
        case .filter:
            // present the view controller modally
            return .present(FilterViewController())
        }
    }
}

final class InsightViewController: UIViewController {
    private let viewModel: InsightViewModelType

    init(with viewmodel: InsightViewModelType) { self.viewModel = viewmodel }

    override viewDidLoad() {
        super.viewDidLoad()
        // tell the view model to navigate to the filter screen
        viewModel.navigateToFilter()
    }
}

final class InsightViewModel {
    private let router: UnownedRouter<InsightRoute>

    init(router: UnownedRouter<InsightRoute>) { self.router = router }

    func navigateToFilter() {
        // trigger the filter route on InsightRoute
        // (will call the prepareTransition(for:) method)
        router.trigger(.filter)
    }
}
```

Listing 5.10: Coordinator Pattern using XCoordinator [59] library

place compared to Storyboards [61]. Additionally, laying out UI components is required to be done programmatically as well using Auto Layout. Natively, using either NSLayoutConstraint class or Visual Format Language (VFL), which allows creating constraints using ASCII formatted strings [62]. In any case, the current implementation uses a library called *SnapKit* [63], providing

the same functionalities as the native methods do; however, making the laying out more readable with less code. Listing 5.11 below shows the usage of *SnapKit* to create a `UIView` component within another one filling its space with padding and the comparison with the NSLayoutConstraint code. All UI components in the implementation are using the demonstrated method.

```
1  import SnapKit
2
3  let superView = UIView() // creating component
4  let childView = UIView()
5  let padding: CGFloat = 20
6
7  superView.addSubview(childView)
8
9  // NSLayoutConstraint version
10 // enable custom Auto Layout constraints
11 childView.translatesAutoresizingMaskIntoConstraints = false
12 NSLayoutConstraint.activate([
13     childView.leadingAnchor.constraint(equalTo: superView.leadingAnchor,
14                                        constant: padding),
15     childView.trailingAnchor.constraint(equalTo: superView.trailingAnchor,
16                                         constant: -padding),
17     childView.topAnchor.constraint(equalTo: superView.topAnchor,
18                                    constant: padding),
19     childView.bottomAnchor.constraint(equalTo: superView.bottomAnchor,
20                                       constant: -padding)
21 ])
22
23 // SnapKit version
24 childView.snp.makeConstraints {
25     $0.edges.equalToSuperview().inset(padding)
26 }
```

Listing 5.11: Creating and laying out UI components example

CHAPTER 6

Testing

This chapter is an overview of the software testing process conducted during and after the implementation process. The app's testing is separated into three test types—unit, UI and usability tests. The following sections describe each test, how it is carried out, and what the results are.

6.1 Unit Tests

As part of the testing, the source code also includes a set of unit tests to ensure the correctness of individual units of the app. The unit tests follow the FIRST principles, which are:

- **Fast**: tests should not take long to execute,
- **Independent**: tests should be isolated from the testing environment,
- **Repeatable**: tests should always return the same result, regardless of the testing environment,
- **Self-validating**: the programmer should not validate the test's result,
- **Thorough**: tests should cover as many use cases as possible. [64]

To write and make unit testing follow the "Three A's of Testing: Arrange, Act, Assert" (also referred to as Given-when-then), the tests use Swift's `XCTest` framework [65], which includes `XCTestCase` class for defining test cases and test methods. The testing environment is set by overriding `XCTestCase's setUp()` method giving space to the definition of an initial custom state set before each test. After each test, the environment may be cleaned up by calling an overridden `tearDown()` method.

Additionally, the tests use *RxTest* [66] library for testing reactive streams of data written in *RxSwift*. This library provides extra testing functionalities, such as `TestScheduler`, that allows easy data stream testing in a virtual time, thanks to which events that are emitted over time can be tested [52].

The testing process checks most of the application's *view model* logic, for instance, sorting, filtering, searching, or credentials format validation. Moreover, with the provided DI (see Section 1), the *view model's* dependencies are effortlessly replaced with mocked implementations conforming to required protocols. For example, the `MockedAppDependency` contains mocked implementations of services and repositories used in *view models*. Listing 6.1 puts all of the above in an example demonstrating a unit test for the searching functionality. The functionality is expected to return a set of transactions, which attributes (title, amount, date created, etc.) match the search phrase.

```
1  final class TransactionsSearchViewModelTests: XCTestCase {
2      private var viewModel: TransactionsSearchViewModelType!
3
4      override func setUp() {
5          super.setUp()
6          let mockedTransactions = [
7              Transaction(amount: 10, title: "Robot"),
8              Transaction(amount: 100, title: "Robert"),
9              Transaction(amount: 120, title: "Berta"),
10             Transaction(amount: 200, title: "Bot"),
11             Transaction(amount: 400, title: "Root"),
12             Transaction(amount: 412, title: "Rotob")
13         ]
14
15         scheduler = TestScheduler(initialClock: 0)
16         viewModel = TransactionsSearchViewModel( // dependency injection
17             allTransactions: mockedTransactions,
18             dependencies: MockedAppDependency.shared
19         )
20     }
21
22     func testSearchByAmount() {
23         // for observing events emitted by the viewModel
24         let filteredObserver = scheduler.createObserver([Double].self)
25
26         // observer 'filtered' observable
27         viewModel.outputs.filtered.drive(filteredObserver).disposed(by: bag)
28
29         // mock user typing search phrases - .next(event time, search phrase)
30         // and bind to the view model (mocks data biding)
31         let mockedEvents = [.next(10, "1"), .next(20, "12"), .next(30, "")]
32         scheduler.createColdObservable(mockedEvents)
33             .bind(to: viewModel.inputs.onSearchTextTyped)
34             .disposed(by: bag)
35
36         scheduler.start()
37
38         // test exact time and result
39         XCTAssertEqual(filteredObserver.events, [
40             .next(10, [10.0, 100.0, 120.0, 412.0]),
41             .next(20, [120.0, 412.0]),
42             .next(30, [10.0, 100.0, 120.0, 200.0, 400.0, 412.0])
43         ])
44     }
45 }
```

Listing 6.1: A unit test example

6.2 UI Tests

Due to the fact that individual UI components were manually tested in parallel with the implementation using a simulator, UI tests were given less attention. Regardless, the source code contains several UI tests validating the UI against business requirements, i.e., the scene's initial state correctness or the screen's state after some workflow.

The UI tests are written with the support of the `XCTest` framework and XCode's recording tool that records the developer's interaction with the simulator, converting them to code that is later asserted. Below, Listing 6.2 illustrates a single UI test that checks the initial state of a screen.

```
1 var app: XCUIApplication!
2
3 override func setUp() {
4     super.setUp()
5     app = XCUIApplication()
6     app.launch()
7 }
8
9 func testDefaultBudgetCreateScreenState() throws {
10     // generated by the recording tool
11     app.tabBars["Tab Bar"].buttons["Budgets"].tap()
12     app.navigationBars["Budgets"].buttons["Add"].tap()
13
14     // assert UI components' state
15     for cell in app.tables.cells.allElementsBoundByIndex {
16         XCTAssertFalse(cell.isSelected)
17         XCTAssertTrue(cell.isEnabled)
18     }
19
20     XCTAssertFalse(app.buttons["Create"].isEnabled)
21 }
```

Listing 6.2: A UI test example

6.3 Usability Tests

The usability test is a method of testing the intuitiveness of the app's design using a group of representative users, which oftentimes leads to exposing design flaws in terms of both the UI and the functionality of the app's features. Also, this method helps to understand the target group's behavior and discover opportunities to make improvements. [67]

The aim is to address a small group of testers having experience in using similar apps. Alternatively, the users showing interest in using such an app. Before the testing, the app's characteristics and basic features were shared with each tester. Testers received an identical set of intuitive and straightforward tasks to perform (see Subsection 6.3.1)—such tasks are critical to understand for day-to-day usage of the app. Moreover, these tasks were to validate the fulfillment of the functional requirements defined in Section 3.1.

The testing was conducted separately, and each tester was required to think out loud to understand his behavior, thoughts, and motivations better.

6.3.1 Test Scenarios

The following list of test scenarios describes individual tasks carried out during usability testing. The descriptions only provide brief information about what is required to do, not the steps how or what is expected to happen, so as not to limit the tester's freedom of choice. This method has been chosen to deduce the app's intuitiveness from the tester's actions better and whether the outcome matches the tester's expectations.

1. **Registration**

   Register as a new user and proceed as required. You do not necessarily have to use your personal e-mail address.

2. **Adding transactions**

   Create one or more transactions with information as you see fit and choose to display one of the transactions detail screen. Also, try adding a transaction in "Sport" category.

3. **Editing transaction**

   Choose one of the trasanctions you have created, edit it as you want and save the changes.

4. **Adding wallets, categories and budgets**

   Go around the app and create one wallet, category, and budget.

5. **Viewing a budget detail**

   Look into one of your budget's detail to see what categories the budget includes.

6. **Searching**

   Search for a transaction that you have added before.

7. **Viewing statistics**

   See how much you have spent in the current month and in what categories. After seeing the statistics, you have realized that the numbers do not add up. Delete one transaction.

8. **Sorting**

   Sort the current month's transactions by an amount to see what was your highest transaction.

6.3.2 Results

The usability testing was attended by five testers of various backgrounds and experience with financial apps. Two of them were using another similar app at the time, whereas the rest were iOS users who wanted to get involved with such an app. The testing was held in-person with the author and each tester individually. The testers were given a brief time to explore the app, considering the fact that it was their first time using it. Therefore, it was a must for them to go through the app at least once to grasp the app's navigation and UI elements layout.

The test scenarios were to verify the app's most critical use cases' intuitiveness, which owing to the testing and the testers' feedback, have proved to be clear and intuitive. Despite the fact that testers were able to complete all tasks successfully, there still were a few uncertainties during the testing. Below are the issues, which were appropriately changed based on the testers' report:

- The label showing a wallet's current balance did not format the amount clearly, making larger amounts hard to read. Therefore, a decimal number formatting has been added to the labels showing an amount.
- It was not apparent that some labels could trigger an action, for instance, wallet's name in the Dashboard scene or the selection of category while creating a transaction. As a result, a downward icon has been added, indicating action.
- While editing a transaction, it was not obvious which fields were editable. In response, fields that are not editable have been disabled or have an indicating icon hidden.
- It was not intuitive what keywords the user could search transactions by; hence, the placeholder changed from "Search" to "Search by transaction attributes".

Further, as reported by the testers, the app and the navigation within felt natural and intuitive by dint of clean separation of scenes, workflows, and native design. The workflows contained the same UI elements in various places in the app with the same functionality, making them understandable. Also, the workflows did not require tricky navigation making features comfortably accessible.

CHAPTER 7

Results and Future Development

7.1 Results

The result of this bachelor thesis is a fully functional prototype of the specified mobile application meeting the requirements in Chapter 3. Compared to the existing solutions analyzed in Chapter 2, the final app provides every analyzed feature, i.e., allows users to create an account, lets users create an infinite number of wallets and budgets, manages transactions under various categories for better categorization, and offers transactions sorting, filtering and searching.

Nevertheless, it is apparent there are many differences in the provided features, the most notable of which is the unlimitedness of wallets and budgets or the absence of extra features such as transaction labels, or an overview of wallets. However, the final prototype is implemented to provide the necessary yet critical features required to manage personal finances and budgeting.

7.2 Future Development

Although the app provides all fundamental features for personal finance management app, there are variety of ideas in regards its improvement. These ideas should be taken into consideration before deploying the app to production.

Onboarding The usability testing outcome resulted in the necessity of onboarding flow as the testing users appreciated a brief introductory time to explore the app. The app would present an onboarding flow to the

user upon its first installation, describing essential feature flows and elements. Such a feature would significantly improve the user's experience when onboarding.

Local data persistence While it was not acknowledged during the development as an essential feature, storing the user data in local persistent storage would decrease the user's network data usage and increase the app's performance considering restoring the app's state and displaying already downloaded data. Additionally, the feature would allow users to create transactions offline when the network is unavailable.

Improve budgets Since the current implementation allows users to create monthly budgets only, future versions of the app could provide more options such as daily, weekly, or yearly budgets. Also, the created budget is not reoccurring, which means users cannot set the budget to reappear each period.

Profile customization Although profile customization is a matter of needs in such an app, future versions could allow users to change their e-mail address and password.

Notifications To further improve users' experience using the app, it could automatically send notifications as a reminder to record daily expenditures and revenues. The app could as well allow users to set up custom notifications.

Support different platforms Considering the app's availability, it would be of great benefit if the app was available on different platforms such as macOS, web, or Android. It would give users various options on where to use the app and attract new users.

7.3 Distribution

The app has not been published yet due to the unavailability of a paid Apple Developer Program account mandatory for its distribution. Therefore, the app's future development might consider publishing it to the App Store.

The project needs to be thoroughly tested and prepared before distribution by providing required information such as a unique bundle ID, build string, app icon, and launch screen. One should take caution when setting the information since most of them is not editable after the distribution. With an active Apple Developer account and a ready-made project, the application can then be archived and uploaded to the App Store Connect for application distribution using the TestFlight testing platform or through the App Store. [68]

While TestFlight is used to distribute beta versions of the app to internal or external testers, the App Store distributes production versions. To distribute the app on the App Store, it must undergo strict app quality control, so it is necessary to submit the app to App Review, which verifies the fulfillment of Apple Store Review Guidelines. Only after the app is approved, it can be published to the App Store. [69]

Conclusion

This bachelor thesis' primary objective was to design an iOS mobile application, which would provide users with a straightforward way to manage their personal finances and budgets. Another goal was to analyze existing solutions, emphasizing their advantages and disadvantages, based on which functional and non-functional requirements would be defined. Finally, develop a functional prototype of the mobile application, which would subsequently be the subject to appropriate tests.

Prior to the application's design, an analysis of existing solutions was performed, specifically *Spendee*, *Wallet*, and *Pocket Expense*. Based on this analysis, the shortcomings of these solutions were identified, and the main functional requirements of the designed application were defined. The analysis was followed by the design of the application, during which great focus was placed on the design of sustainable architecture, reusability, scalability, and testability of the code base, as well as the simplicity of the user interface. During the development, best practices were applied, and proven technologies in iOS mobile development were used. Throughout the implementation, the application was continuously tested by the developer, and in the final part was subjected to usability testing. Finally, possible steps for future development were described. The result of this work is a fully functional prototype that meets all defined goals and requirements. Additionally, the application is ready for future development.

Last but not least, working on the bachelor thesis has provided the author with the opportunity to try the whole process of software development all alone, starting from the analysis through the design to the implementation and the testing process. It has undoubtedly allowed the author for the acquirement of valuable knowledge and experience in writing academic work and broadening his horizons of iOS mobile development and software development methodologies.

Bibliography

1. GORDON, Kyle. Topic: Smartphones. In: *statista.com* [online] [visited on 2020-04-16]. Available from: https://www.statista.com/topics/840/smartphones.
2. Impact of Smartphones on Society – Use of Mobile Phones. In: *keyideas-infotech.com* [online] [visited on 2020-04-16]. Available from: https://www.keyideasinfotech.com/blog/impact-of-smartphone-on-society.
3. *App Store* [online]. Apple Inc., © 2020 [visited on 2020-04-17]. Available from: https://www.apple.com/ios/app-store.
4. CLEEVIO S.R.O. *Spendee - peníze a rozpočty* [software]. Version 3.15.8 [visited on 2020-04-17]. Available from: https://apps.apple.com/cz/app/spendee-pen%C3%ADze-a-rozpo%C4%8Dty/id635861140?l=cs.
5. *Case Studies – Spendee* [online]. Cleevio, © 2008–2020 [visited on 2020-04-16]. Available from: https://www.cleevio.com/spendee.
6. *Hashtag* [online]. Wikimedia Foundation [visited on 2020-04-17]. Available from: https://en.wikipedia.org/wiki/Hashtag.
7. BUDGETBAKERS S.R.O. *Wallet – příjmy a výdaje* [software]. © 2016. Version 2.12.1 [visited on 2020-04-28]. Available from: https://apps.apple.com/cz/app/wallet-daily-budget-profit/id1032467659.
8. *Budgetbakers* [online]. BudgetBakers s.r.o, © 2016 [visited on 2020-04-28]. Available from: https://budgetbakers.com.
9. HOLZMAN, Ondřej. *Revoluce u Budgetbakers, v aplikacích českého startupu půjde i platit. Hlásíme se o místo nad všemi bankami, říká jeho šéf* [online]. CzechCrunch [visited on 2020-04-28]. Available from: https://www.czechcrunch.cz/2020/04/revoluce-u-budgetbakers-v-aplikacich-ceskeho-startupu-pujde-i-platit-hlasime-se-o-misto-nad-vsemi-bankami-rika-jeho-sef/.
10. *Human Interface Guidelines* [online]. Apple Inc., 2020 [visited on 2020-04-27]. Available from: https://developer.apple.com/design/human-interface-guidelines/ios.

11. APPXY INFORMATION TECHNOLOGY CO., LTD. *Pocket Expense 6* [software]. 2018. Version 6.4.2 [visited on 2020-05-04]. Available from: https://apps.apple.com/cz/app/pocket-expense-6/id424575621?l=cs.
12. ABRAN, Alain; MOORE, James W; BOURQUE, Pierre; DUPUIS, Robert; TRIPP, L. Software Requirements. In: *Software engineering body of knowledge*. IEEE Computer Society, Angela Burgess, 2004, chap. 2.
13. *Non-functional requirement* [online]. Wikimedia Foundation [visited on 2020-06-23]. Available from: https://en.wikipedia.org/wiki/Non-functional_requirement.
14. COCKBURN, Alistair. Introduction. In: *Writing effective use cases*. Addison-Wesley Professional, 2000, chap. 1, p. 1.
15. About Objective-C. In: *Programming with Objective-C* [online]. Apple Inc., 2014 [visited on 2020-11-29]. Available from: https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html.
16. *Swift.org* [online]. Apple Inc., 2020 [visited on 2020-11-29]. Available from: https://swift.org.
17. *Swift* [online]. Apple Inc., 2020 [visited on 2020-11-29]. Available from: https://developer.apple.com/swift.
18. *UIKit* [online]. Apple Inc., 2020 [visited on 2020-12-01]. Available from: https://developer.apple.com/documentation/uikit.
19. HUDSON, Paul. What is SwiftUI? In: [online]. Hacking With Swift, 2019 [visited on 2020-12-01]. Available from: https://www.hackingwithswift.com/quick-start/swiftui/what-is-swiftui.
20. MOAKLEY, Brian. UIKit Fundamentals. In: [online]. Razeware LLC., 2020 [visited on 2020-12-01]. Available from: https://www.raywenderlich.com/16124941-uikit-fundamentals/lessons/1.
21. HUDSON, Paul. Answering the big question: should you learn SwiftUI, UIKit, or both? In: [online]. Hacking With Swift, 2019 [visited on 2020-12-01]. Available from: https://www.hackingwithswift.com/quick-start/swiftui/answering-the-big-question-should-you-learn-swiftui-uikit-or-both.
22. TAM, Audrey. SwiftUI: Getting Started. In: [online]. Razeware LLC., 2019 [visited on 2020-12-01]. Available from: https://www.raywenderlich.com/3715234-swiftui-getting-started.
23. *SwiftUI Tutorials* [online]. Apple Inc., 2020 [visited on 2020-12-01]. Available from: https://developer.apple.com/tutorials/swiftui.

24. Introduction. In: *Architectural Patterns* [online]. The Open Group, 2001 [visited on 2020-11-29]. Available from: http://www.opengroup.org/public/arch/p4/patterns/patterns.htm.

25. KALELKAR, Medha; CHURI, Prathamesh; KALELKAR, Deepa. Implementation of model-view-controller architecture pattern for business intelligence architecture. *International Journal of Computer Applications* [online]. 2014, vol. 102, no. 12 [visited on 2020-11-29].

26. VERWER, Dave. *The iOS Developer Community Survey* [online]. © 2019-2020 [visited on 2020-11-29]. Available from: https://iosdevsurvey.com/2019.

27. LASO-MARSETTI, Felipe. Model-View-Controller (MVC) in iOS – A Modern Approach. In: [online]. Razeware LLC., 2019 [visited on 2020-11-29]. Available from: https://www.raywenderlich.com/1000705-model-view-controller-mvc-in-ios-a-modern-approach.

28. HUDSON, Paul. What is MVC? In: [online]. Hacking With Swift, 2019 [visited on 2020-11-29]. Available from: https://www.hackingwithswift.com/example-code/language/what-is-mvc.

29. Model-View-Controller. In: [online]. Apple Inc., © 2012 [visited on 2020-11-29]. Available from: https://developer.apple.com/library/archive/documentation/General/Conceptual/CocoaEncyclopedia/Model-View-Controller/Model-View-Controller.html.

30. ALJAMEA, Mariam; ALKANDARI, Mohammad. MMVMi: A validation model for MVC and MVVM design patterns in iOS applications. *IAENG Int. J. Comput. Sci* [online]. 2018 [visited on 2020-11-29].

31. MVVM in Practice - RWDevCon Session - raywenderlich.com. In: *Youtube* [online]. 2016 [visited on 2020-11-30]. Available from: https://www.youtube.com/watch?v=sWx8TtRBOfk. Channel raywenderlich.com.

32. GREENE, Joshua; STRAWN, Jay, et al. *Design Patterns by Tutorials: Learning design patterns in Swift 4*. Razeware LLC., 2018.

33. Model-View-Controller. In: [online]. Apple Inc., © 2018 [visited on 2020-11-29]. Available from: https://developer.apple.com/library/archive/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html.

34. KRUTSINGER, Chuck. iOS MVVM Tutorial: Refactoring from MVC. In: [online]. Razeware LLC., 2020 [visited on 2020-11-30]. Available from: https://www.raywenderlich.com/6733535-ios-mvvm-tutorial-refactoring-from-mvc.

35. CACHEAUX, Rene. Advanced Unidirectional Architecture with Rene Cacheaux - Live Tutorial Session - RWDevCon 2018. In: *Youtube* [online]. 2019 [visited on 2020-11-30]. Available from: https://www.youtube.com/watch?v=OD_yHH_R7qo. Channel raywenderlich.com.

36. LEE, Antoine Van Der. Picking your minimum iOS version to support. In: [online]. SwiftLee, 2019 [visited on 2020-12-30]. Available from: https://www.avanderlee.com/workflow/minimum-ios-version.
37. WUERTHELE, Mike. Vast majority of all iPads, iPhones in service use iOS 13. In: [online]. Quiller Media, Inc., 2020 [visited on 2020-12-30]. Available from: https://appleinsider.com/articles/20/01/28/vast-majority-of-all-ipads-iphones-in-service-use-ios-13.
38. STAFF, MacRumors. iOS 14. In: [online]. MacRumors.com, LLC., 2020 [visited on 2020-12-30]. Available from: https://www.macrumors.com/roundup/ios-14.
39. *App Store* [online]. Apple Inc., © 2020 [visited on 2020-12-30]. Available from: https://developer.apple.com/support/app-store.
40. LARMAN, Craig. Domain Model - Visualizing Concepts. In: *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Pearson Educations (US), 2004, chap. 10.
41. LAUESEN, Soren. *User interface design: a software engineering perspective*. Pearson Education, 2005.
42. NIELSEN, Jakob. 10 Usability Heuristics for User Interface Design. In: [online]. Nielsen Norman Group, 2020 [visited on 2020-12-02]. Available from: https://www.nngroup.com/articles/ten-usability-heuristics.
43. JOHNSON, Jeff. *Designing with the mind in mind: simple guide to understanding user interface design guidelines*. Morgan Kaufmann, 2020.
44. *Firebase* [online]. Google LLC., 2020 [visited on 2020-12-04]. Available from: https://firebase.google.com.
45. *Firebase Authentication* [online]. Google LLC., 2020 [visited on 2020-12-04]. Available from: https://firebase.google.com/products/auth.
46. *Cloud Firestore* [online]. Google LLC., 2020 [visited on 2020-12-08]. Available from: https://firebase.google.com/products/firestore.
47. *Cloud Firestore Documentation* [online]. Google LLC., 2020 [visited on 2020-12-08]. Available from: https://firebase.google.com/docs/firestore.
48. FRIESE, Peter. Mapping Firestore documents using Swift Codable. In: *Youtube* [online]. 2020 [visited on 2020-12-08]. Available from: https://www.youtube.com/watch?v=3-yQeAf3bLE&feature=youtu.be. Channel Firebase.
49. *Encoding and Decoding Custom Types* [online]. Apple Inc., 2020 [visited on 2020-12-08]. Available from: https://developer.apple.com/documentation/foundation/archives_and_serialization/encoding_and_decoding_custom_types.

50. *Cloud Functions for Firebase* [online]. Google LLC., 2020 [visited on 2020-12-08]. Available from: https://firebase.google.com/products/functions.
51. SARU. Introduction to Functional Reactive Programming using Swift. In: [online]. A Medium Corporation, 2019 [visited on 2020-12-10]. Available from: https://medium.com/@saru2020/introduction-to-functional-reactive-programming-using-swift-ea30b1e38309.
52. PILLET, Florent; BONTOGNALI, Junior; TODOROV, Marin; GARDNER, Scott, et al. *RxSwift: Reactive Programming with Swift*. Razeware LLC., 2017.
53. MARTIN, Robert C. The Clean Architecture. In: [online]. 2012 [visited on 2020-12-09]. Available from: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html.
54. KUDINOV, Oleh. Clean Architecture and MVVM on iOS. In: [online]. 2019 [visited on 2020-12-09]. Available from: https://tech.olx.com/clean-architecture-and-mvvm-on-ios-c9d167d9f5b3.
55. KARIA, Bhavya. A quick intro to Dependency Injection: what it is, and when to use it. In: [online]. 2018 [visited on 2020-12-10]. Available from: https://www.freecodecamp.org/news/a-quick-intro-to-dependency-injection-what-it-is-and-when-to-use-it-7578c84fa88f.
56. *Protocols* [online]. Apple Inc., 2020 [visited on 2020-12-10]. Available from: https://docs.swift.org/swift-book/LanguageGuide/Protocols.html.
57. KOFLER, Stefan. Dependency Injection Strategies in Swift. In: [online]. 2019 [visited on 2020-12-10]. Available from: https://quickbirdstudios.com/blog/swift-dependency-injection-service-locators.
58. KHANLOU, Soroush. The Coordinator. In: [online]. 2015 [visited on 2020-12-14]. Available from: https://khanlou.com/2015/01/the-coordinator.
59. *XCoordinator* [online]. Stefan Kofler and Paul Kraft, 2019 [visited on 2020-12-14]. Available from: https://quickbirdstudios.github.io/XCoordinator.
60. CARNEY, TJ. What's a XIB and Why Would I Ever Use One? In: [online]. A Medium Corporation, 2017 [visited on 2020-12-17]. Available from: https://medium.com/@tjcarney89/whats-a-xib-and-why-would-i-ever-use-one-58d608cd5e9b.

61. XCode: Using Storyboards and Xibs Versus Creating Views Programmatically. In: [online]. ByDesign Development Inc., 2020 [visited on 2020-12-17]. Available from: `https://codewithchris.com/xcode-using-storyboards-and-xibs-versus-creating-views-programmatically`.
62. SINGH, Bhagat. Building an App with only code using Auto Layout. In: [online]. Razeware LLC., 2019 [visited on 2020-12-17]. Available from: `https://www.raywenderlich.com/6004856-building-an-app-with-only-code-using-auto-layout`.
63. *SnapKit* [online]. © 2011-2020 [visited on 2020-12-17]. Available from: `https://github.com/SnapKit`.
64. RAHMAN, Tasdik. F.I.R.S.T principles of testing. In: [online]. A Medium Corporation, 2019 [visited on 2020-12-22]. Available from: `https://medium.com/@tasdikrahman/f-i-r-s-t-principles-of-testing-1a497acda8d6`.
65. *XCTest* [online]. Apple Inc., © 2020 [visited on 2020-12-22]. Available from: `https://developer.apple.com/documentation/xctest`.
66. *RxTest* [online]. CocoaPods, © 2020 [visited on 2020-12-22]. Available from: `https://cocoapods.org/pods/RxTest`.
67. Usability Testing. In: [online]. The Interaction Design Foundation, 2019 [visited on 2020-12-22]. Available from: `https://www.interaction-design.org/literature/topics/usability-testing`.
68. *Preparing Your App for Distribution* [online]. Apple Inc., © 2020 [visited on 2020-12-29]. Available from: `https://developer.apple.com/documentation/xcode/preparing_your_app_for_distribution`.
69. *App Review* [online]. Apple Inc., © 2020 [visited on 2020-12-29]. Available from: `https://developer.apple.com/app-store/review`.

APPENDIX A

Acronyms

ASCII American Standard Code for Information Interchange
API App Programming Interface
App Mobile Application
DI Dependency Injection
ID Identifier
IoC Inversion Of Control
JSON JavaScript Object Notation
MVC Model View Controller
MVVM Model View ViewModel
NIB NeXTSTEP Interface Builder
NoSQL Non-Structured Query Language
OS Operating System
SoC Separation of Concern
UI User Interface
UML Unified Modeling Language
UX User Experience
VFL Visual Format Language
WWDC Worldwide Developer Conference
XIB XML Interface Builder
XML Extensible Markup Language

APPENDIX B

Contents of Enclosed SD Card

```
readme.md.......the file with SD card contents description in MD format
src.........................................the directory of source codes
  implementation..............the directory of source codes of the app
  thesis...............the directory of LaTeX source codes of the thesis
samples............................the directory of application samples
  app.mov........................the app sample video in MOV format
text....................................the directory of the thesis text
  thesis.pdf...........................the thesis text in PDF format
```