FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF BACHELOR’S THESIS

Title:	ElateMe application for Android (continuation)
Student:	Ilia Liferov
Supervisor:	Ing. Petr Pauš, Ph.D.
Study Programme:	Informatics
Study Branch:	Web and Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of summer semester 2018/19

Instructions

A goal of the thesis is to continue in development of Android application ElateMe, that have been introduced by Bc. Georgii Solovev in his thesis. You should get the application to the commercial ready state.

Analyse:
- follow a new customer’s instructions and requirements, use FURPS+
- available libraries for Android Pay, Bitcoin payments, Paypal payments
- use appropriate UML diagrams

Design:
- update platform specific model

Implement:
- Bitcoin payment, Paypal, Android Pay
- new requirements, specially a handling of collection/wish
- sharing a product from web browser into application

Test:
- use appropriate tests
- automate unit tests and UI tests using Continuous integration

References

Will be provided by the supervisor.

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

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

Prague January 4, 2018

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

ElateMe - Android client (continuation)

Ilia Liferov

Department of Software Engineering
Supervisor: Ing. Petr Pauš, Ph.D.

May 11, 2018

Acknowledgements

First and foremost, I have to thank my supervisor Petr Pauš and project supervisor Michal Maněna. I also express my gratitude to Georgii Solovev, a previous developer, for a high quality of work done by him in his thesis.

I would like to take this opportunity to thank my family for their support during the whole period of my study.

Declaration

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

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

In Prague on May 11, 2018 ....

Citation of this thesis

Liferov, Ilia. *ElateMe - Android client (continuation).* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2018.

Abstrakt

Systém ElateMe je kombinace sociální sítě a crowdfundingové platformy. Systém pomáhá uživatelům vytvářet přání a přispívat na přání ostatních uživatelů. Jednoduše řečeno, systém ElateMe řeší problém obdarování vhodným dárkem člena rodiny, přítele nebo kolegu. Systém je integrován do autorizačního systému Facebooku, a proto uživatel ani nemusí vytvářet ElateMe účet, aby mohl systém používat. Aplikace jsou dostupné pro různé platformy, například pro webový prohlížeč, operační systémy Android a iOS. Aplikace používají backendový server, který řídí přání a příspěvky, posílá různé notifikace.

Cílem této bakalářské práce je pokračování ve vývoje klientské aplikace pro operační systém Android, kterou začal vyvíjet ve své bakalářské práci Georgii Solovev. Nejprve je potřeba zanalyzovat nové zákaznické požadavky, například správu přání, podporu přispívání a nové metody plateb (pomocí baknovní karty, PayPal, bitcoinů a Android Pay). Existující architektura aplikace může být zlepšená v případě potřeby a všechny důležité bugy musí být opravené. Nakonec je potřeba vytvořit vhodné testy pro aplikační kód a automatizovat je.

Tento text obsahuje analýzu a návrh nových požadavků, aktualizované diagramy, popisy důležitých změn v architektuře aplikace, úvod do testování aplikačního kódu a automatizace testů. Nové požadavky (především platby) byly úspěšně implementovány a popsány pomocí vhodných UML diagramů. Testy byly napsané a úspěšně automatizované.

Klíčová slova Klientská aplikace, Sociální síť, Crowdfunding, Platby, Testovaní, Průběžná integrace, Bitcoin, PayPal, Android Pay, Android, Java, RxJava 2, Robolectric, JUnit, Mockito

Abstract

The ElateMe system is a combination of social network and crowdfunding platform. It helps users to create own wishes and contribute to the others. Simply said, the ElateMe system solves the problem of giving a suitable gift to a family member, a friend or a colleague. The ElateMe system is integrated into Facebook authorization systems, so a user does not even need to create an ElateMe account in order to use the system. Applications for the ElateMe system are available on different platforms such as web browsers, Android and iOS. Applications use a backend server, which manages wishes, contributions, sending various notifications and so on.

The goal of this thesis is to continue to develop the ElateMe application for the Android operating system, which was started by Georgii Solovev in his thesis. First of all, it is needed to analyze and implement new user requirements, such as wish managing, handling of contributions, new methods of payment (wish bank card, PayPal, bitcoins and Android Pay). Existing application architecture may be improved if necessary, and all critical bugs should also be fixed. Finally, it is required to write appropriate tests for the application's code and automate them.

This text contains the new requirements analysis and design, updated platform specific diagrams, description of important changes of the application architecture, an introduction to the application's code testing and automation. The new customer's requirements (especially payments) were successfully implemented and described with appropriate UML diagrams. The application's code tests were created and successfully automated.

Keywords Client app, Social network, Crowdfunding, Payments, Testing, Continuous integration, Bitcoin, PayPal, Android Pay, Android, Java, RxJava 2, Robolectric, JUnit, Mockito

Contents

List of Figures

Introduction

Nowadays almost everyone facing the problem of choosing a suitable gift for someone. The ElateMe is the system, which is able to solve such kind of problem. It allows users to create wishes, contribute to wishes of the friends, commenting wishes and even making a surprise wishes for a chosen friend. These system's features and integration into the Facebook's identity system (for logging in) make the ElateMe system a powerful social network with features of a crowdfunding platform. The ElateMe system has applications for various platforms, such as web browser, Android operating system, and iOS.

The goal of this thesis is to continue to develop the ElateMe Android application, which was started by Georgii Solovev in his thesis. Georgii analyzed basic customer's requirements and successfully implemented them, although there were a lot of features to implement in order to make the application usable for the end users. For example, screen for creating a new wish and payments were not implemented. So, in more detail, the goal of this thesis is to analyze, design, implement the new customer's requirements and test the application's code.

The primary motivation to choose this thesis's task was to practice software development processes on the real running project, try advanced practices of Android development and contribute to the growing project.

The following text is divided into five main chapters: Analysis, Design, Implementation, Testing and Continuous Integration. The Analysis chapter describes the ElateMe application and introduces the analysis of all requirements, which should be designed and implemented. The Design chapter's goal is to present a design of the analyzed requirements with appropriate diagrams and screenshots. The most important approaches and libraries, which were used during the development are placed in the Implementation chapter. The Testing and Continuous Integration chapters describe used approaches in the testing of the application's source code and automating of the tests.

Chapter 1

Analysis

The goal of this chapter is to describe the ElateMe Android application, introduce the thesis' task to the reader, submerge the reader into some details of Android development world and identify the primary system requirements.

1.1 System description

The system description is described well in the bachelor's thesis of the previous developer, Georgii Solovev. While the primary architecture of the app will not differ too much, some parts will undergo changes because of the new requirements.

Similarities and differences:

- "*ElateMe's system will follow client-server pattern with a thin client. It will have two parts: server and mobile clients on Android and iOS platforms.*" [1] The global architecture of the ElateMe system will be the same – we will continue to develop a thin mobile client application for Android operation system.
- "*This application will use Facebook Software Development Kit (SDK) for registration and login.*" [1] The login process will remain the same, although some improvements are required. (Explained below in the design section.)
- "*For the payment transactions will be used FIO bank SDK.*" [1] Bank payments will be implemented in a different way, via in-app webview approach. (Explained below in the Bank payments section.)

The implementation approaches for the new features will be chosen based on feature requirements.

Besides the implementation of the new features, the application requires several bugfixes. (The solutions of the most significant bugs are described in this thesis.) The less important changes, from the business value point of view, are code reorganization and improvement of the code quality. These improvements have low business value (which means little impact on the new features) but help to make code cleaner and more readable for further development sake.

Thus, all changes, done in the scope of this bachelor's thesis, are divided into three main groups: for features, for bugfixes and code improvements.

1.2 Task overview

In this section a detailed analysis for each item of this thesis task is given. (some of the task items are grouped together for more convenient description)

1. **Analyzing using FURPS+ and implementing new customer's instructions and requirements (especially handling of collection/wish).**

   We need to analyze the new requirements using FURPS+ and implement them. FURPS+ is a classifying model of software quality attributes. It contains such functional and non-functional requirements as Functionality, Usability, Reliability, Performance, Supportability, and Constraints. Requirements are described below in this chapter under the Requirements specification section.

2. **Analyze available libraries for Android Pay, Bitcoin payments, Paypal payment and bank payments and implement those kinds of payments.**

   Those kinds of payments are entirely different, so we need to analyze the opportunity to implement them in the ElateMe Android application and implement if it is possible.

3. **Sharing a product from web browser into the application.**

   This is one of the functional requirements. The essence of this requirement is to make the ElateMe Android application handle links from web browser, trying to recognize the potential product for wish creation.

4. **Update platform specific model/use appropriate UML diagrams.**

   We need to update platform specific model, which was introduced in the previous bachelors' thesis [1]. It is important because this model has been changed since then and we need to reflect the changes in it. For describing new processes and architecture decisions, appropriate UML diagrams will be used.

5. **Use appropriate tests and automate testing using Continuous Integration.**

   We need to write proper tests and automate them with some Continuous Integration framework. We can choose UI tests and Unit tests as appropriate tests for Android applications.

1.3 Previous developer

The goal of this thesis is to continue in the development of ElateMe Android application. Georgii Solovev performed previous development. He has written his bachelor's thesis, and there is a lot of references to it in this text.

We got the ElateMe Android application project from the previous developer – Georgii Solovev, and it was in the good state mostly, although there were some bugs and problems with the Android SDK usage. Also, some architecture approaches were not so good, so we decided to replace them with more suitable alternatives. These improvements are described in this bachelors' thesis.

1.4 Google Play Market

To make the application publicly available, we need to choose distribution platform. There are several of platforms: Google Play Market, Amazon Appstore, SlideMe, Samsung Galaxy Apps and other less popular platforms. We are choosing Google Play Market because of several reasons:

- it is available out of the box (there is no need to install anything and register a new account),
- it feels more native and trusted for users,
- ElateMe application is building with Android SDK, so using Google Play Market is the most tested and comfortable way to publish and support the application.

Google Play is a service for digital distribution by Google. It serves as the official application store for the Android operating system, allowing users to browse and download applications developed with the Android software development kit (SDK) and published through Google.

Google Play Services provides Google Play Console. It helps a developer to test, publish and manage the application, gather various statistics and other useful features, which are essential during development and support phases of the application.

1.4.1 Publishing on Google Play Market

Publishing the application on Google Play Market can be done via Google Play Console. Before the first release rollout, we have done several required steps such as defining product details (title, descriptions), determining a category of the application, uploading graphical assets (logos and banner), providing information about supported languages and translations. All this information was provided by Michal Maněna (a project manager), so we were able to prepare our first rollout.

Each rollout of the next version of the application was done by performing such steps:

- updating version of the application in the gradle file,
- generating (and signing) of the .apk file,
- uploading to the Google Play Console,
- creating the new release in the Google Play Console.

Google Play Console offers several types of application releases:

- Alpha release (usually small number of allowed users, is used for testing purposes)
- Beta release (usually higher number of allowed users, is used for testing purposes)
- Production (an application is publicly available in the Google Play Market)

During the development phase of the project, we decided to stick to the Alpha and Beta types of releases because of testing purposes. Making the application available for a group of trusted users is very important. User's feedback is useful and helpful in finding bugs, usability problems and performance issues.

1.4.2 Google App Signing

"*Android requires that all APKs be digitally signed with a certificate before they can be installed.*" [2] Because of this requirement, we needed to generate keystore, containing our public/private key pair for app signing.

"*Because your app signing key is used to verify your identity as a developer and to ensure seamless and secure updates for your users, managing your key and keeping it secure are very important, both for you and for your users. You can choose either to opt in to use Google Play App Signing to securely manage and store your app signing key using Google's infrastructure or to manage and secure your own keystore and app signing key.*" [2]

One way is to manage signing key for our application by ourselves as shown in the Figure 1.1. "*If you choose to manage your own app signing key and keystore, you are responsible for securing the key and the keystore.*" [2] We decided to use the other way, Google App Signing, because it is more

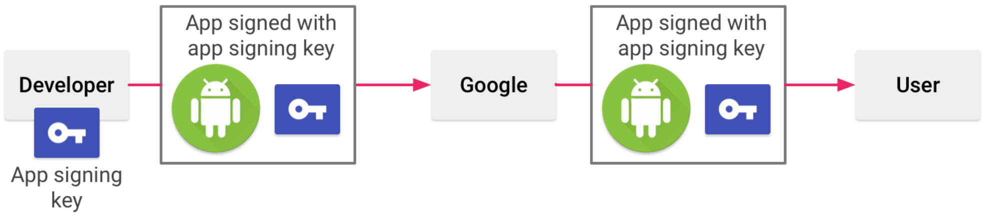

Developer
App signing key
App signed with app signing key
Google
App signed with app signing key
User

Figure 1.1: Without Google Play App Signing

convenient and secure way of managing the keystore for APK signing. Each developer has it is own key, which is used by Google App Signing for identifying trusted developers. None of the developers has access to the actual app signing key, so it can not be lost or compromised. Schematic view of this process is shown in the Figure 1.2.

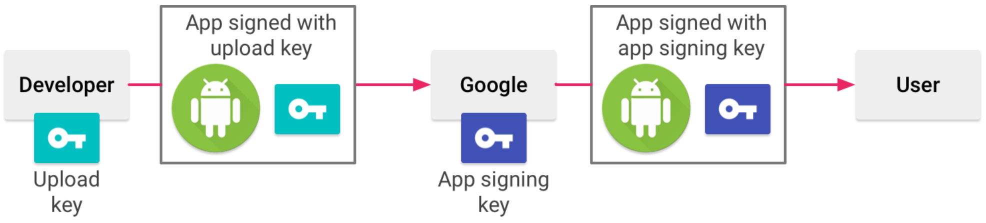


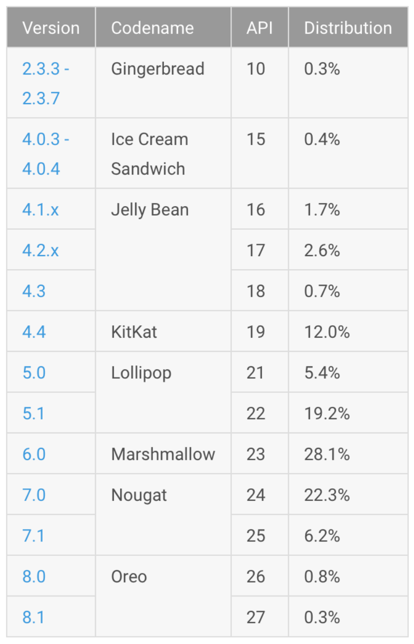
Figure 1.2: With Google Play App Signing

1.5 Minimum API level

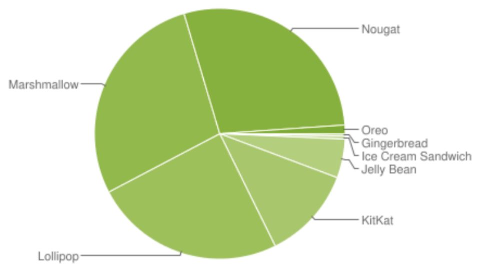
"*API Level is an integer value that uniquely identifies the framework API revision offered by a version of the Android platform.*" [3] Minimum API Level (or minimum Android SDK version) identifies the lowest Android OS version, supported by an application. Simply put, a user cannot install an application if it has higher minimum API Level than his device supports. Minimum API Level is the critical thing when we are talking about the distribution among the users.
On the other hand, some new features and conveniences (from a development perspective) become available from the higher versions of API Level. (For example, Java 8 Stream API is available from API level 24 or higher.) Thus, it is necessary to choose minimum API Level wisely.

The previous research (done by Georgii Solovev in his bachelor's thesis) shows that minimum API Level for ElateMe application is 19. The decision is based on statistics, provided by official Google pages. The API level was picked in order to cover about 90% of devices.

Michal Maněna stated that the target group of the ElateMe Android application should be about 95%. To fulfill this requirement, we needed to analyze the current state of Android devices market.
Referring to the official Google pages in the Figure 1.3, we established, that minimum API level could not be increased. Increasing even in one step, to the 21. version, would entail loss of 12% of the users.



Version	Codename	API	Distribution
2.3.3 - 2.3.7	Gingerbread	10	0.3%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	0.4%
4.1.x	Jelly Bean	16	1.7%
4.2.x		17	2.6%
4.3		18	0.7%
4.4	KitKat	19	12.0%
5.0	Lollipop	21	5.4%
5.1		22	19.2%
6.0	Marshmallow	23	28.1%
7.0	Nougat	24	22.3%
7.1		25	6.2%
8.0	Oreo	26	0.8%
8.1		27	0.3%



Data collected during a 7-day period ending on February 5, 2018.
Any versions with less than 0.1% distribution are not shown.

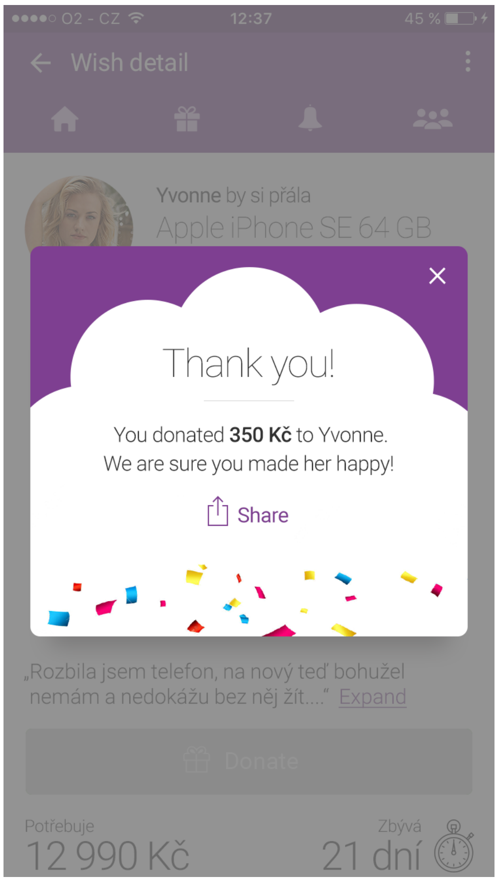
Figure 1.3: Android versions distribution [4]

1.6 Popups and notifications

The next important requirement was implementing layouts for popups and notifications.

What is popup? Popup is a small window with some essential information for a user. Each popup contains some text and possible images, which help to describe particular popup better. Any popup can suddenly appear while a user is using the application. When it appears, the user should close it in order to continue using the application. It is important to note that popup is not a push notifications and it can not be received while the application is closed or minimized.

The goal is to implement popups for all kind of messages and events in the system. Example of such popup designed by Jan Hoffman (a team designer) is shown in Figure 1.4.



Thank you!

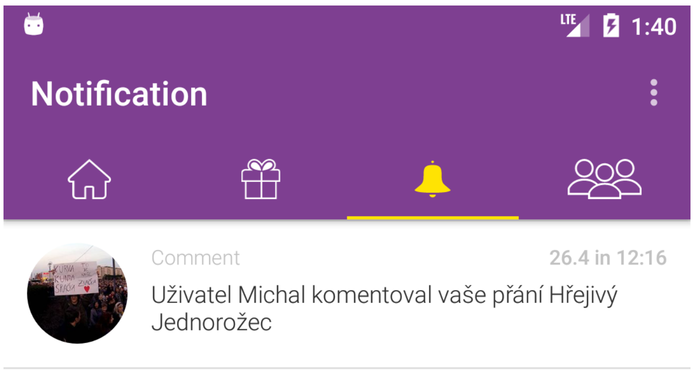
You donated **350 Kč** to Yvonne.
We are sure you made her happy!

„Rozbila jsem telefon, na nový teď bohužel nemám a nedokážu bez něj žít....“

Figure 1.4: Wireframe of some appeared popup in the application

What are notifications? Notifications are the other source of important information for a user. In our application, they are presented by the list of items on a separate fragment. The notification list is available by clicking on the corresponding tab with bell icon on the main screen.

Such functionality has been already primitively implemented by Georgii Solovev in his thesis, but in this thesis, the goal is to implement layouts for each notification type in the system and make a counter of unread notifications, which should be placed on the notifications tab icon (tab is shown in Figure 1.5).



Comment 26.4 in 12:16
Uživatel Michal komentoval vaše přání Hřejivý Jednorožec

Figure 1.5: Notifications tab (active) on the main screen of the application

1.7 Bank payments

Bank payment is one of the main features of ElateMe Android application. ElateMe system uses its bank account to collect money and hold them until a wish collection will be successful or rejected. ElateMe system has its account in the Fio banka, a.s., so our application needs to communicate with the bank somehow.

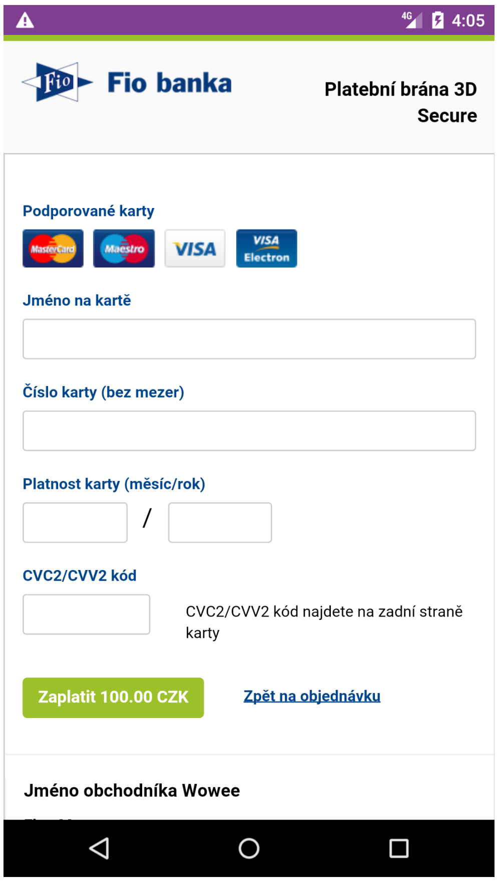
According to the previous research done by Georgii Solovev in his thesis, the application should be directly connected to the Fio bank servers by using Fio bank SDK. Despite this, we decided to do it in another way. Instead of communicating directly with Fio bank servers, ElateMe Android application will interact with the main backend server, as well as the application does it, for example, for creating wish or getting feed. Each time the application wants to make the bank payment, it will get the specific link to the Fio bank gateway, placed on the web. Thus, a user of the application can proceed his payment via a browser. You can see the example of such payment page in the Figure 1.6

1.8 Bitcoin payments

Payment in bitcoins is the other main feature of ElateMe Android application. It will allow users to make payments not only with bank card but in the most popular crypto-currency – Bitcoins.

1.8.1 What are bitcoins?

"*Bitcoin is an innovative payment network and a new kind of money.*" [5] "*Bitcoin is open-source; its design is public, nobody owns or controls Bitcoin, and everyone can take part.*" [5] Paying in bitcoins means to make a bitcoin transaction from one bitcoin address to another. "*A Bitcoin address, or simply*



CVC2/CVV2 kód najdete na zadní straně karty

Figure 1.6: Fio bank payment page in the application

address, is an identifier of 26-35 alphanumeric characters, beginning with the number 1 or 3, that represents a possible destination for a bitcoin payment." [6] To pay with bitcoins, you need to have bitcoins and address for payment. For simplicity, there are several applications, which help you to manage your bitcoins and make payments.

1.8.2 Whom to pay?

Since our ElateMe backend server is responsible for managing wish collections, choosing of a bitcoin payments provider depends on the backend server very much. Our team decided to use BitPay as a provider for bitcoin payments. The ElateMe system will have its own account in BitPay, and all bitcoin transactions will be done via this service. Thus, I need to integrate BitPay services into our Android application. Luka Lukašević has research of available providers of bitcoin payments in his thesis. [7]

1.9 PayPal payments

1.9.1 What is PayPal?

PayPal is online payment service, established in 1998. It allows users to transfer funds electronically. The money can be sent to anyone with an e-mail address, even if the recipient does not have PayPal account. There is an ability to associate PayPal account with a bank account or credit card in order to add or withdraw money. PayPal is one of the most popular payment services around the world, it is accepted by millions of e-shops, for example, by eBay.

ElateMe service is not an exception. It is important to support as much widely used payment services as possible, so PayPal is one of them. With PayPal acceptance, any ElateMe user will be able to donate for some wish with his PayPal account, where he has funds or set credit cards and bank accounts.

1.9.2 Whom to pay?

ElateMe system has own PayPal account for money holding during collecting for some wish. Holding, collecting and manipulating wish money is a task for our backend servers, and it is not covered in this bachelor's thesis. The actual task is to make ElateMe Android application support PayPal payments.

Our team decided to use Braintree service as a payment service provider in our ElateMe system. Braintree is a PayPal service, so it is the most convenient and preferable way to support PayPal payments. Braintree supports a lot of different payment methods besides PayPal such as Apple Pay, Android Pay, Masterpass, and others. By choosing Braintree services our team got not only PayPal payments, but also Apple Pay for iOS devices and Android Pay for Android devices.

1.10 Android Pay

1.10.1 What is Android Pay?

Mobile payment services are becoming more and more popular during last few years. The main convenience of mobile payment is that you can pay with your smartphone or smartwatch simply just by keeping them close to a payment terminal. The other handy feature of mobile payments is that you can use it for internet payments.

Android Pay is such mobile payment service developed by Google and supported by Android devices (with Android KitKat 4.4 and above).

1.10.2 Payment flow

The other important task of this thesis is to make ElateMe Android application support donation with Android Pay, which means an ability to make an internet payment with Android Pay. The flow of payment is pretty straightforward: a user chooses Android Pay as a payment method, then he selects a card he wants to pay with, then he should authenticate himself (by password or fingerprint) and finally proceed the payment. Example of such payment in some Android application where a user choosing his card is shown in Figure 1.7.

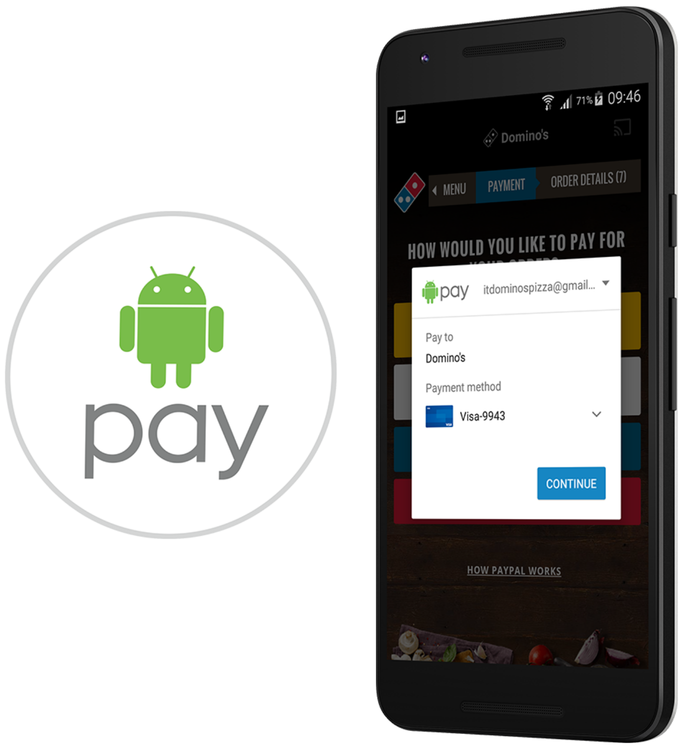


Figure 1.7: Example of Android Pay payment is some application [8]

1.11 Sharing a product from a web browser

Let's imagine the user of ElateMe service surfing the web using his smartphone, where ElateMe application installed. Supposably he finds an interesting product on some e-shop website and wants to create wish with it in ElateMe service. Of course, he could create wish manually, setting by hands all required parameters such as title, amount, image and so on. But what if our Android application would do all these routine operations for the user? Here comes sharing from a web browser.

1.11.1 Possible solution

Although it is called sharing from a web browser, actual trick is that application should be provided with webpage URL in order to recognize the potential product. So, we need to make our application receive data from other applications somehow in the Android operating system.

1.12 Requirements specification

1.12.1 Functionality

R1 New feed item layout: The goal is to implement new feed item according to a new design.

R2 Comments in the feed: The goal is to implement the latest comment for each feed item.

R3 Paging for all lists of items: A user should be able to load all of the items in series.

R4 New comments in wish detail screen: The goal is to implement new layout for comments, which should be placed at the end of the screen with detailed wish view.

R5 New screen with wish creation: Besides the primitive creation of the wish, a user should be able to choose wish deadline, visibility and fixed/any amount of donation.

R6 Sharing from a web browser: A user should be able to share any page (i.e., while surfing the web) into the app in order to create the new wish. If the product on the page will be successfully recognized, a user will be able to create a new wish with it.

R7 FIO Bank payments integration: The app should support payments via FIO Bank payment gateway. (card payments)

R8 PayPal payments integration: The app should support payments with PayPal account.

R9 Bitcoin payments: The app should support payments in bitcoins.

R10 Android Pay payments: The app should support payments via Android Pay.

R11 New notification popups: Notification popups with some important information should be implemented. When some new event takes place, a user should be informed with a corresponding small window (popup) in the application.

R12 Unread notifications indicator: The user interface should have an indication of the number of unread notifications.

R13 New notification list: List of all notifications should be reworked. New layout for the list item and click on notification should be implemented.

1.12.2 Usability

R8 App stability: High stability of the app in terms of "living" in Android OS is needed. The app should correctly handle such cases as force exiting, being in the background and returning to the foreground.

1.12.3 Reliability

R9 Security during the bank payment: Bank payments should be sufficiently secured.

1.12.4 Performance

R10 Network performance optimizations: To take measures aimed at reducing the number of network calls.

1.12.5 Supportability

R11 Clean code: The source code should be clean and understandable.

R12 Separation of concerns: Application architecture should meet this design principle.

R13 Continuous Integration and Automatic Testing: Automatic testing should be configured and deployed.

1.12.6 Constraints

R14 Minimal supported API: Minimal supported API is 19.

R15 Landscape screen orientation: No landscape layout is supported.

Chapter 2

Design

This chapter introduces the major user interface changes, describes the architecture decisions of the most important features of the application and provides the updated platform-specific model.

2.1 UI design

All user interface requirements I received from Michal Maněna or Jan Hoffman, and all my recommendation and request were discussed with them in advance before implementation.

While implementing, I tried to follow Google Material Design Guidelines [9] and use the latest approaches and modern libraries.

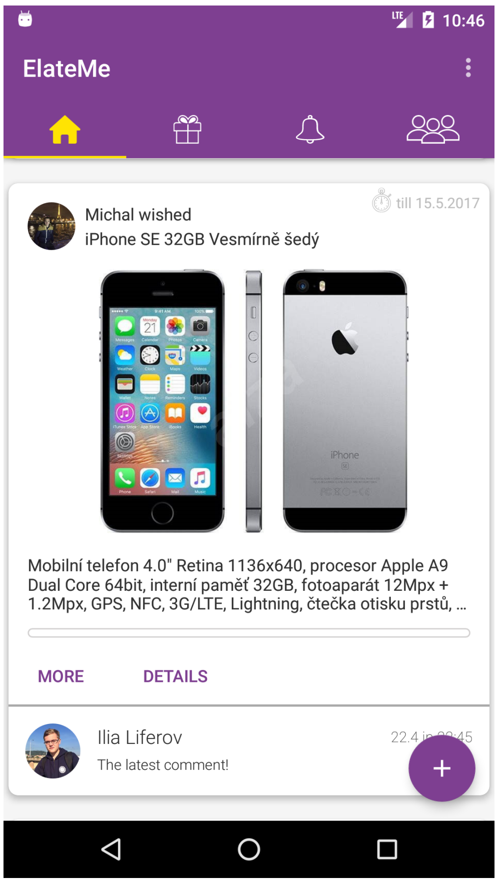
2.1.1 New feed screen

The feed screen is the first screen (except the screen with login, which is shown in case a user is not logged in yet) which is presented to a user when he opens the application. This screen contains a list of wishes, so in order to make each entry of the list outstanding and informative, I decided to use card view. Each card will have brief information about the wish, such as title, author image, wish image, collection progress and the latest comment if present.

Implemented feed screen with one entry of the list of wishes is shown in Figure 2.1.

You can see:

- facebook picture of the author and his name,
- wish deadline,
- wish title, image, and a short description,
- collection progress in form of a horizontal progress bar,



Michal wished
iPhone SE 32GB Vesmírně šedý

Mobilní telefon 4.0" Retina 1136x640, procesor Apple A9 Dual Core 64bit, interní paměť 32GB, fotoaparát 12Mpx + 1.2Mpx, GPS, NFC, 3G/LTE, Lightning, čtečka otisku prstů, ...

The latest comment!

Figure 2.1: Implemented feed screen

- “more” button, which opens the context menu of the wish,
- “details” button, which opens the new screen with details of this particular wish,
- the latest comment with author name and image, text and date.

Moreover, some edge cases of the view logic were handled:

- Deadline icon and date is not shown if the deadline of the wish is not set.
- Card changes its height based on the presence of wish image. Which means, that card is getting lower when the image of the wish is absent.
- Wish description is shown with ellipsis in order to keep the normal height of the card. Too long descriptions are shown ÑĄut and ended with three dots.

- The latest comment section is not showing in case of absence of any comment.

2.1.2 New comments in wish detail screen

The next essential requirement was the implementing comments on the screen with wish details. Jan Hoffman provided the user interface design for this. I implemented this feature by strictly following the provided design, although some behavioral improvements were made by myself.

The use cases which should be implemented were reading comments and adding comments. I added hiding comments and hiding the input field for comments as useful improvements. Important points:

- message with a number of comments,
- list of comments (showing only the latest comment when comments are hidden),
- button for collapsing the list of comments (changing its text accordingly),
- input field for making a new comment. This input field is hidden behind the bottom of the screen when the user is looking at details of the wish. As soon as the user scrolls down to the comments, input field appears with smooth animation. When the user navigates back to the details, input field hides as well. It's important to note, that text in the input field survives these hidings. If the user leaves some text in the input field without sending, then triggers hiding of the input field by scrolling up and then scrolls back down, his text will remain.

Examples of the implemented screen are shown in Figure 2.2.

2.1.3 New wish creation screen

According to domain rules and business-rules of our ElateMe system, creating a new wish requires some information, namely, wish recipient, set of allowed (for viewing and donating) users, deadline of the wish and type of donations (such as fixed amount or currency). All those things along with wish title, image and amount make up the new wish, created by a user of ElateMe system. Thus, our screen should provide a convenient and intuitive user interface for creating a new wish.

The first solution

The first solution of wish creation screen was implemented following an interactive design of the web version, which is available at website [10]. I had to

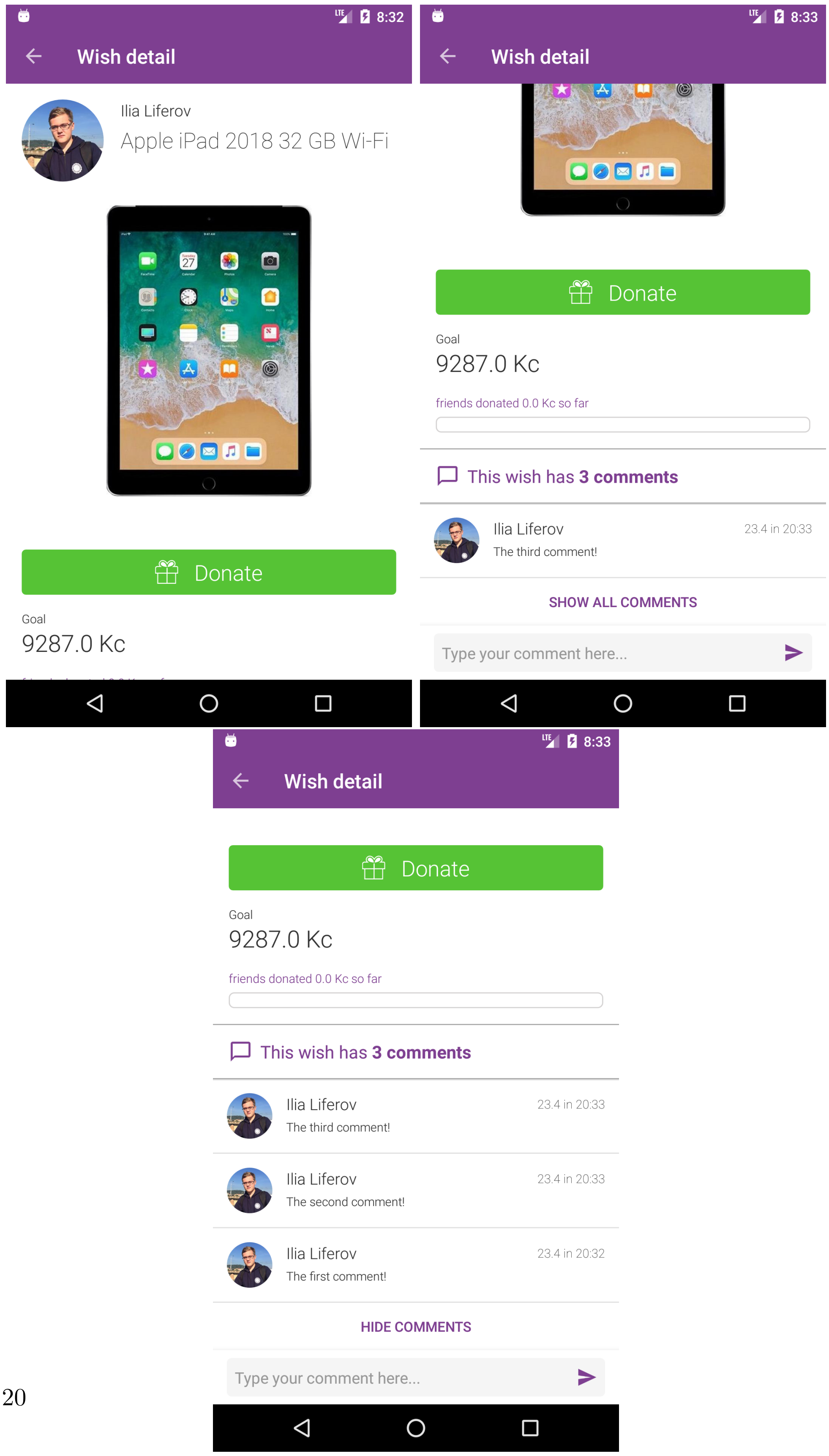


Figure 2.2: Implemented comments in the wish detail screen

use mobile mode in my web browser in order to get a mobile-looking example of the interface. The resulting screen is shown in Figure 2.3.

The problem is that this design is not adaptive to mobile devices and is made based on the version for web browsers. There are two significant disadvantages:

1. All information is concentrated in one single screen.
2. There is one fragment with tabs in the such another fragment. Since tabs can be changed by left or right swiping, such behavior with inner tabs might be ambiguous and unclear for a user.

The current solution

To make the screen of wish creation more adaptive for mobile devices and intuitive, we decided to rework it. The new screen was reworked base on Jah Hoffman's design mockups. The most important change to mention was separating the screen into the main screen and four subsequent screens. Each subsequent screen is responsible for picking required options of the new wish such as wish recipient, wish visibility, wish deadline and type of donations. You can see all those implemented screens in Figures 2.4 and 2.5. It must be noted that some user experience improvements were implemented such as picking a currency of the donation and choosing visibility among recipient friends.

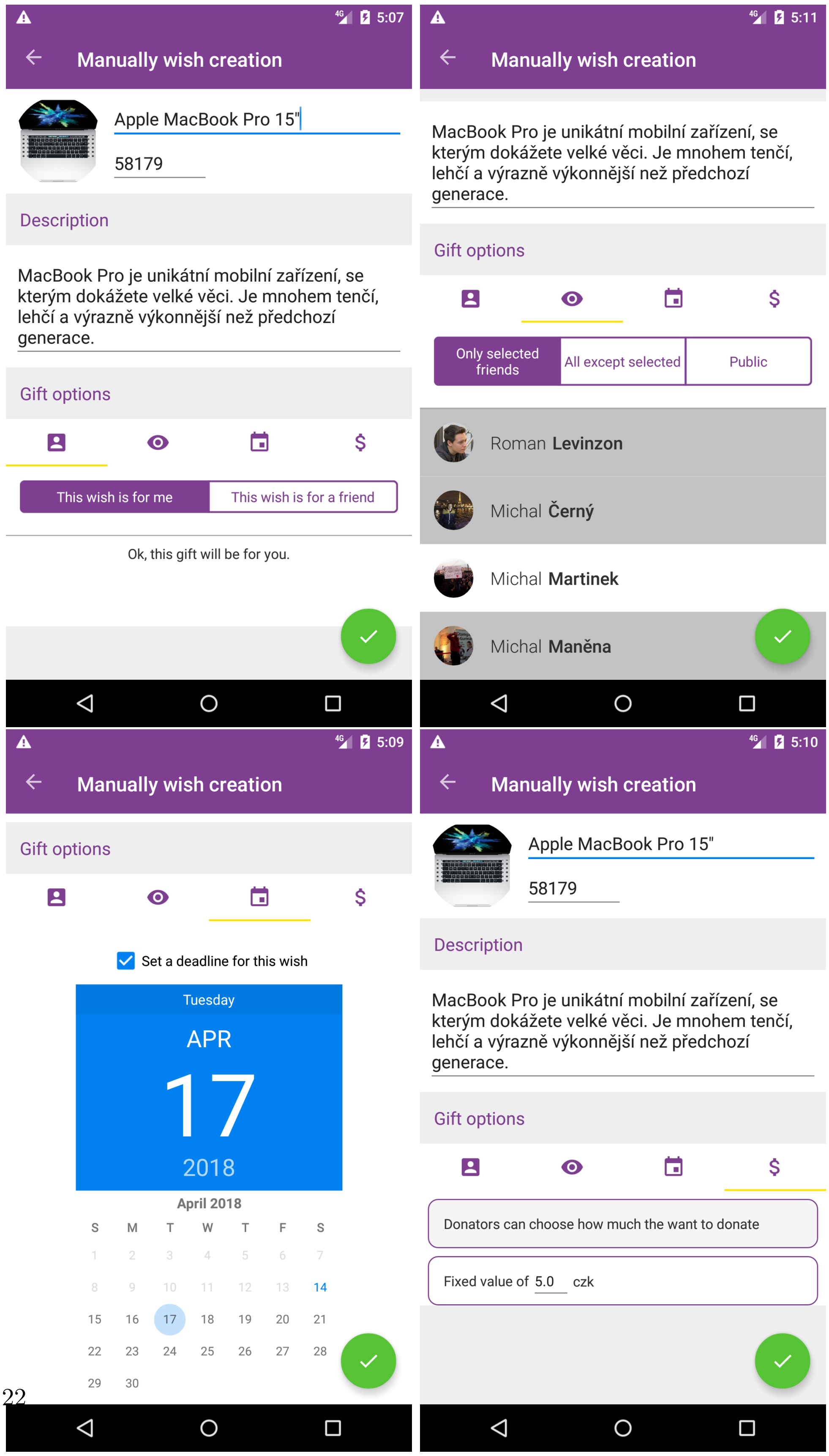
2.1.4 New popups and notifications

To implement popups and notifications we needed not only to organize service layer in order to make them update recurrently but also to design all logic of viewing the popups and notifications in our application.

The popups

According to provided UI design, all popups look almost the same way – they all have a header, a text, and an optional image. On the other hand, each type of popup has its own data, which should be presented to the user in header or text. (For example amount and image of the wish in one popup, name and author of the wish in another popup). The problem is to distinguish such different logic of what should be shown and how it should be shown from each other, trying to follow such important design principle as separation of concerns.

The solution is to place the view logic in the base class of popup view model and concentrate the content logic in the derived classes. Thus, derived class logic is responsible for determining a content of the popup and base class


5:07

Manually wish creation

Apple MacBook Pro 15"

58179

Description

MacBook Pro je unikátní mobilní zařízení, se kterým dokážete velké věci. Je mnohem tenčí, lehčí a výrazně výkonnější než předchozí generace.

Gift options

[✓ This wish is for me] This wish is for a friend

Ok, this gift will be for you.

5:11

Manually wish creation

MacBook Pro je unikátní mobilní zařízení, se kterým dokážete velké věci. Je mnohem tenčí, lehčí a výrazně výkonnější než předchozí generace.

Gift options

[✓ Only selected friends] All except selected Public

Roman **Levinzon**

Michal **Černý**

Michal **Martinek**

Michal **Maněna**

5:09

Manually wish creation

Gift options

☑ Set a deadline for this wish

Tuesday

APR

17

2018

April 2018

S	M	T	W	T	F	S
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30					

5:10

Manually wish creation

Apple MacBook Pro 15"

58179

Description

MacBook Pro je unikátní mobilní zařízení, se kterým dokážete velké věci. Je mnohem tenčí, lehčí a výrazně výkonnější než předchozí generace.

Gift options

Donators can choose how much the want to donate

Fixed value of 5.0 czk

Figure 2.3: The first version of implementation of the wish creation screen

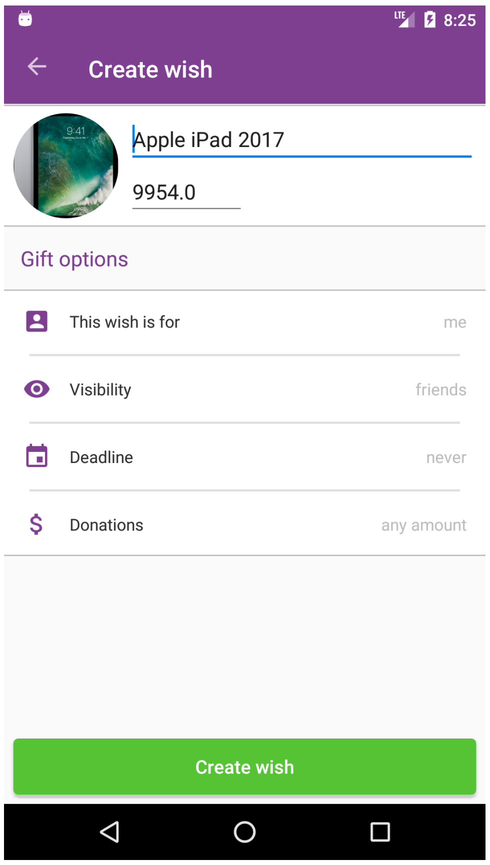


Figure 2.4: The current version of the wish creation screen (main screen)

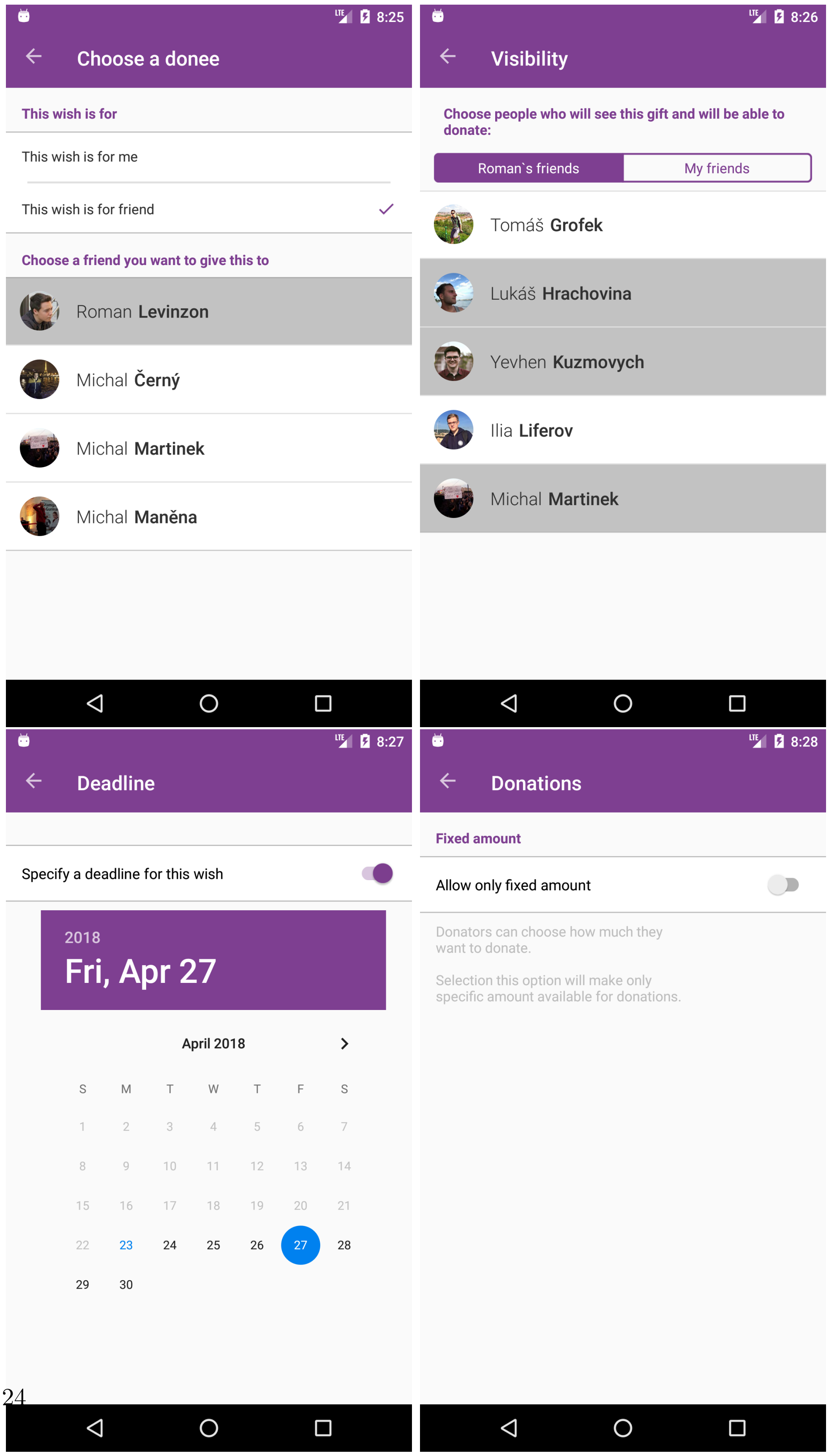
logic is responsible for showing the content properly (with all appropriate logic such as treating the absence of the popup image).

There is an example of implemented popup 2.6a and its class diagram 2.6b.

The notifications

The same solution was applied to the same problem with notifications. All view logic was placed in the base class and content logic in the derived classes.

There is an example of one item of the notification list 2.7 and its class diagram.


8:25

Choose a donee

This wish is for

This wish is for me

This wish is for friend ✓

Choose a friend you want to give this to

Roman Levinzon

Michal Černý

Michal Martinek

Michal Maněna

8:26

Visibility

Choose people who will see this gift and will be able to donate:

Roman`s friends | My friends

Tomáš Grofek

Lukáš Hrachovina

Yevhen Kuzmovych

Ilia Liferov

Michal Martinek

8:27

Deadline

Specify a deadline for this wish

2018

Fri, Apr 27

April 2018

S	M	T	W	T	F	S
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30					

8:28

Donations

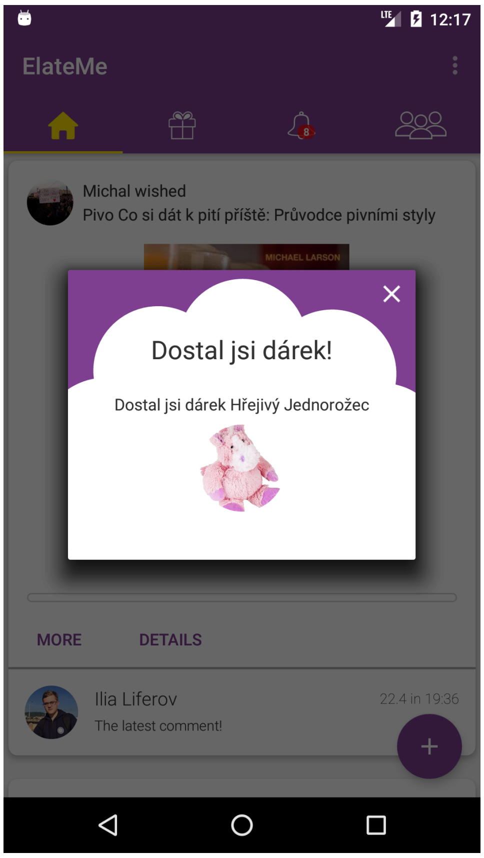
Fixed amount

Allow only fixed amount

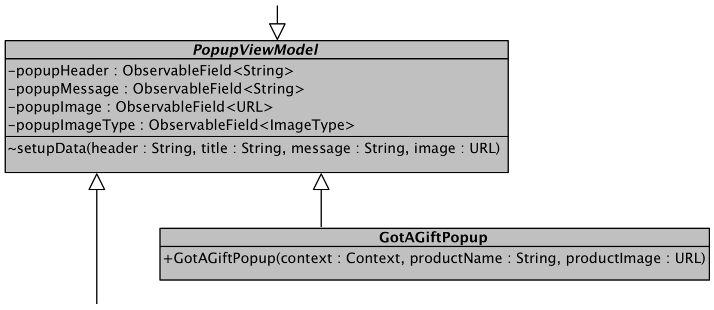
Donators can choose how much they want to donate.

Selection this option will make only specific amount available for donations.

Figure 2.5: The current version of the wish creation screen (subscreens)



(a) Example of the implemented popup


PopupViewModel
-popupHeader : ObservableField<String>
-popupMessage : ObservableField<String>
-popupImage : ObservableField<URL>
-popupImageType : ObservableField<ImageType>
~setupData(header : String, title : String, message : String, image : URL)
GotAGiftPopup
+GotAGiftPopup(context : Context, productName : String, productImage : URL)

(b) Class diagram of the implemented popup

Figure 2.6: Implemented popup and its class diagram

2.2 Application architecture

2.2.1 Service layer rework

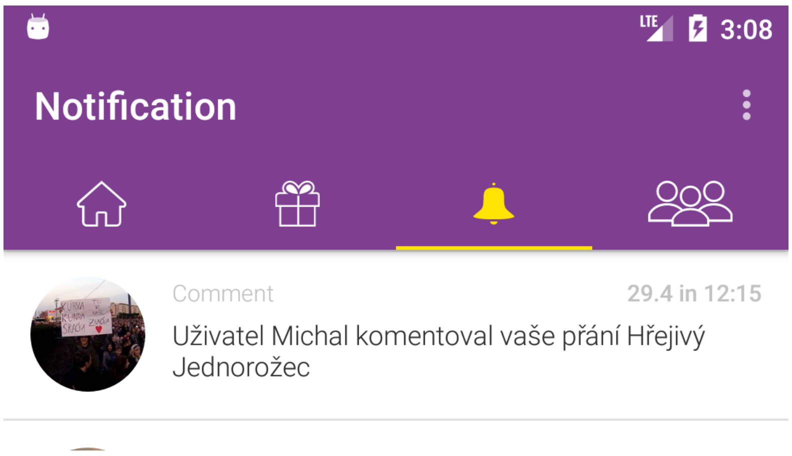
What is a service layer?

A service layer is a layer of the application software architecture, which exists between the UI layer and the data access layer. The goal of such service layer is to encapsulate business-logic of an application, to communicate with the UI and to access the needed data.

Problems with the previous realization

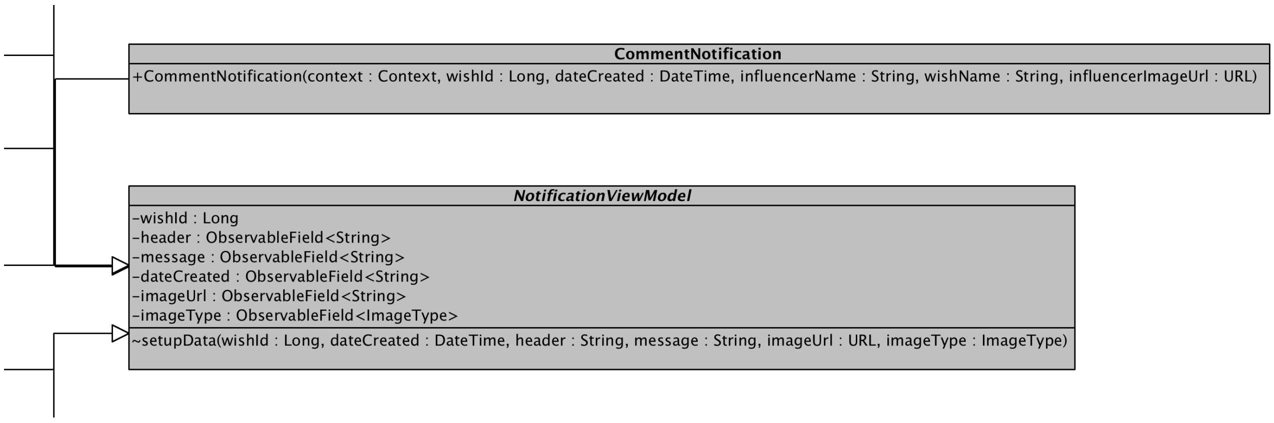
Previous organization and implementation of this important layer was inflexible and complicated in usage, and there is an explanation why.

The previous realization was based on so-called Interactor pattern which is the part of Clean Architecture pattern in Android development (you can



Uživatel Michal komentoval vaše přání Hřejivý Jednorožec

(a) Example of one item of the notification list


CommentNotification
+CommentNotification(context : Context, wishId : Long, dateCreated : DateTime, influencerName : String, wishName : String, influencerImageUrl : URL)
NotificationViewModel
-wishId : Long
-header : ObservableField<String>
-message : ObservableField<String>
-dateCreated : ObservableField<String>
-imageUrl : ObservableField<String>
-imageType : ObservableField<ImageType>
~setupData(wishId : Long, dateCreated : DateTime, header : String, message : String, imageUrl : URL, imageType : ImageType)

(b) Class diagram of one item of the notification list

Figure 2.7: One implemented item of the notification list and its class diagram

read more about it in the article [11]). There are a lot of details to explain, but I will provide a short description of the main concept of these interactors and explain, why I decided to replace them with a more appropriate solution.

Simply put, interactor is an object, which encapsulates some business-logic (like any member of service layer should do) of either strictly one use case or a particular combination of them. The other important thing to mention is that there is only one way to get the result of interactor execution – subscribe to the result. For example, previously in the ElateMe application, there was interactor for getting wish by its id, called GetWithInteractor. To use it I needed to inject this interactor object, call its execution by providing some wish id and subscribe to the result, which is wish found by provided id. To summarize, I can request service layer for some data and get the result asynchronously. It sounds like the service layer fulfills its role, but there is a pitfall over here. The last thing to mention before diving in explanations is that ElateMe Android application uses reactive extensions. It is "*an API for asynchronous programming with observable streams.*" [12] Using reactive extension helps a lot in writing asynchronous code, managing threads, process user's inputs and so on. It is a very powerful tool with the ability to convert absolutely anything (user clicks or network response) to a stream of data and combine such streams if needed. For example, we use Retrofit library, which helps us to "convert" our any HTTP request to an observable stream of data (response or error of the HTTP request).

Back to the pitfall of approach with interactors, let's describe some situation. What will we do, if we need to get some wish by its id and get all users who contributed to it at the same time? The solution that comes to mind is to use one interactor in the subscription on another interactor, which in our situation means take GetWishInteractor, subscribe to it and then use the next, GetConributedUsers, interactor in this subscription. The problem is that it is a wrong, impractical solution and furthermore huge misuse of reactive extensions.

So, there are two main problems with this approach:

1. Interactors ask, but not being asked. Each interactor needs you to provide a subscription to it instead of giving you the observable stream of data you requested.

2. We can not combine interactors, so we are forced to create the new interactor every time we need to combine some primitive use cases. GetWishInteractor is for getting a wish, GetWishWithContributedUsersInteractor is for getting wish with its contributed users, GetWishWithCachedContributedUsersInteractor is for getting a wish with cached contributed users and so on. It makes code dirty and difficult to support.

New realization

The new realization is based on dividing out service layer into sublayers:

- Data access layer. For example Retrofit service, which returns observable for an HTTP request.

- Data provider layer. There is no actual need for this sublayer for now because the only one source of data for out ElateMe application, for now, is network. It will be useful in case of using several sources of data, for example, network and some local storage. Then this sublayer will be responsible for managing those sources.

- Service layer. This is a sublayer, where all application business-logic is concentrated in. For example, here we can combine observables and enrich our wish with its contributed users.

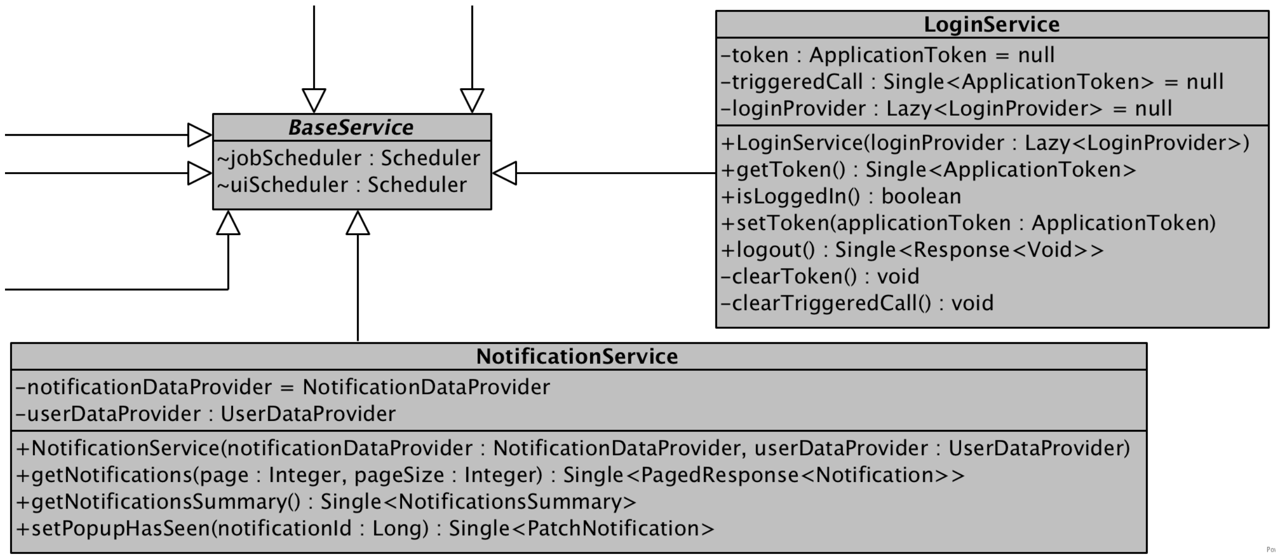

BaseService
~jobScheduler : Scheduler
~uiScheduler : Scheduler
LoginService
-token : ApplicationToken = null
-triggeredCall : Single<ApplicationToken> = null
-loginProvider : Lazy<LoginProvider> = null
+LoginService(loginProvider : Lazy<LoginProvider>)
+getToken() : Single<ApplicationToken>
+isLoggedIn() : boolean
+setToken(applicationToken : ApplicationToken)
+logout() : Single<Response<Void>>
-clearToken() : void
-clearTriggeredCall() : void
NotificationService
-notificationDataProvider = NotificationDataProvider
-userDataProvider : UserDataProvider
+NotificationService(notificationDataProvider : NotificationDataProvider, userDataProvider : UserDataProvider)
+getNotifications(page : Integer, pageSize : Integer) : Single<PagedResponse<Notification>>
+getNotificationsSummary() : Single<NotificationsSummary>
+setPopupHasSeen(notificationId : Long) : Single<PatchNotification>

Figure 2.8: Service layer class diagram (a part of class diagram)

Thus, the problem of combining observables is solved. Every member of the UI layer is able now to obtain some needed observable streams of data, combine them the way it wants and use all the power of reactive extensions. You can see a part of the class diagram of the service sublayer in Figure 2.8.

2.2.2 Handling Android application lifecycle

The problem

The ElateMe system uses Facebook as a login system. Connecting using Facebook SDK ElateMe Android application allows users to sign with their Facebook accounts. So, to make the application communicate with our ElateMe backend server, a user must be authorized through Facebook systems. During the authorization of a user, the application obtains specific token which is required for communication with ElateMe backend server. Solovev Georgii's implementation is pretty straightforward – when a user opens the application, the first screen he sees is the login screen, which checks whether the user is logged in and either let him proceed to the app or showing him welcome screen with a login button. Also, the application obtains necessary token if the user is already logged in. This implementation is perfect unless there is not any other entry point to the application besides mentioned login screen.

The problem was how Android OS manages its applications. If a user opens an app on his Android phone, then minimize it and lets it run in the background, Android OS can easily unload minimized application from memory if it is necessary. Then, if the user decides to open the minimized application back, Android OS will try to recreate this application in the state the user saw

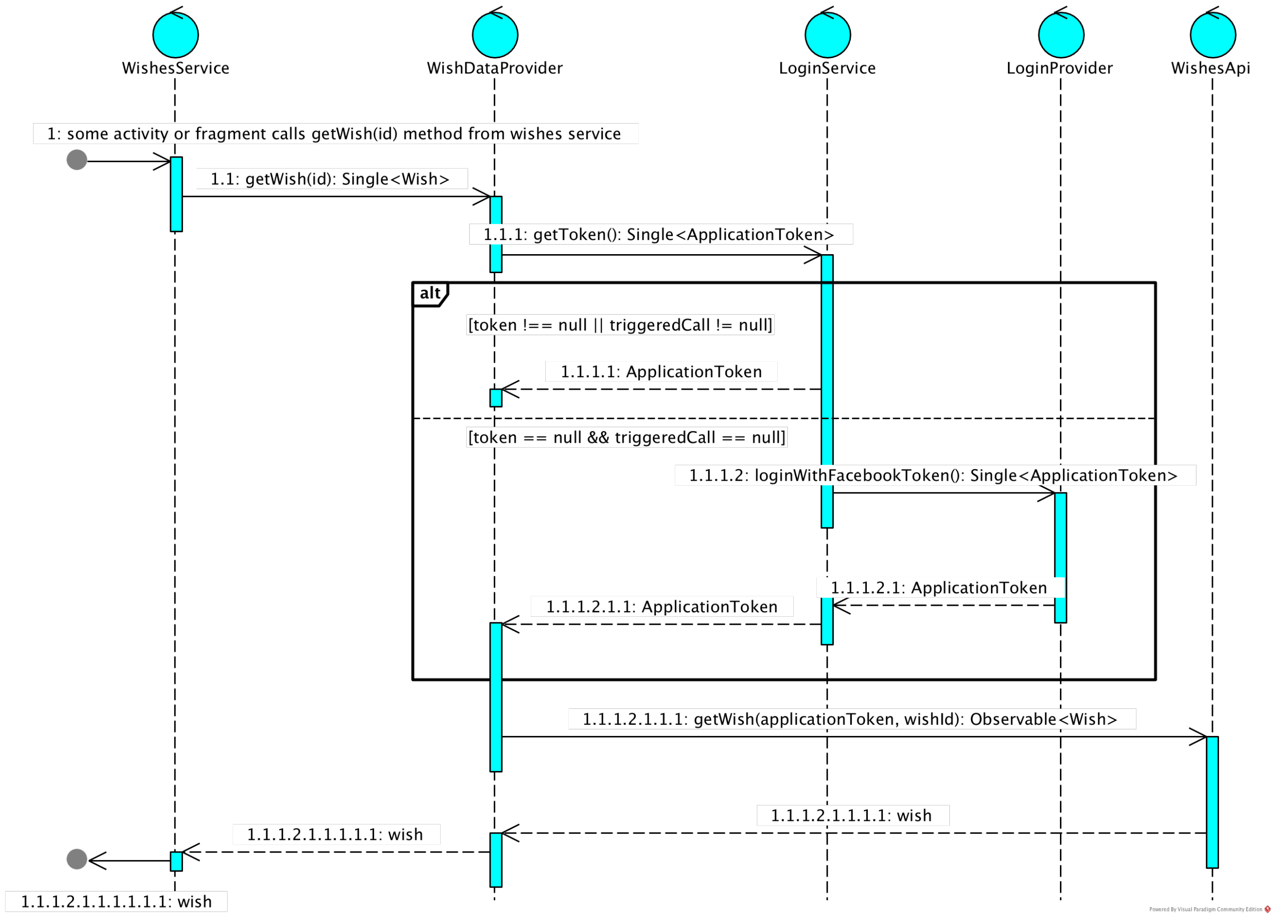

WishesService
WishDataProvider
LoginService
LoginProvider
WishesApi
1: some activity or fragment calls getWish(id) method from wishes service
1.1: getWish(id): Single<Wish>
1.1.1: getToken(): Single<ApplicationToken>
alt
[token !== null || triggeredCall != null]
1.1.1.1: ApplicationToken
[token == null && triggeredCall == null]
1.1.1.2: loginWithFacebookToken(): Single<ApplicationToken>
1.1.1.2.1: ApplicationToken
1.1.1.2.1.1: ApplicationToken
1.1.1.2.1.1.1: getWish(applicationToken, wishId): Observable<Wish>
1.1.1.2.1.1.1.1: wish
1.1.1.2.1.1.1.1.1: wish
1.1.1.2.1.1.1.1.1.1: wish

Figure 2.9: Getting wish from the network with token management (sequence diagram)

it the last time before minimizing. When this happens to our ElateMe application, needed application token is missings in our restored application, and all requests to the backend server won't succeed (will return 401 Unauthorized).

The solution

I decided to change the way our application obtains authorization token. Instead of getting it on the first screen, it should be got when the first request which required this token is triggered. To implement this, I created login service for obtaining and caching this important token. So any method of data provider sublayer (described in section 2.2.1) now asks this login service for the token before making the actual request. The login service checks if the token is cached and either returns it or obtains the new one. Detailed sequence diagram of this process is shown in Figure 2.9.

2.2.3 Popups and notifications

This subsection describes architecture details of the popups and notifications in the ElateMe Android application.

The main distinction between this feature and the others is that popups and notifications should be managed by some Android application component, which is able to perform background operations. There are Android services for such cases.

"*A Service is an application component that can perform long-running operations in the background, and it doesn't provide a user interface. Another application component can start a service, and it continues to run in the background even if the user switches to another application.*" [13]

There are the three different types of services:

- Foreground service, which "*performs some operation that is noticeable to the user. For example, an audio app would use a foreground service to play an audio track.*" [13]
- Background service, which "*performs an operation that isn't directly noticed by the user. For example, if an app used a service to compact its storage, that would usually be a background service.*" [13]
- Bound service, to which an application component can be bound. "*A bound service runs only as long as another application component is bound to it. Multiple components can bind to the service at once, but when all of them unbind, the service is destroyed.*" [13]

Bound services perfectly fit our needs, because we need to update and show popups and notifications only when the user sees and uses our application. We can bind it when the application is in the foreground and unbind when it goes off. Such service will be used for recurrent updates of new popups and counter of unread notifications from the ElateMe backend server. Notification service will publish via event bus an event, indicating updates of popups or unread notifications counter and all interested components of the application such activities or fragments will receive such event and handle it on their own. Thus, main activity (one which contains four tabs) will update its notification counter (notifications tab bell icon) and all activities will be able to show new popups. Sequence diagram of this process is shown in Figure 2.10.

2.2.4 Payments

Before describing the architecture of each type of payment, we need to describe some general things, which are common for all payments.

All payments can be made only from the screen with details of the wish. If a user opens wish to which he is able to donate, he will see donate button. By clicking on it, the user will be asked to choose amount and donation

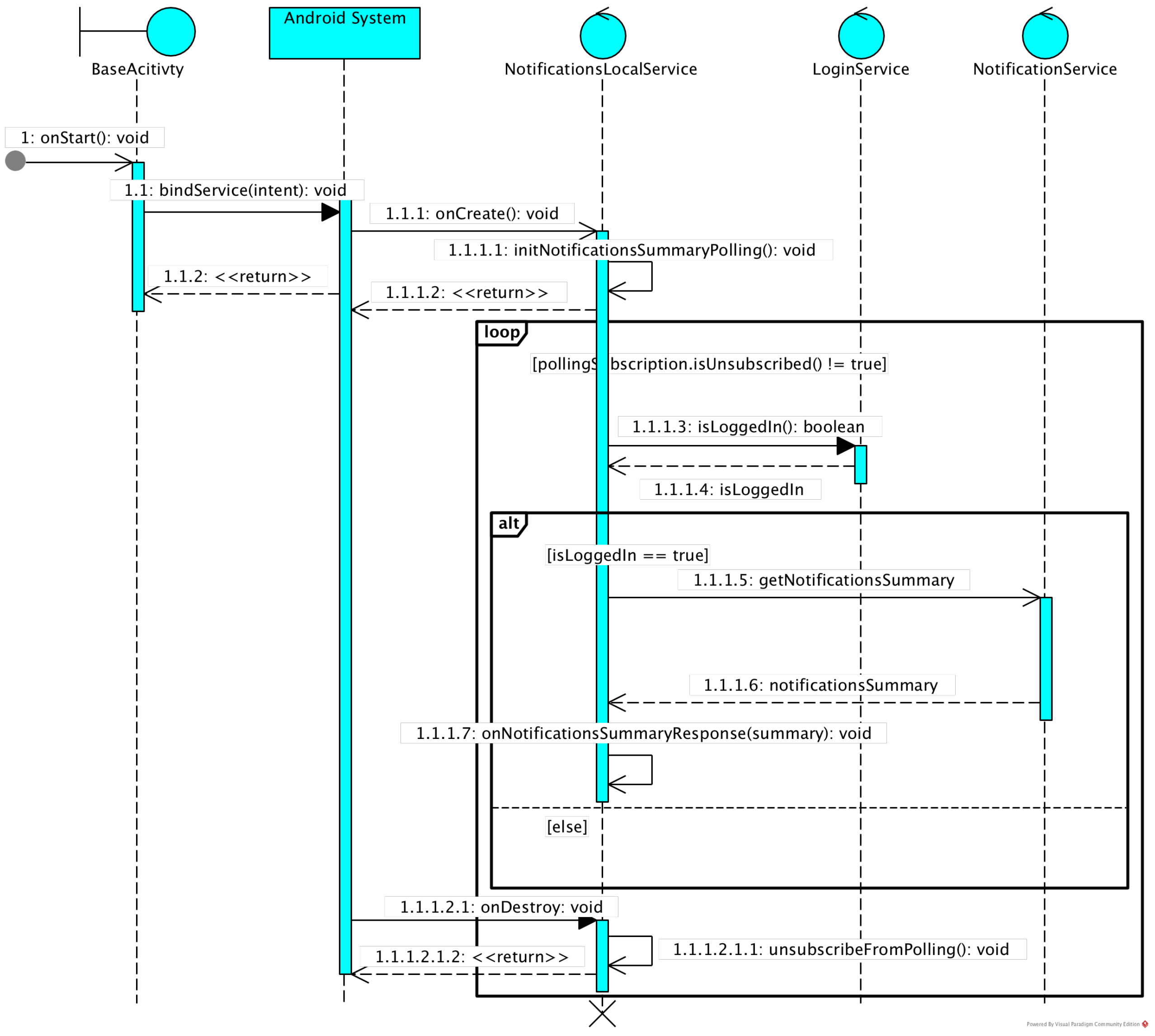

Android System
BaseAcitivty
NotificationsLocalService
LoginService
NotificationService
1: onStart(): void
1.1: bindService(intent): void
1.1.1: onCreate(): void
1.1.1.1: initNotificationsSummaryPolling(): void
1.1.2: <<return>>
1.1.1.2: <<return>>
loop
[pollingSubscription.isUnsubscribed() != true]
1.1.1.3: isLoggedIn(): boolean
1.1.1.4: isLoggedIn
alt
[isLoggedIn == true]
1.1.1.5: getNotificationsSummary
1.1.1.6: notificationsSummary
1.1.1.7: onNotificationsSummaryResponse(summary): void
[else]
1.1.1.2.1: onDestroy: void
1.1.1.2.1.1: unsubscribeFromPolling(): void
1.1.1.2.1.2: <<return>>
Powered By Visual Paradigm Community Edition

Figure 2.10: Polling popups and notifications from the backend server (sequence diagram)

method (bank card, Bitcoins, PayPal or Android Pay). Then the user initializes payment by accepting some donation method. This process is described by sequence diagram, which can be found in Figure 2.11. The screenshot of picking donation amount and method is shown in Figure 2.11.

All kinds of payments are described separately in the next parts.

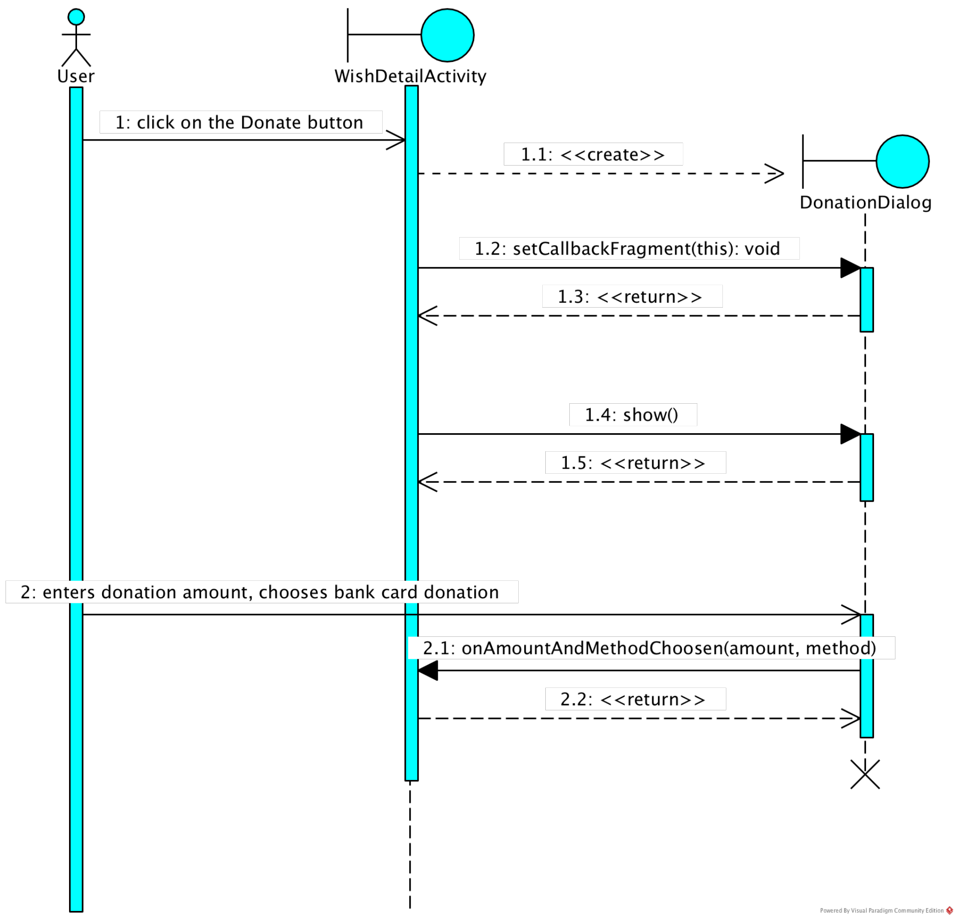

User
WishDetailActivity
1: click on the Donate button
1.1: <<create>>
DonationDialog
1.2: setCallbackFragment(this): void
1.3: <<return>>
1.4: show()
1.5: <<return>>
2: enters donation amount, chooses bank card donation
2.1: onAmountAndMethodChoosen(amount, method)
2.2: <<return>>

Figure 2.11: Picking donation options by a user (sequence diagram)

2.2.4.1 Bank payments

As I mentioned earlier in the analysis chapter, our team decided to change the approach of realization of the bank payments. The new bank payments will not differ from the analogical payments on web site in the web browser. Since the goal is to give the user ability to visit the Fio bank webpage with bank gateway, I decided to use Android webview. WebView is "*a view that displays web pages.*" [14] "*You can roll your own web browser or simply display some online content within your Activity. It uses the WebKit rendering engine* *to display web pages and includes methods to navigate forward and backward through a history, zoom in and out, perform text searches and more.”* [14]

The flow of payment with a bank card is pretty simple – all our application need is to obtain URL for payment gateway webpage from the ElateMe backend server and show this webpage to the user. When the user enters the amount and chooses donation with a bank card, the ElateMe application obtains URL and showing Fio bank gateway in a webview.

This flow is described by sequence diagram which is shown in figure 2.12.

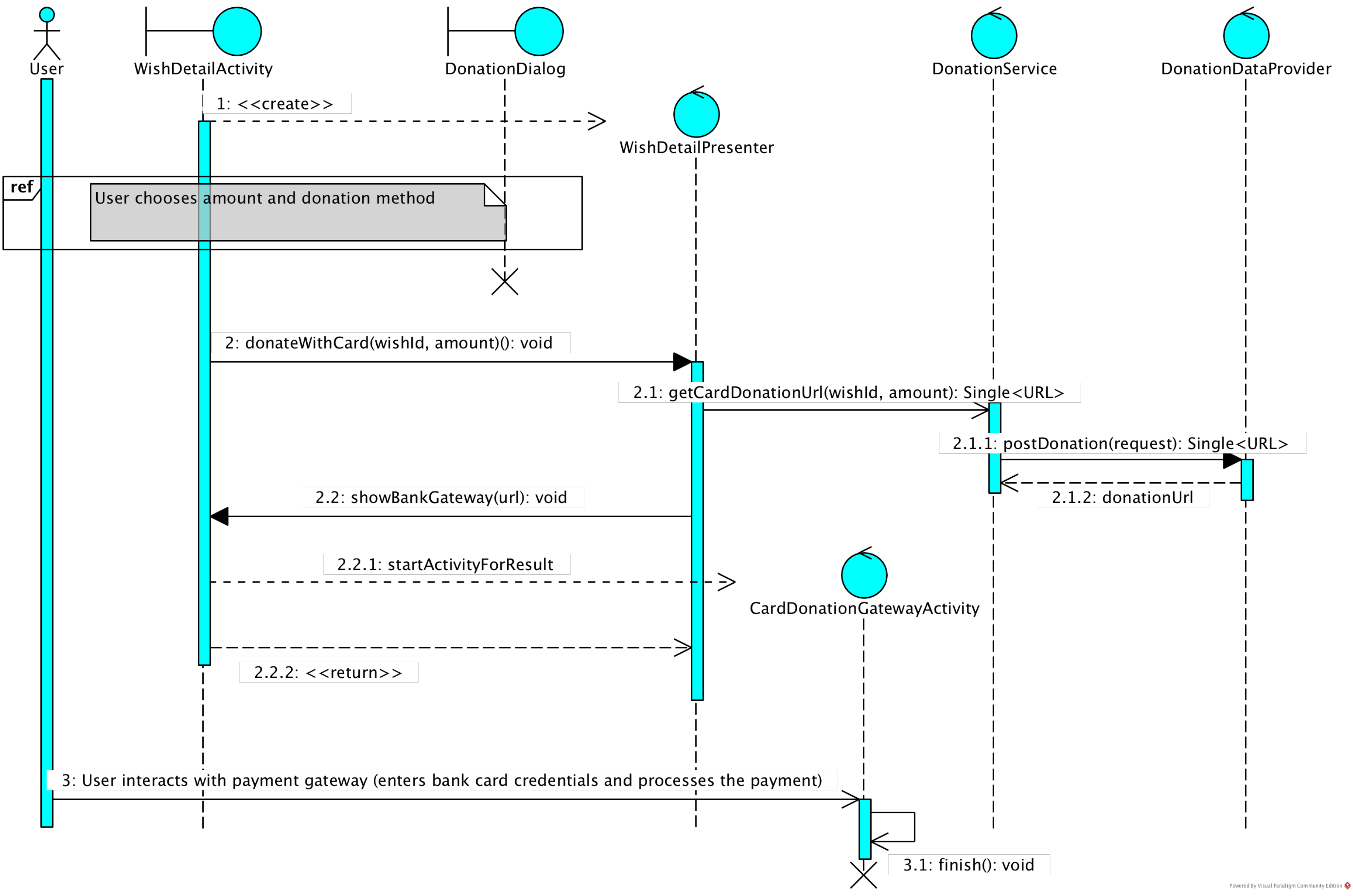

User
WishDetailActivity
DonationDialog
DonationService
DonationDataProvider
1: <<create>>
WishDetailPresenter
ref
User chooses amount and donation method
2: donateWithCard(wishId, amount)(): void
2.1: getCardDonationUrl(wishId, amount): Single<URL>
2.1.1: postDonation(request): Single<URL>
2.2: showBankGateway(url): void
2.1.2: donationUrl
2.2.1: startActivityForResult
CardDonationGatewayActivity
2.2.2: <<return>>
3: User interacts with payment gateway (enters bank card credentials and processes the payment)
3.1: finish(): void

Figure 2.12: Donation with bank card (sequence diagram)

2.2.4.2 Bitcoin payments

As was mentioned in the analysis section of this thesis, ElateMe system uses its own account for all types of payments. It is an essential thing in organizing refunds and donations from several users.

Since our team decided to use BitPay service for managing bitcoin payments, the goal of this thesis is to implement the integration of BitPay payments into our Android application. BitPay supports over 40 integrations for accepting payments in bitcoins such as shopping carts, plugins, donation systems and libraries for today's most popular programming languages.

Official documentation says, that there is Android SDK for integration BitPay payments into an Android application. This SDK "*allows our application to create an invoice quickly, show the user an option to pay, and track the status of the payment.*" [15] This is the one which I used for BitPay integration.

According to the documentation for BitPay Android SDK, the first thing we needed to do is to create client token, which is used by the SDK internally for the purpose of identifying our ElateMe system account. This client token is a kind of address for newly created BitPay invoices. The token can be generated via the web interface of the ElateMe BitPay account, so I asked our product owner to provide it for me. In speaking of security, this client token is used only for creating invoices for our ElateMe account, so we don't have to worry about storing the token in a secure way.

The second step is to set up SDK correctly for creating new invoices. This is a pretty simple step, all we need is to instantiate BitPayAndroid class and provide it with client token. During this step, we can specify what server our application should use for BitPay payments. There are two options – test server and production server. We decided to use the test server for now in order to test BitPay payments without manipulating with real bitcoins. Switching to production server can be done by changing a couple of lines in the configuration file of our Android application.

Once we have our SDK set up, we can create invoices everytime a user wants to donate to a wish in bitcoins. First of all, we need to instantiate Invoice class by providing it with donation amount, currency and webhook URL. (Webhook URL is the HTTP callback, which is used to track a state of the invoice. As soon as the state of the invoice changed, BitPay sends the updated state to this URL address.) Then, we need to send a newly created invoice to the BitPay server. There is a special method for it in the BitPay SDK, so we do not have to handle this by ourselves.

Once the invoice is successfully sent, we can show a special screen to the user, which contains bitcoin address (with QR code), expiration date of the invoice and button with proceeding to the BitPay wallet. BitPay SDK provides this screen, so there is no need to implement it.

Thus, with the help of the BitPay Android SDK, we can easily integrate

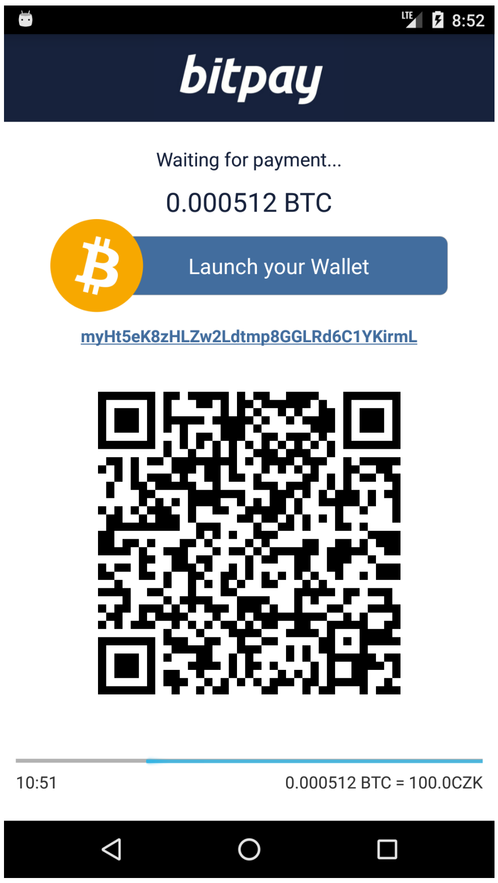


Figure 2.13: BitPay payment screen in the ElateMe application

BitPay payments into our application. Screenshot of bitcoin payment in the ElateMe Android application is shown in 2.13. Corresponding sequence diagram of the payment flow is shown in 2.14.

2.2.4.3 PayPal/Android Pay payments

Since our team chose Braintree service as a payment provider for PayPal and Android Pay payments, the goal of this thesis is to integrate Braintree payments into the application.

First of all, it is important to describe Braintree payment flow from the client point of view. There are three steps in the flow:

1. A client (Android application) must obtain a client token from our ElateMe backend server. This token identifies the user and is used for Braintree SDK initialization.

2. The user chooses a payment method (PayPal or Android Pay) on a

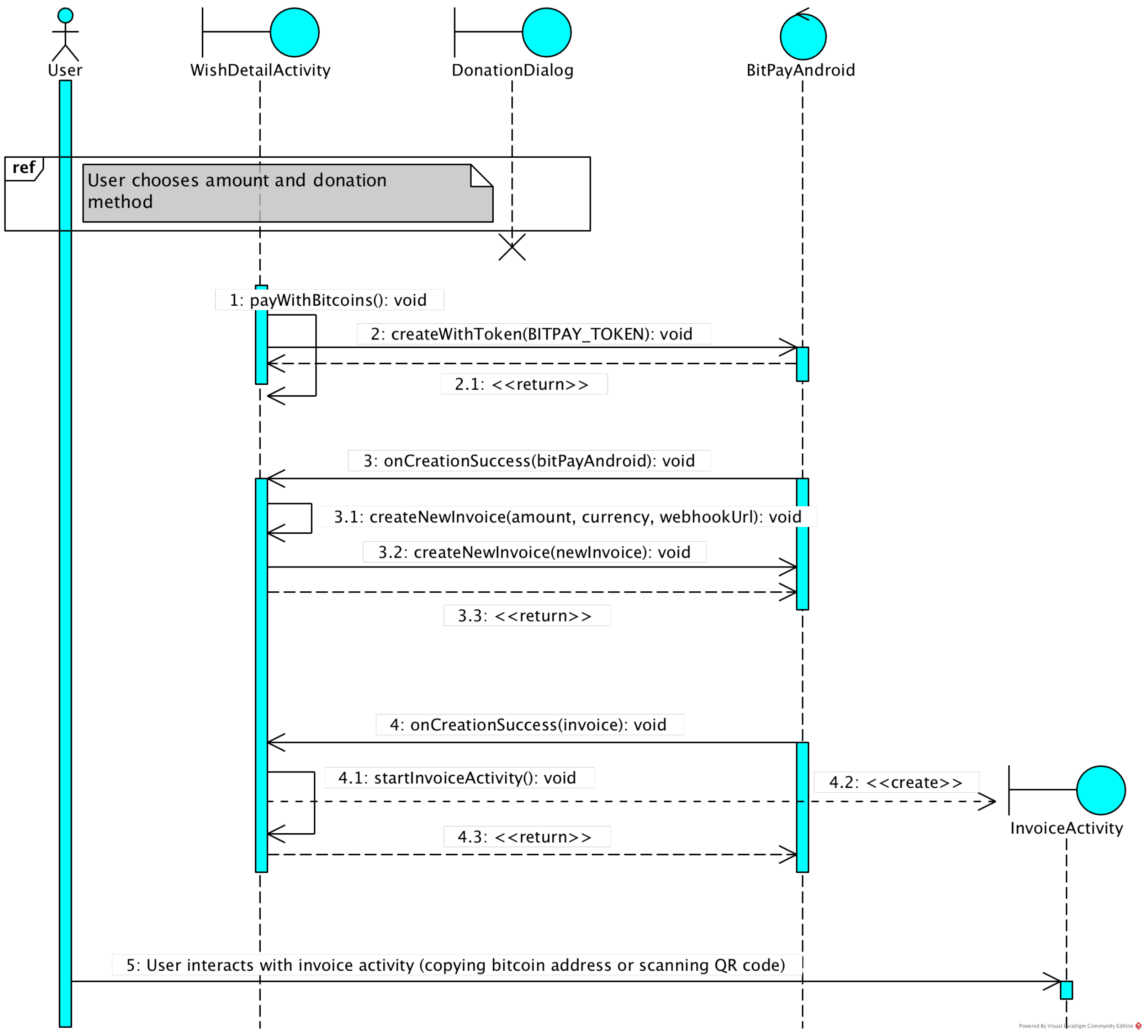

User
WishDetailActivity
DonationDialog
BitPayAndroid
ref
User chooses amount and donation method
1: payWithBitcoins(): void
2: createWithToken(BITPAY_TOKEN): void
2.1: <<return>>
3: onCreationSuccess(bitPayAndroid): void
3.1: createNewInvoice(amount, currency, webhookUrl): void
3.2: createNewInvoice(newInvoice): void
3.3: <<return>>
4: onCreationSuccess(invoice): void
4.1: startInvoiceActivity(): void
4.2: <<create>>
InvoiceActivity
4.3: <<return>>
5: User interacts with invoice activity (copying bitcoin address or scanning QR code)

Figure 2.14: Donation with bitcoins via BitPay service (sequence diagram)

special Braintree Drop-In UI and proceeds the payment. The Drop-in is shown in Figure 2.15.

3. Braintree SDK finishes the payment by making a request to Braintree servers.

4. When payment is finished, our client code should make a request to our backend server in order to inform the server about finished payment. The client-side flow is shown in an appropriate sequence diagram in a

Figure 2.16.

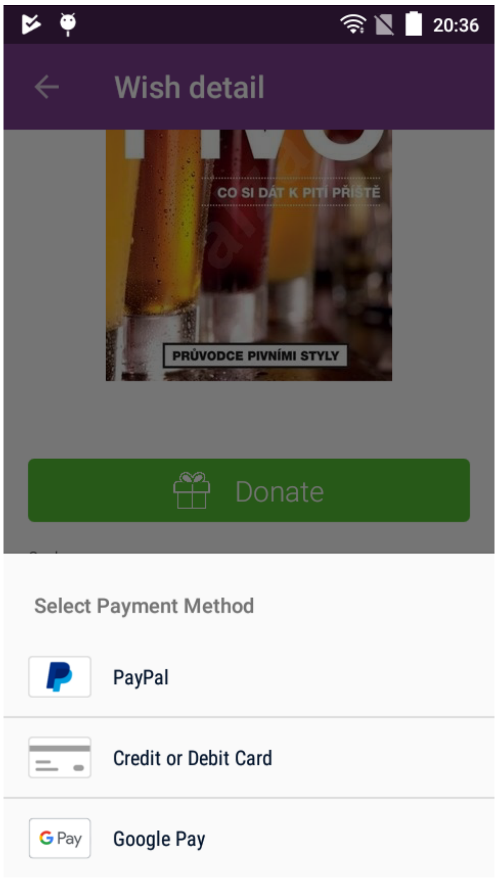


Figure 2.15: Braintree Drop-in picker for payment method

According to the Braintree's documentation and integration guide, our Android application must define a special intent filter. It is important when a user wants to pay via PayPal. If the user chooses PayPal payment method in the Drop-in, Braintree SDK will redirect the user to a web browser for proceeding his payment. Thus, the intent filter defines kind of entry point in our Android application and is required for returning from the opened web browser back to the application.

Speaking of payments via Android Pay, there are no any additional steps to get it done. Everything is handled either by Braintree SDK or Android Pay (integrated into Android OS if supported).

Thanks to the perfect Braintree's integration guide integration into an Android application is clear and easy. All I needed to do was handle a couple of network requests to our ElateMe backend server and setup Drop-in.

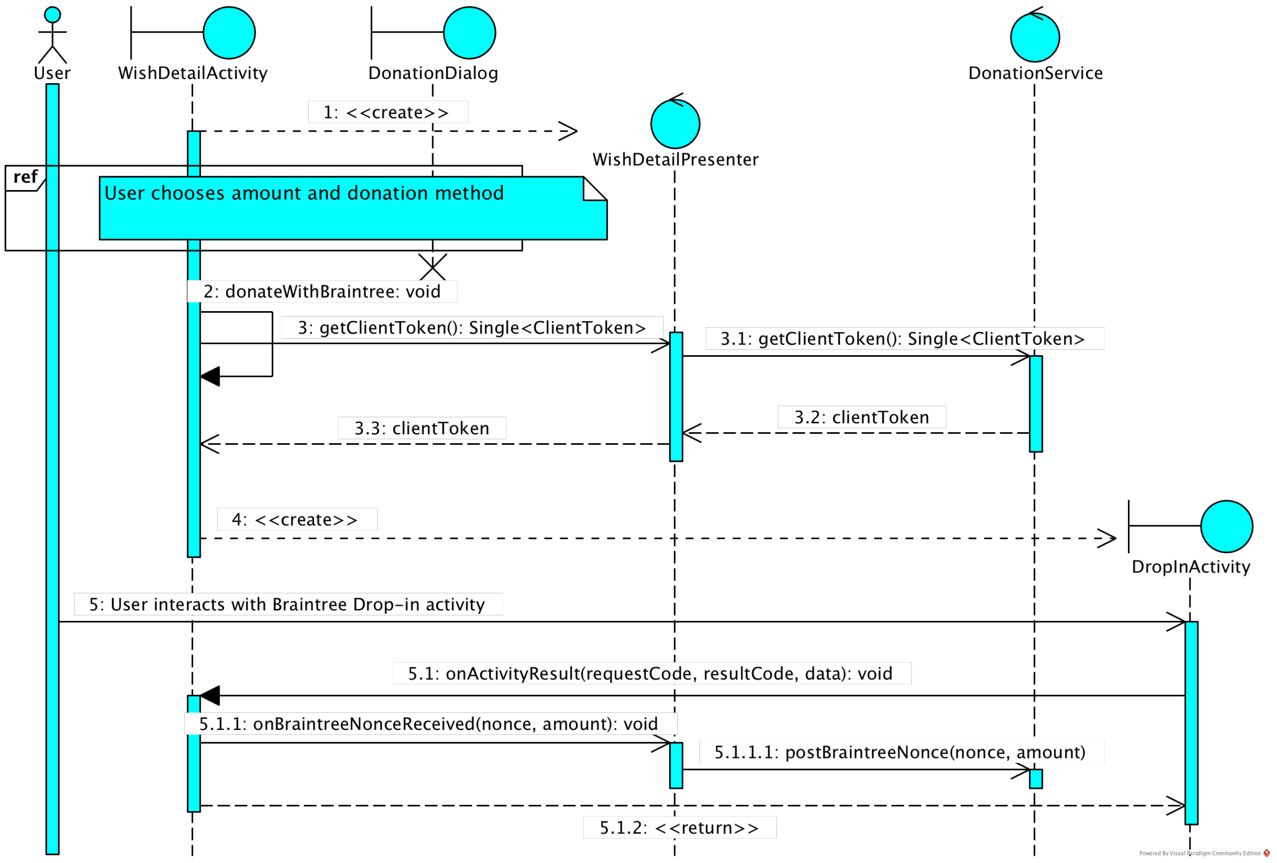

User
WishDetailActivity
DonationDialog
DonationService
WishDetailPresenter
DropInActivity
ref
User chooses amount and donation method
1: <<create>>
2: donateWithBraintree: void
3: getClientToken(): Single<ClientToken>
3.1: getClientToken(): Single<ClientToken>
3.2: clientToken
3.3: clientToken
4: <<create>>
5: User interacts with Braintree Drop-in activity
5.1: onActivityResult(requestCode, resultCode, data): void
5.1.1: onBraintreeNonceReceived(nonce, amount): void
5.1.1.1: postBraintreeNonce(nonce, amount)
5.1.2: <<return>>

Figure 2.16: Donation with the Braintree service (sequence diagram)

2.2.5 Sharing from a web browser

As we described earlier in the analysis section, our application should be able to obtain URL of a webpage, where some product should be recognized. Research showed that there exists Google recommended approach, which is based on Android Intents [16] and Intent Filters [16]. As a result, our application may be available in sharing menu of some applications including a web browser. Example of such sharing menu is shown in Figure 2.17. The algorithm which is dealing with recognition from a web page is placed at the backend server, and its description and details are not in the scope of this bachelor's thesis. "*An Intent is a messaging object you can use to request an action from another app component.*" [16]

There are two types of intents:

- "*Explicit intents specify which application will satisfy the intent, by supplying either the target app's package name or a fully-qualified component class name. You'll typically use an explicit intent to start a com-*

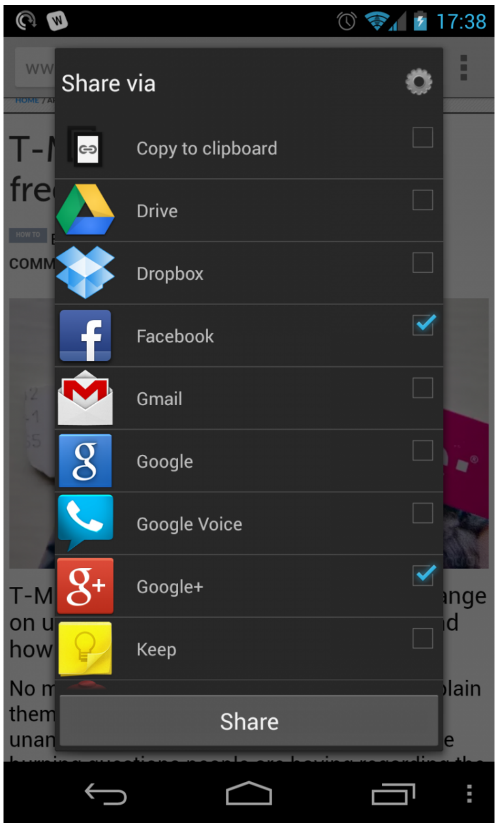


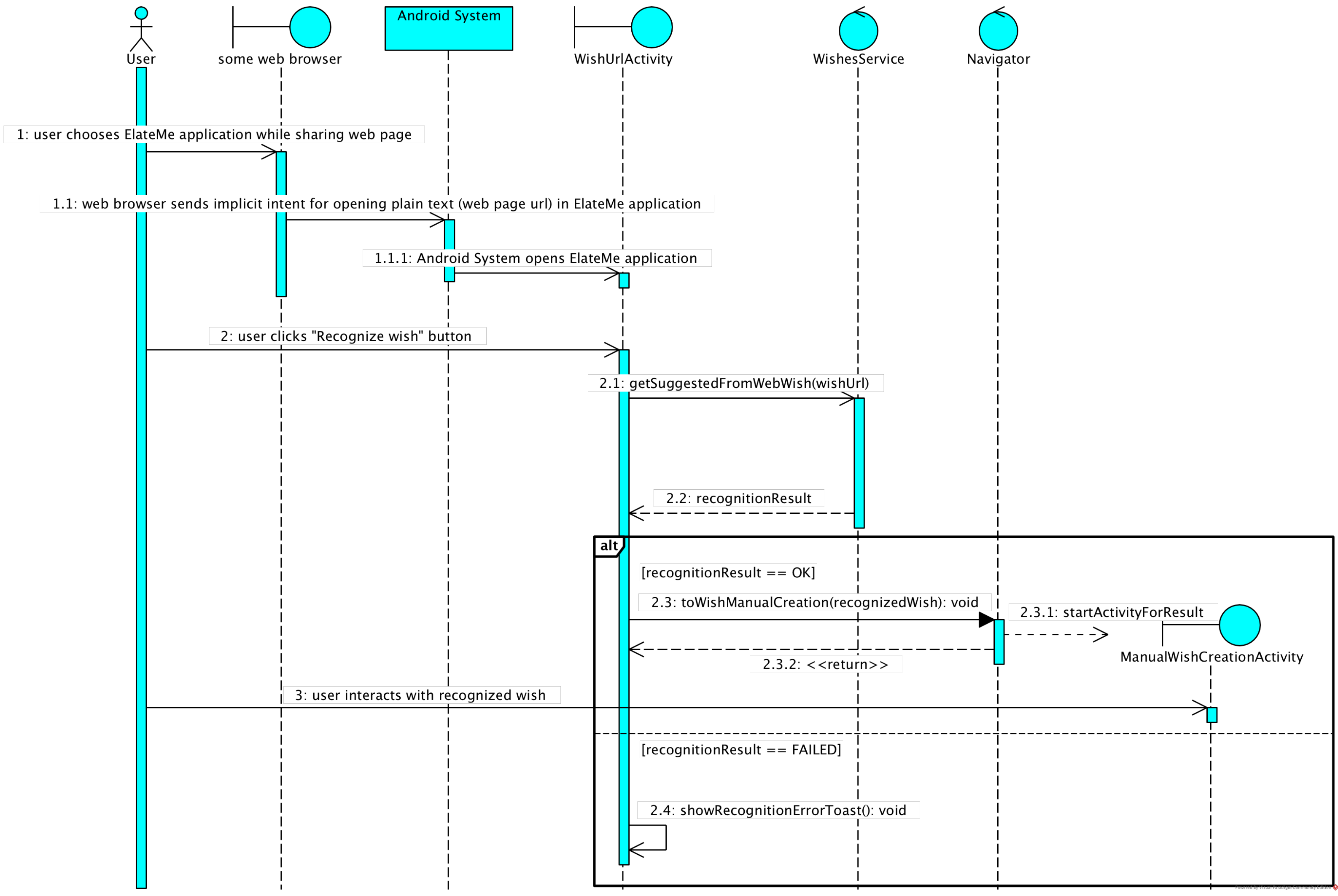
Figure 2.17: Example of sharing menu in web browser [17]

ponent in your own app, because you know the class name of the activity or service you want to start." [16]

- "*Implicit intents do not name a specific component, but instead declare a general action to perform, which allows a component from another app to handle it.*" [16]

In our case, we need implicit intent. Our application must register the intent filter in order to receive implicit intents. An intent filter is a set of rules applying to some activity (android application screen) or other application components (such as services). These rules describe how our application will "answer" if other application "asks." In our case, we chose "android.intent.action.SEND" as intent action (because we need to make our application available via sharing menu) and "text/plain" as intent data type (because an URL of the webpage is just a string).

The next step is to handle received URL webpage string. To do this, our application checks whether some text was provided to it. If it is provided and this text is a valid URL, the application will request our backend server for wish recognition. The user will be redirected to the wish creation screen with filled in data of recognized product (such as title, amount and product image). This process described by the sequence diagram which is shown in Figure 2.18.



User
some web browser
Android System
WishUrlActivity
WishesService
Navigator
1: user chooses ElateMe application while sharing web page
1.1: web browser sends implicit intent for opening plain text (web page url) in ElateMe application
1.1.1: Android System opens ElateMe application
2: user clicks "Recognize wish" button
2.1: getSuggestedFromWebWish(wishUrl)
2.2: recognitionResult
alt
[recognitionResult == OK]
2.3: toWishManualCreation(recognizedWish): void
2.3.1: startActivityForResult
ManualWishCreationActivity
2.3.2: <<return>>
3: user interacts with recognized wish
[recognitionResult == FAILED]
2.4: showRecognitionErrorToast(): void

Figure 2.18: Sharing product from a web browser (sequence diagram)

2.3 Platform-specific model updates

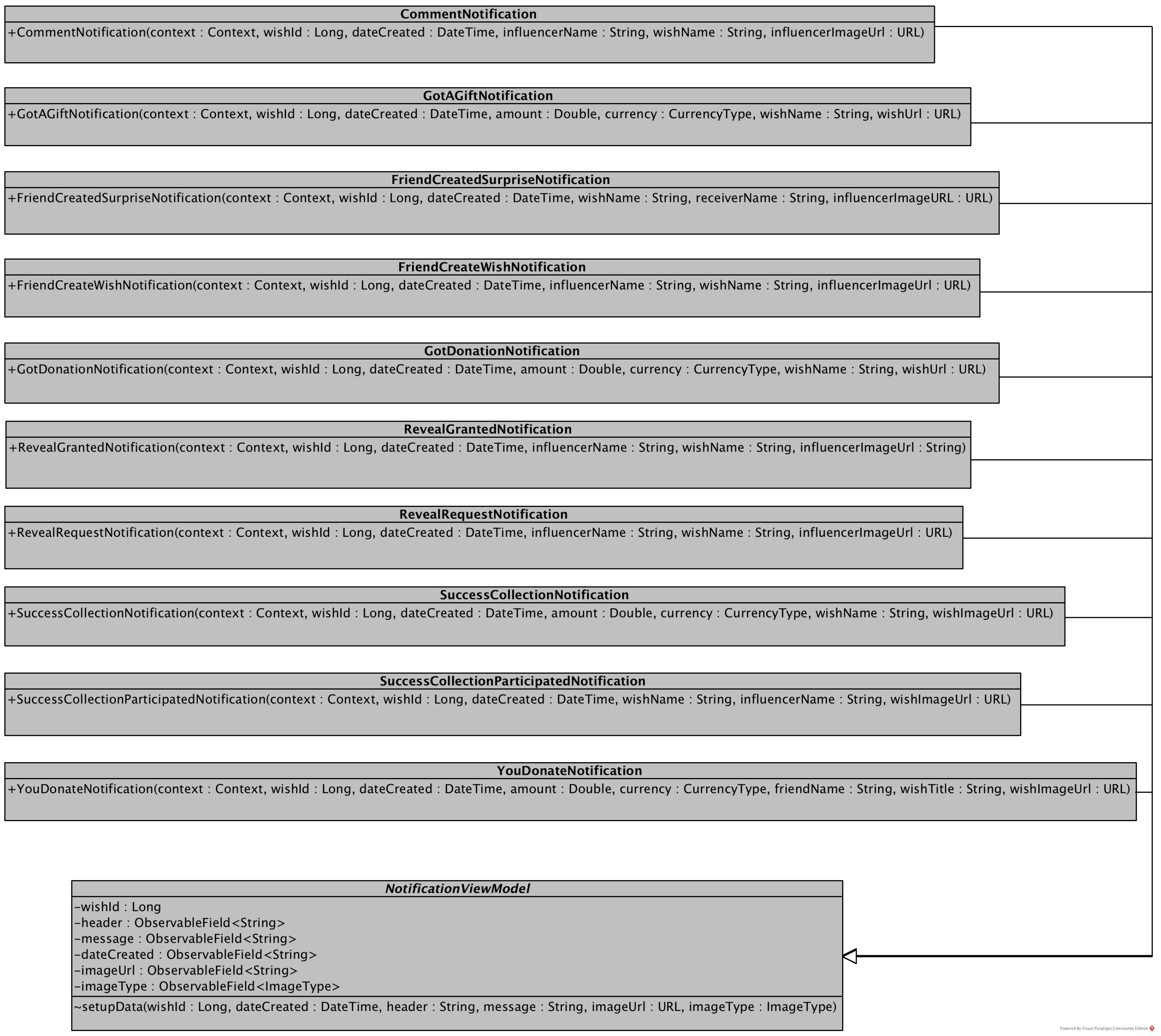

CommentNotification
+CommentNotification(context : Context, wishId : Long, dateCreated : DateTime, influencerName : String, wishName : String, influencerImageUrl : URL)
GotAGiftNotification
+GotAGiftNotification(context : Context, wishId : Long, dateCreated : DateTime, amount : Double, currency : CurrencyType, wishName : String, wishUrl : URL)
FriendCreatedSurpriseNotification
+FriendCreatedSurpriseNotification(context : Context, wishId : Long, dateCreated : DateTime, wishName : String, receiverName : String, influencerImageURL : URL)
FriendCreateWishNotification
+FriendCreateWishNotification(context : Context, wishId : Long, dateCreated : DateTime, influencerName : String, wishName : String, influencerImageUrl : URL)
GotDonationNotification
+GotDonationNotification(context : Context, wishId : Long, dateCreated : DateTime, amount : Double, currency : CurrencyType, wishName : String, wishUrl : URL)
RevealGrantedNotification
+RevealGrantedNotification(context : Context, wishId : Long, dateCreated : DateTime, influencerName : String, wishName : String, influencerImageUrl : String)
RevealRequestNotification
+RevealRequestNotification(context : Context, wishId : Long, dateCreated : DateTime, influencerName : String, wishName : String, influencerImageUrl : URL)
SuccessCollectionNotification
+SuccessCollectionNotification(context : Context, wishId : Long, dateCreated : DateTime, amount : Double, currency : CurrencyType, wishName : String, wishImageUrl : URL)
SuccessCollectionParticipatedNotification
+SuccessCollectionParticipatedNotification(context : Context, wishId : Long, dateCreated : DateTime, wishName : String, influencerName : String, wishImageUrl : URL)
YouDonateNotification
+YouDonateNotification(context : Context, wishId : Long, dateCreated : DateTime, amount : Double, currency : CurrencyType, friendName : String, wishTitle : String, wishImageUrl : URL)
NotificationViewModel
-wishId : Long
-header : ObservableField<String>
-message : ObservableField<String>
-dateCreated : ObservableField<String>
-imageUrl : ObservableField<String>
-imageType : ObservableField<ImageType>
~setupData(wishId : Long, dateCreated : DateTime, header : String, message : String, imageUrl : URL, imageType : ImageType)

Figure 2.19: Notification view models of the application (full class diagram)

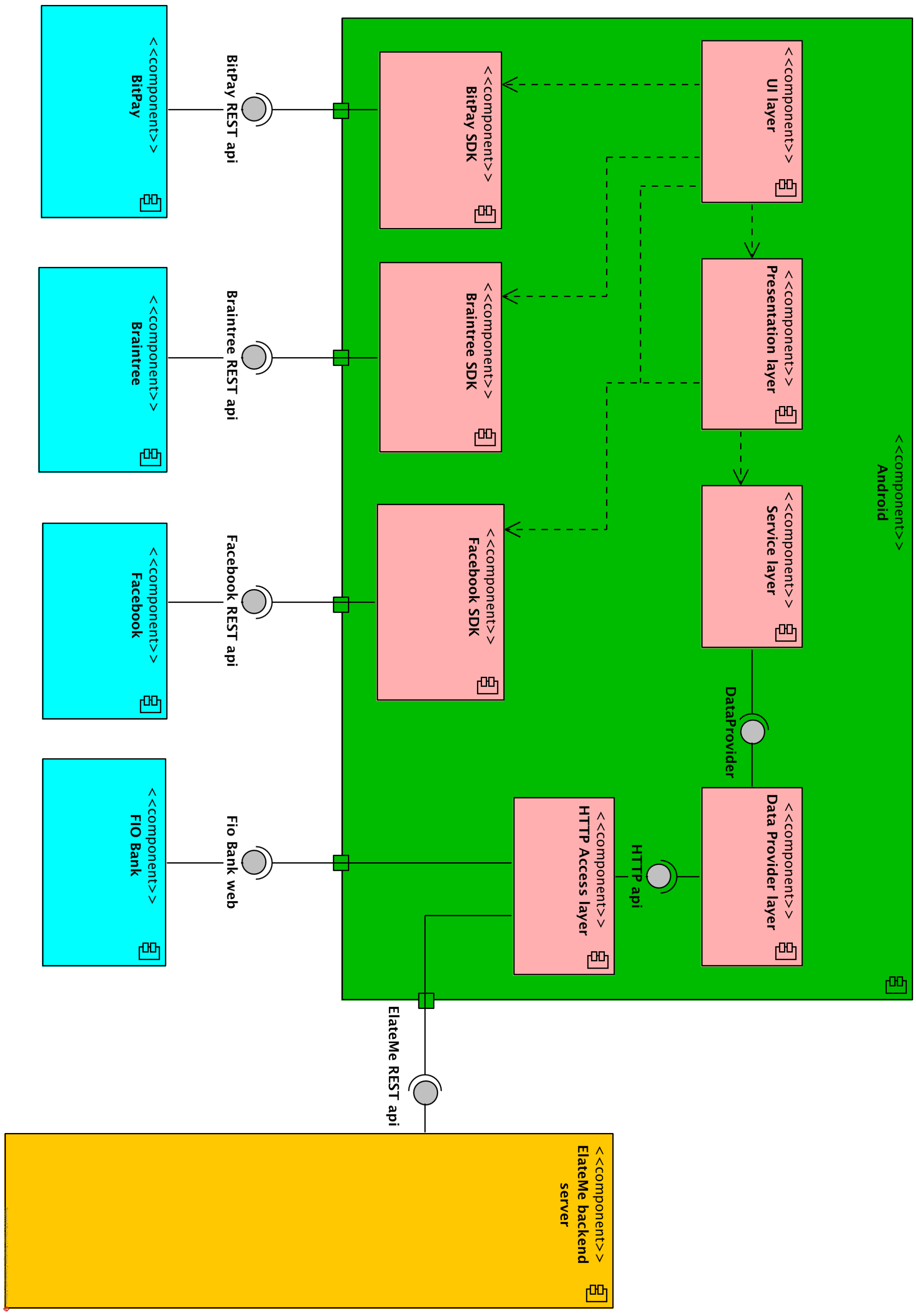


Figure 2.20: Component diagram of the application

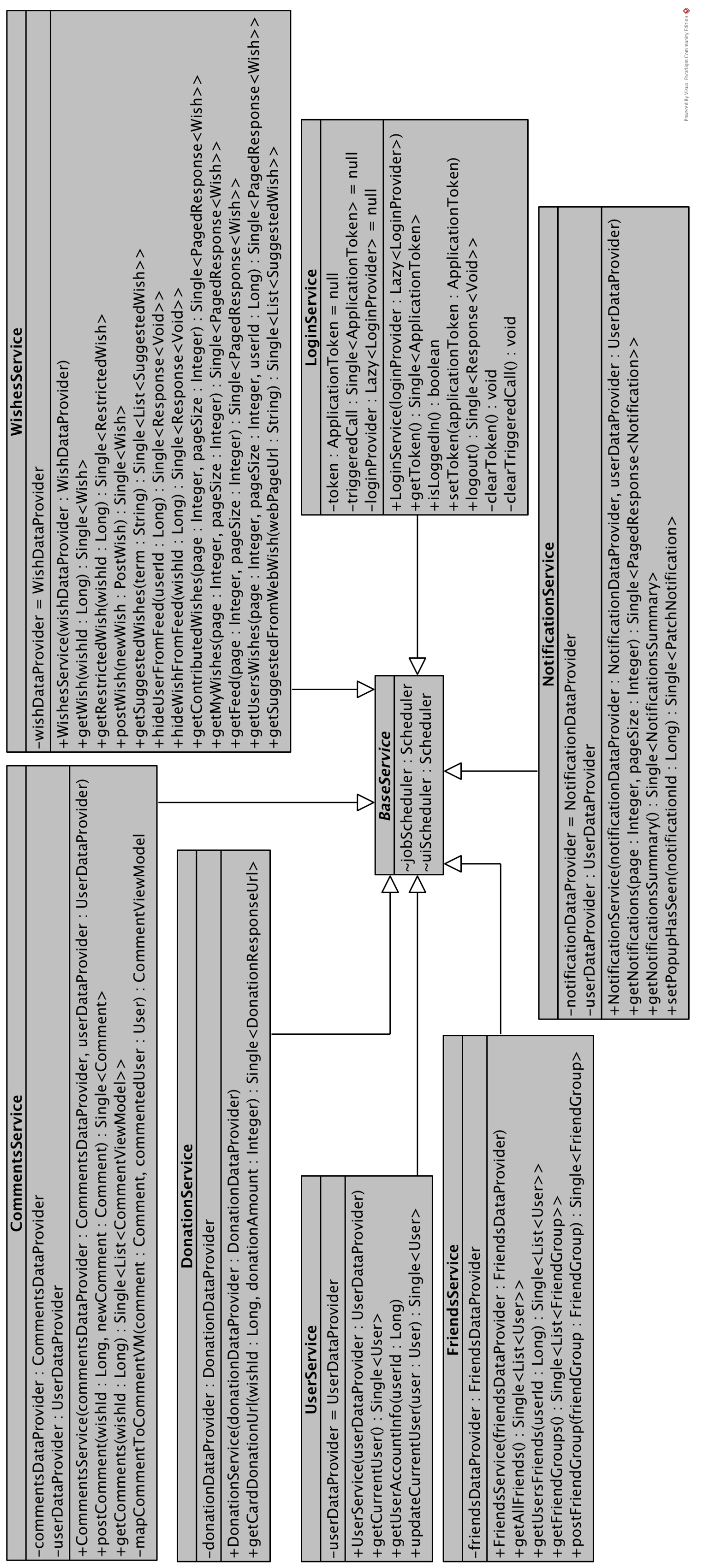

WishesService
-wishDataProvider = WishDataProvider
+WishesService(wishDataProvider : WishDataProvider)
+getWish(wishId : Long) : Single<Wish>
+getRestrictedWish(wishId : Long) : Single<RestrictedWish>
+postWish(newWish : PostWish) : Single<Wish>
+getSuggestedWishes(term : String) : Single<List<SuggestedWish>>
+hideUserFromFeed(userId : Long) : Single<Response<Void>>
+hideWishFromFeed(wishId : Long) : Single<Response<Void>>
+getContributedWishes(page : Integer, pageSize : Integer) : Single<PagedResponse<Wish>>
+getMyWishes(page : Integer, pageSize : Integer) : Single<PagedResponse<Wish>>
+getFeed(page : Integer, pageSize : Integer) : Single<PagedResponse<Wish>>
+getUsersWishes(page : Integer, pageSize : Integer, userId : Long) : Single<PagedResponse<Wish>>
+getSuggestedFromWebWish(webPageUrl : String) : Single<List<SuggestedWish>>
LoginService
-token : ApplicationToken = null
-triggeredCall : Single<ApplicationToken> = null
-loginProvider : Lazy<LoginProvider> = null
+LoginService(loginProvider : Lazy<LoginProvider>)
+getToken() : Single<ApplicationToken>
+isLoggedIn() : boolean
+setToken(applicationToken : ApplicationToken)
+logout() : Single<Response<Void>>
-clearToken() : void
-clearTriggeredCall() : void
NotificationService
-notificationDataProvider = NotificationDataProvider
-userDataProvider : UserDataProvider
+NotificationService(notificationDataProvider : NotificationDataProvider, userDataProvider : UserDataProvider)
+getNotifications(page : Integer, pageSize : Integer) : Single<PagedResponse<Notification>>
+getNotificationsSummary() : Single<NotificationsSummary>
+setPopupHasSeen(notificationId : Long) : Single<PatchNotification>
BaseService
~jobScheduler : Scheduler
~uiScheduler : Scheduler
CommentsService
-commentsDataProvider : CommentsDataProvider
-userDataProvider : UserDataProvider
+CommentsService(commentsDataProvider : CommentsDataProvider, userDataProvider : UserDataProvider)
+postComment(wishId : Long, newComment : Comment) : Single<Comment>
+getComments(wishId : Long) : Single<List<CommentViewModel>>
-mapCommentToCommentVM(comment : Comment, commentedUser : User) : CommentViewModel
DonationService
-donationDataProvider : DonationDataProvider
+DonationService(donationDataProvider : DonationDataProvider)
+getCardDonationUrl(wishId : Long, donationAmount : Integer) : Single<DonationResponseUrl>
UserService
-userDataProvider = UserDataProvider
+UserService(userDataProvider : UserDataProvider)
+getCurrentUser() : Single<User>
+getUserAccountInfo(userId : Long)
+updateCurrentUser(user : User) : Single<User>
FriendsService
-friendsDataProvider : FriendsDataProvider
+FriendsService(friendsDataProvider : FriendsDataProvider)
+getAllFriends() : Single<List<User>>
+getUsersFriends(userId : Long) : Single<List<User>>
+getFriendGroups() : Single<List<FriendGroup>>
+postFriendGroup(friendGroup : FriendGroup) : Single<FriendGroup>
Powered By Visual Paradigm Community Edition

Figure 2.21: Service layer (full class diagram)

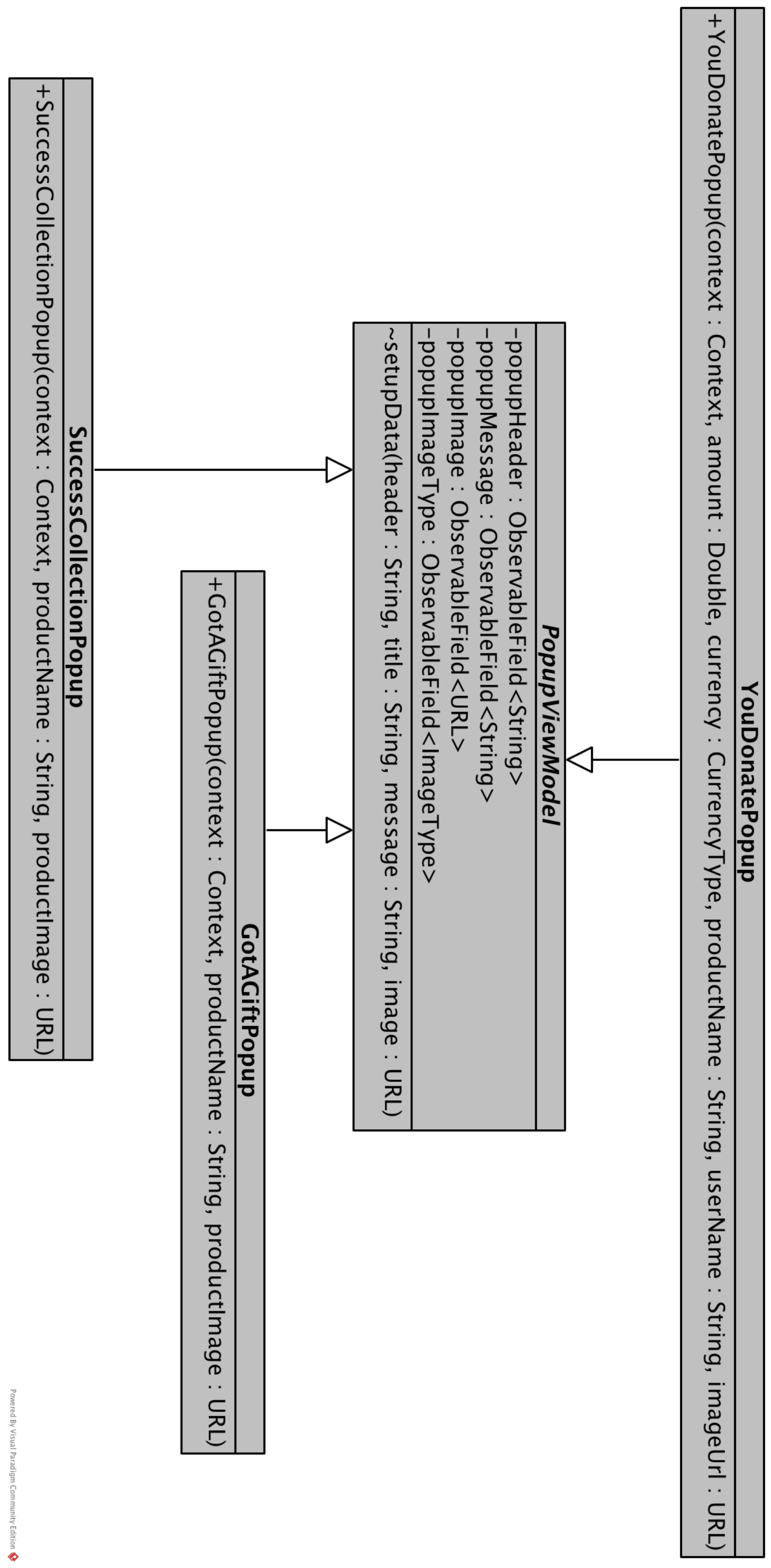

YouDonatePopup
+YouDonatePopup(context : Context, amount : Double, currency : CurrencyType, productName : String, userName : String, imageUrl : URL)
PopupViewModel
-popupHeader : ObservableField<String>
-popupMessage : ObservableField<String>
-popupImage : ObservableField<URL>
-popupImageType : ObservableField<ImageType>
~setupData(header : String, title : String, message : String, image : URL)
GotAGiftPopup
+GotAGiftPopup(context : Context, productName : String, productImage : URL)
SuccessCollectionPopup
+SuccessCollectionPopup(context : Context, productName : String, productImage : URL)
Powered By Visual Paradigm Community Edition

Figure 2.22: Popup view models of the application (full class diagram)

Chapter 3

Implementation

3.1 Code organization

3.1.1 Package organization (by feature)

Inspired by the article [18], I decided to reorganize source code of our application. Since source code is written in Java, code organization is based on packages. Earlier it was organized by layer. For example, there was a package for adapters, a package for activities, a package for fragments and so on. But the problem is, that working on one feature you need to have all related components (for example, screen, which is represented by one activity, two fragments, and several corresponding adapters). So, organization by feature is more convenient for developers. "*Packaging stuff together by what it is, and not by what it does, will only make you jump 10 times to the place you are looking for.*" [18]

Structure:

1. **dagger** – all classes related to Dagger 2, such as components and modules.
2. **data.api** – all interfaces, describing ElateMe backend API.
3. **data.api.model** – all data classes (models), related only to the data.API (such as server response model and so on).
4. **data.entities** – all data classes (models), which is used by service sublayer (classes which business logic manipulates with).
5. **data.interface** – all service layer interfaces.
6. **data.implementation** – all implementations of service layer interfaces
7. **domain.service** – all implementation of service layer (contains all business logic).

8. **presentation.base** – base classes, which are widely used in the presentation layer.
9. **presentation.component** – component classes, which are widely used in the presentation layer.
10. **presentation.splash** – all classes, related to the splash screen of the application.
11. **presentation.main** – all classes, related to the main screens of the application.
12. **presentation.main.wishes/settings/notifications/ friends/feed/createwish** – all classes, related to corresponding screens of the application.

3.1.2 Git workflow

"*Git is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency.*" [19]. Git manages the source code of ElateMe Android application project.

I developed new features in separate branches (new branch for each feature) and integrate them into application in one special branch (development branch). Each release of the application was done by separate commit and tagged by git tags in order to identify it quickly. Furthermore, I used interactive rebasing to keep git repository clean and demonstrative.

3.2 Libraries

This chapter introduces the most important libraries, which were used during development of this project.

3.2.1 Stetho

Stetho is a debug bridge for Android applications developed by Facebook. It allows using the Chrome Developer Tools for debugging Android applications. I mostly used it for network inspection because Android Studio does not provide any convenient tool for it. Stetho would also be used for inspection if the application had had a local database.

3.2.2 RxJava 2

RxJava 2 is an implementation of Reactive Extensions for Java language. It is "*an API for asynchronous programming with observable streams*", "*combination of the best ideas from the Observer pattern, the Iterator pattern, and functional programming.*" [12] Simply said, this library can help you to convert almost anything into a stream of data and allows you to manipulate with such streams differently.

I used this library mostly for obtaining data from the ElateMe backend server and manipulating them before showing to the user. For example, in Figure 3.1 is shown how I get a stream of data, which receives the authentication token at first and then downloads wish by id using this token.

```
@Override
public Single<Wish> getWish(Long id) {
    //obtaining the token from loginService
    return loginService.getToken()
            .flatMap(token ->
            //using the token to make a request
                wishesApi.getWish(token.getTokenAsHeader(), id)
                .toSingle());
}
```

Figure 3.1: Example of using RxJava for obtaining token before actual request

3.2.3 RxTuples 2

"*RxTuples is a library to smooth RxJava usage by adding simple Tuple creation functions.*" [20] This library comes into action, when we need to pass more than one value from one RxJava operation into another. There is a special tuple data type for such cases in other languages (for example TypeScript), but not in Java. "*A tuple is a collection of several elements that may or may not be related to each other.*" [21] So, I used this library as a replacement for tuples in Java. In Figure 3.2 is shown how I used triplet (tuple of three values) to pass three values from the switchMap operator to the map operator.

3.2.4 EventBus

This library developed by Greenrobot is "*event bus for Android and Java that simplifies communication between Activities, Fragments, Threads, Services, etc.*" [22] We can post some messages into event bus and deliver them in our subscribers. It is important to mention, that event bus messages are simple POJOs. So, it is a perfect solution for communicating between Android Activities and Services. I used EventBus for one way communication between service for popups and notifications updating and activities (screens) of our application. All popups and notification updates are delivered to the activities via EventBus.

```
    //the first RxJava operator
    .switchMap(wishResponse ->
        Observable.zip(
            //zipping wish
            Observable.just(wishResponse),
            //zipping author of the wish
            this.userService
                .getUserAccountInfo(wishResponse.getAuthor().getId())
                .toObservable(),
            //zipping comment of the wish
            this.commentsService
                .getComments(wishResponse.getId())
                .toObservable(),
            //using Triple to "pack" all three values into one tuple
            Triplet::with))
    //the second RxJava operator
    .map(triplet ->
        //accessing tuple values
        mapToMyWishViewModel(triplet.getValue0(), triplet.getValue1(),
            triplet.getValue2())
    );
```

Figure 3.2: Example of using RxTuples for passing several values between RxJava operators

3.2.5 Lombok

“*Lombok is a java library that automatically plugs into your editor and build tools, spicing up your java. Never write another getter or equals method again. Early access to future java features such as val, and much more.*” [23]

I used Lombok’s @Data annotation to create data classes (such as classes for responses and manipulating data) in a clean and demonstrative way. Lombok generates getters and setters for non-final fields, equals, hashCode and toString methods for classes, which are marked with this annotation. Example of such data class is shown in Figure 3.3.

The other useful feature of Lombok is a @Builder annotation. It “*produces complex builder APIs for classes.*” [24] The other words, Lombok implements builder pattern for a class, which is annotated with this annotation. Builder pattern is very handy in case of instantiating an object with a lot of parameters. Without builder, you would need to instantiate an object and provide all parameters via setters. The problem with this approach is that your object may be in an inconsistent state after instantiation but before providing required parameters. The other way to instantiate an object with a lot of parameters is to use a constructor. But there will be a lot of different constructors to cover all needed combinations of parameters and edge cases.

```
@Data
@Builder
public class Donation {
    @SerializedName("id")
    private Long id;

    @SerializedName("donator")
    private User donator;

    @SerializedName("wish")
    private Long wish;

    @SerializedName("amount")
    private Double amount;

    @SerializedName("date")
    private DateTime date;
}
```

Figure 3.3: Example of Donation data class with Lombok's annotations

When the builder pattern is realized, a class will have inner static helper class, which can be used to instantiate an object. You can provide parameters in any order and finally call a method for building an object. Example of usage of the builder pattern is shown in Figure 3.4.

```
Donation newDonation = Donation.builder()
                .donator(donator)
                .wish(wish)
                .amount(amount)
                .date(todayDate)
                .build();
```

Figure 3.4: Example of usage Donation builder generated by Lombok library

3.2.6 Constraint layout

Android SDK has a lot of different layouts, which help to build a user interface. Actually a layout is a group of views, and all different kinds of layouts place its own child views in a different way. There is, for example, a linear layout, which "*aligns all children in a single direction, vertically or horizontally.*" [25], or relative layout, which displays child views in relative positions. "*The position of each view can be specified as relative to sibling elements or in positions relative to the parent area.*" [26] The main problem with those layouts is that you often need to insert one layout into another, making the tree of views very deep. And this can be a very serious performance issue. Fortunately, Google introduced a more productive and convenient layout – Constraint Layout.

"*Constraint layout allows you to create large and complex layouts with a flat view hierarchy (no nested view groups).*" [27] All the power of this layout is that "*ConstraintLayout is available directly from the Layout Editor's visual tools because the layout API and the Layout Editor were specially built for each other. So you can build your layout with ConstraintLayout entirely by drag-and-dropping instead of editing the XML.*" [27]

While creating new layouts for our application, I used Constraint layout because it is convenient, intuitive and provides better performance.

3.2.7 ViewModel

It is important to mention that in Android SDK you can not pass an object between activities (screens) and fragments (piece of an application's user interface). You can only pass primitive types of data (such as string, integers and so on) or serializable objects (objects, which can be decomposed to primitive data types). The problem is you can not share the same object between activities and fragments. ViewModel is a library developed by Google especially for such problem in Android development. If your activity consists of fragments, you can share one object called ViewModel between those fragments and parent activity itself. Ability to share the same object solves the problem with serialization/deserialization objects and passing them from one fragment to the other. Your fragments can interact with the same object without additional code and overhead.

For example, I used this library to share a list of visible (for a wish) friends between two fragments (list of recipient's friends and list of my friends) in the screen of wish creation.

3.2.8 Android View Badger

Android View Bagder is a library, which helps you to put a badge on your view. A badge is simply a counter (for example, a counter of unread messages). It is a very popular way of showing something countable to the user, and it is widely used among different platforms (such as iOS, Android, macOS). I used this library to put a badge on the button of notifications tab on the main screen of our application.

3.3 Custom binding adapters

Our application uses data binding library, which helps you bind some data to the appropriate views. You can set, for example, a boolean variable to view's visibility and this view will be visible or gone depending on a value of this bound variable. Furthermore, this can be done without writing Java code. All bindings happen in the XML file of the view, and it is a great advantage because your Java code won't be flooded by primitive boilerplate code (for example, controlling the visibility of a view, as was mentioned above).

Custom adapters come into play when you need to write your binding logic. Simply put, you can write Java code, describing what should happen when data binding library binds your data to a view. Also, you can define your own XML attributes (which your view will use) by writing custom binding adapters.

I wrote a custom binding adapter for loading images into views. I defined two new XML attributes. The first one is for providing URL of the image, and the other one is for providing the type of the image. The main purpose of the type is that image loader (Picasso library) will use default image (placeholder) in case of absence or unavailability of the image. There are three types of default images defined in our application, such as male image, female image, and gift image. The implemented binding adapter is shown in Figure 3.5 and usage of this adapter is shown in Figure 3.6.

```java
@BindingAdapter({"android:imageUrl", "android:imageType"})
public static void setImage(View view, URL imageUrl, ImageType type) {
    ImageLoader
        .loadImage(view.getContext(), imageUrl, (ImageView) view,
            type);
}
```

Figure 3.5: Implemented binding adapter for loading images into a view

```xml
<de.hdodenhof.circleimageview.CircleImageView
           android:id="@+id/comment_avatar"
           android:layout_width="46dp"
           android:layout_height="46dp"
           android:layout_alignParentLeft="true"
           android:layout_alignParentTop="true"
           android:layout_marginRight="14dp"
           android:imageUrl="@{CommentVM.imageUrl}"
           android:imageType="@{CommentVM.imageType}"
           app:civ_border_width="0dp"
           tools:src="@drawable/avatar" />
```

Figure 3.6: Usage of the binding adapter for loading images into a view

Chapter 4

Testing

This chapter introduces approaches, which were used for writing tests of our application. Nonetheless, it is necessary to describe basics of the application architecture before the introduction.

An architecture of our application is based on MVP architecture pattern. Communication between layers is shown in application component diagram in Figure 2.20.

Since our application has three global layers, we need to test each of them. There are three types of tests:

- UI layer testing (View in MVP).
- Presentation layer testing (Presenter in MVP).
- Service layer testing (Model in MVP).

Testing of each layer (including used libraries and solutions of essential problems) is described further.

4.1 Common libraries for testing

JUnit

"*JUnit is a simple framework to write repeatable tests.*" [28] This framework helps to write tests, group tests by test cases and write code, which should be run before/after each test or test case. It also contains a lot of functions, which allow us to compare primitive values, arrays, and even objects quickly.

Mockito

Mockito is "*the most popular mocking framework for Java.*" [29] Mocking is replacing an object with mock, that simulate the behavior of the real object. It helps in testing when we need to isolate object we want to test and surround it with mocked dependencies. Example of mocking an object is shown in Figure 4.1.

```
@Mock
CommentsDataProvider commentsDataProvider;

@Before
public void setUp() {
    MockitoAnnotations.initMocks(this);

    gson = new GsonBuilder()
            .registerTypeAdapter(DateTime.class, new
                DateTimeTypeConverter())
            .create();

    CommentList commentList = gson.fromJson(COMMENTS,
        CommentList.class);

    //making mocked CommentsDataProvider return Single<CommentList>
    //when getWishComments(Long wishId, Integer page, Integer
        pageSize) is called
    when(commentsDataProvider.getWishComments(anyLong(), anyInt(),
        anyInt()))
            .thenReturn(Single.just(commentList));
}
```

Figure 4.1: Example of mocking the CommentsDataProvider object

4.2 Testing of the UI layer

A UI layer is the only one layer of our application, which interacts with Android SDK. It means that code of the UI layer has Android SDK dependencies. Since Android OS runs on Dalvik/ART virtual machine (not on the JVM), the only way to run such code is to use real Android device or emulator. That is why tests of a UI layer require to run them in Dalvik/ART virtual machine as well.

Testing scheme

- UI layer is emulated by Robolectric.
- Presenter layer is mocked by Mokito.

Robolectric

But there is one way how to run code with Android SDK dependencies on the JVM (which can be installed fast and easy on any PC or server). "*Running tests on an Android emulator or device is slow! Building, deploying, and launching the app often takes a minute or more.*" [30] "*Robolectric rewrites Android SDK classes as they are being loaded and making it possible for them to run on a regular JVM.*" [30] Thanks to this testing framework we can run the UI layer tests locally on development machine or server without having real Android device or emulator.

I used Robolectric to emulate the behavior of Android SDK. But there is another problem which should be solved in order to get working tests. To test how our UI components communicate (usually delegate user interactions) with presenters, the presenters should be mocked. The mocking is not a big problem and can be done with the help of Mockito library. The actual problem is to inject our mocked presenter into our UI component (usually Android activity or fragment).

Moxy

"*Moxy is a library that helps to use MVP pattern when you do the Android Application. Without problems of lifecycle and boilerplate code!*" [31] One of the main advantages of the library is that it handles creating and injecting a presenter into UI component by itself. This library is very handy, although a problem takes place when you want to write tests. Moxy does not provide any convenient way to mock a presenter, so we need to do it somehow.

Research showed that we can override Moxy's store (which contains and provides presenters) and set our mocked presenter to it. The last thing we need to do is use Moxy's static MvpFacade and set the custom store to it. As a result, if UI component is created, our mocked presenter will be provided to it.

4.3 Testing of the presentation layer

Since the main purpose of a presenter is to be an abstract layer between data and UI, presenters in our application do not contain any Android SDK dependencies. Thus, we can easily test them in JVM.

Testing scheme

- UI layer (Android activities and fragments) is mocked by Mockito.
- Presentation layer is normally instantiated (as it would do during the application running).

- Service layer is mocked by Mockito.

Android schedulers problem

Android schedulers are threads in which code of our application runs. There are two important threads – UI thread and computation thread. UI thread is a thread where code of UI components runs. Computation thread is a thread where the other code runs (such as network calls or heavy computations). The reason for separation UI code running is pretty simple – we do not want to block user's interactions during computations.

But this separation can be a problem during a testing of the presentation layer. Reacting to some user interactions (which are simulated by mocked UI component) a presenter usually performs a long-running operation. The problem is that a test does not know when it should check if the presenter did something at the end of the operation. Thus, we need to make our code run synchronously to know when we should check presenter's behavior.

Our application uses Android schedulers static class from RxJava library to obtain appropriate threads. Fortunately, RxJava allows us to override provided threads. The solution is to provide so-called trampoline thread for all operations. It helps us to run all, and as a result, we can test our asynchronous code easily.

4.4 Testing of the service layer

A code of a service layer does not contain Android SDK dependencies as well. But it contains a lot of reactive code (observable data streams), and we need to test it somehow.

Testing scheme

- Service layer is normally instantiated.
- Data provider layer is mocked by Mockito.

Dependency injection and mocking

The main problem with testing of the service layer is mocking our dependencies. Each service contains several dependencies on our data providers. Moreover, those dependencies are provided by Dagger 2 framework. I used DaggerMockRule library to mock all required needed dependencies easy without boilerplate code. The first thing to do is define a component (component inject dependencies in Dagger 2). And the second one is to mark all dependencies, which should be mocked. As a result, a testing service gets mocked dependencies instead of real ones. Usage of DaggerMockRule library is shown in Figure 4.2.

```
//setting up DaggerMockRule
@Rule
public DaggerMockRule<AppComponent> daggerMockRule =
       //specifying component
       new DaggerMockRule<>(AppComponent.class,
              //providing domain module (test module can be provided)
              new DomainModule(RuntimeEnvironment.systemContext),
              //providing data module (test module can be provided)
              new DataModule(),
              //providing service module (test module can be
                  provided)
              new ServiceModule())
              //setting mocked component to the application
              .set(component ->
                  ((App)RuntimeEnvironment.application)
                  .setAppComponent(component));

//marking dependency for mocking
@Mock
UserDataProvider userDataProvider;

//marking dependency for mocking
@Mock
CommentsDataProvider commentsDataProvider;

//obtaining object for testing
//with injected mocked dependencies
@InjectFromComponent
CommentsService commentsService;
```

Figure 4.2: Usage of DaggerMockRule library

Testing data streams (observables)

Testing of a service layer, unlike testing of a presentation layer, does not have the problem with threads because we are able to get needed data stream. Fortunately, RxJava library provides some test operators and classes. The main feature is an ability to make test observer from any observable. The test observers support a lot of operators for testing, such as different assertions and checks. Usage of test observer is shown in Figure 4.3.

```java
@Test
public void commentsService_shouldDeserializeCommentsList() {
    //obtaining test observable from normal observable
    TestObserver<List<CommentViewModel>> testObserver =
        commentsService.getComments(0L).test();

    //synchronously awaiting of terminating of the observable
    testObserver.awaitTerminalEvent();

    testObserver
        //checking observable was completed without errors
        .assertComplete();
}
```

Figure 4.3: Usage of a test observer

Chapter 5

Continuous integration

"*Continuous Integration (CI) is the process of automating the build and testing of code every time a team member commits changes to version control.*" [32] There are a variety of tools for continuous integration (for example Jenkins, Circle CI and others). Our team decided to use GitLab CI. GitLab CI is CI for building and testing a code which is integrated into GitLab. "GitLab is one of the more popular Git servers." [33] Our team uses it for development, so integrated CI is a good solution.

To have our builds and tests run automatically, we needed to do two steps:

1. Create/find appropriate Docker image.

2. Set up GitLab CI.

Docker image

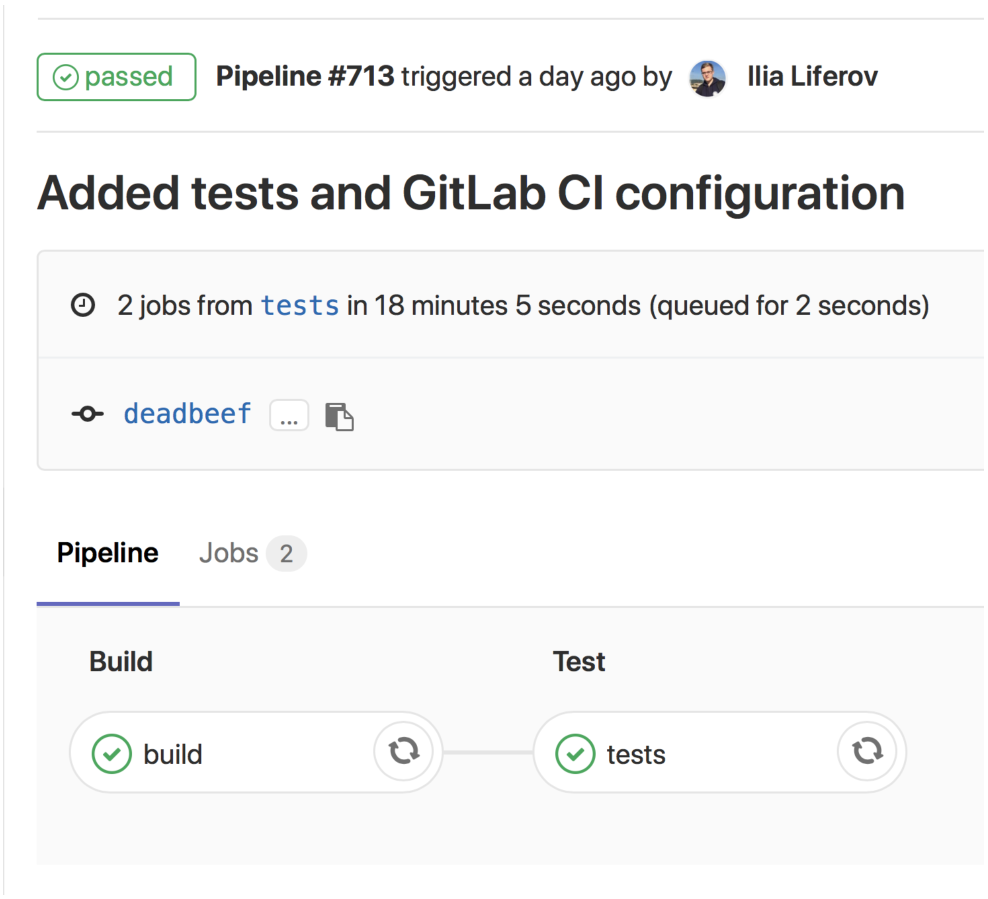
"*A Docker container image is a lightweight, stand-alone, executable package of a piece of software that includes everything needed to run it: code, runtime, system tools, system libraries, settings.*" [34] GitLab CI requires docker image to run the build and tests. I used already prepared open-source docker image, which was found by Michal Maněna. "*This Docker image contains the Android SDK and most common packages necessary for building Android apps in a CI tool like GitLab CI.*" [35]

Setting up GitLab CI

According to GitLab CI's Getting Started guide, all we need is to add a .gitlab-ci.yml file to our repository. This file specifies Docker image, which should be used and defines stages. There are two stages (build and test) in our case. .gitlab-ci.yml file for our application's CI pipeline is shown in Figure 5.1 and the result of running this pipeline is shown in Figure 5.2.

```
#docker image for android build
image: jangrewe/ gitlab −ci−android

 before_script :
     - export GRADLE_USER_HOME=‘pwd‘/.gradle
     - chmod +x ./gradlew

cache:
  key: "$CI_COMMIT_REF_NAME"
  paths:
      - .gradle/

#stages of pipeline
stages:
  #stage for building an android application artifacts
  - build
  #stage for testing an application
  - test

#description of build stage
build:
  stage:  build
  #gradle command for running the build
   script :
      - ./gradlew assembleDebug
   artifacts :
    #folder for storing application artifacts after the
        successful build
    paths:
      - app/build/outputs/apk/

#description of test stage
tests :
  stage:  test
  #gradle command for running the tests
   script :
     - ./gradlew test
```

Figure 5.1: The application’s CI pipeline file


passed Pipeline #713 triggered a day ago by Ilia Liferov

Added tests and GitLab CI configuration

2 jobs from tests in 18 minutes 5 seconds (queued for 2 seconds)

deadbeef ...

Figure 5.2: Result of running the application's build/test pipeline in GitLab CI

Conclusion

The goal of this thesis was to continue in the development of the Android application for ElateMe. The task was to analyze, design and implement new requirements such as handling of collection/wish, sharing a product from a web browser into the application and using different kinds of payments (bank card, Bitcoin, PayPal, Android Pay). Moreover, the application should be tested with appropriate automated tests.

Fulfillment of the thesis's task:

1. **Analyze and implement new requirements using FURPS+.**

   All important requirements were analyzed, designed and implemented. For example, the new screen for wish creation, wish comments in several screens, popups, and notifications.

2. **Analyze and implement new kinds of payments (bank payments, Bitcoin payments, PayPal payments and Android Pay payments).**

   All those kinds of payments were analyzed, designed and implemented. Bank payments were made with the help of Fio bank web gateway. Bitcoin payments were made by integrating with BitPay service (using the BitPay Android SDK). PayPal and Android Pay payments were made by integrating with Braintree service (using Braintree Android SDK).

3. **Analyzed and implement sharing a product from a web browser.**

   The requirement was successfully analyzed, designed and implemented. The new feature was achieved by interacting with product recognition algorithm, which is placed on the ElateMe backend server. Communication between web browsers and our application was done with the help of standard Android OS approach - usage of intents and intent filters.

4. **Update platform specific model/use appropriate UML diagrams.**

   Sequence diagrams were provided for every important and sophisticated process described in this thesis. Appropriate class diagrams and component diagrams were provided (or updated) as well.

5. **Unit tests were written for the application.** Tests of the UI was done by emulating Android SDK with the Robolectric framework. Although code coverage is pretty short, testing environment and approaches were analyzed and worked out. Thus, next developers can use written tests as examples. Tests were successfully automated with GitLab CI.

Information for next developers:

- Please notice that two dependencies (Android view badger and BitPay SDK) are local in the gradle build system of the application, so they will not be downloaded by gradle. Nevertheless, it is just for your information. You should not face any problems during a building because these two dependencies are stored in version control system.

- If you want to release a new version of the application, you need to sign apk file before uploading to Google Play Console. (see section 1.4.2) Signing script is configured at the beginning of a build gradle file, but you will need a file with keystore from Michal Maněna in order to successfully sign the apk.

- Data provider layer is almost useless for now, but you will need it if you decide to cache application data locally.

Bibliography

[1] Solovev, G. *ElateMe - Android client*. Bachelor's thesis, Czech Technical University in Prague, Faculty of Information Technology, 2017.

[2] Sign Your App. [online], [viewed 15 April 2018]. Available from: https://developer.android.com/studio/publish/app-signing.html

[3] App manifest file. [online], [viewed 15 April 2018]. Available from: https://developer.android.com/guide/topics/manifest/uses-sdk-element.html

[4] Distribution dashboard. [online], [viewed 5 Feb 2018]. Available from: https://developer.android.com/about/dashboards/

[5] Bitcoin. [online], [viewed 15 April 2018]. Available from: https://bitcoin.org/en/

[6] Bitcoint address. [online], [viewed 18 April 2018]. Available from: https://en.bitcoin.it/wiki/Address

[7] Lukaševič, L. *Analýza a implementace internetových plateb v rámci aplikace Elateme*. Bachelor's thesis, Czech Technical University in Prague, Faculty of Information Technology, 2018, in preparation.

[8] Domino's Android app. [online], [viewed 5 May 2018]. Available from: https://www.dominos.com.au/inside-dominos/technology/android-pay

[9] Material design. [online], [viewed 20 April 2018]. Available from: https://material.io/design/

[10] Jan Hoffman's design. [online], [viewed 19 April 2018]. Available from: http://elateme-web.janhoffman.cz

[11] Miličic, D. A detailed guide on developing Android apps using the Clean Architecture pattern. [online], Feb 2016, [viewed 19 April 2018]. Available from: https://medium.com/@dmilicic/a-detailed-guide-on-developing-android-apps-using-the-clean-architecture-pattern-d38d71e94029

[12] ReactiveX. [online], [viewed 23 April 2018]. Available from: http://reactivex.io

[13] Services overview. [online], [viewed 23 April 2018]. Available from: https://developer.android.com/guide/components/services

[14] WebView. [online], [viewed 23 April 2018]. Available from: https://developer.android.com/reference/android/webkit/WebView

[15] BitPay Android SDK. [online], [viewed 23 April 2018]. Available from: https://github.com/bitpay/android-sdk

[16] Intent and Intent Filters. [online], [viewed 23 April 2018]. Available from: https://developer.android.com/guide/components/intents-filters

[17] Andmade Share: For a more robust sharing menu. [online], [viewed 3 May 2018]. Available from: https://www.androidcentral.com/andmade-share-more-robust-sharing-menu

[18] Ferreira, C. Package by features, not layers. [online], Nov 2015, [viewed 25 April 2018]. Available from: https://hackernoon.com/package-by-features-not-layers-2d076df1964d

[19] Git. [online], [viewed 27 April 2018]. Available from: https://git-scm.com

[20] RxTuples. [online], [viewed 27 April 2018]. Available from: https://github.com/pakoito/RxTuples

[21] Introduction to Javatuples. [online], [viewed 27 April 2018]. Available from: http://www.baeldung.com/java-tuples

[22] EventBus. [online], [viewed 27 April 2018]. Available from: https://github.com/greenrobot/EventBus

[23] Project Lombok. [online], [viewed 27 April 2018]. Available from: https://projectlombok.org

[24] Lombok Builder. [online], [viewed 27 April 2018]. Available from: https://projectlombok.org/features/Builder

[25] Linear layout. [online], [viewed 27 April 2018]. Available from: https://developer.android.com/guide/topics/ui/layout/linear

[26] Relative layout. [online], [viewed 27 April 2018]. Available from: https://developer.android.com/guide/topics/ui/layout/relative

[27] Build a Responsive UI with ConstraintLayout. [online], [viewed 27 April 2018]. Available from: https://developer.android.com/training/constraint-layout

[28] JUnit. [online], [viewed 27 April 2018]. Available from: https://junit.org/junit4

[29] Mockito. [online], [viewed 27 April 2018]. Available from: https://github.com/mockito/mockito

[30] Robolectric. [online], [viewed 27 April 2018]. Available from: http://robolectric.org

[31] Moxy. [online], [viewed 27 April 2018]. Available from: https://github.com/Arello-Mobile/Moxy

[32] What is Continuous Integration? [online], [viewed 29 April 2018]. Available from: https://docs.microsoft.com/en-us/azure/devops/what-is-continuous-integration

[33] Git on the Server - GitLab. [online], [viewed 29 April 2018]. Available from: https://git-scm.com/book/en/v2/Git-on-the-Server-GitLab

[34] What is a container. [online], [viewed 29 April 2018]. Available from: https://www.docker.com/what-container

[35] GitLab CI image for building Android apps. [online], [viewed 29 April 2018]. Available from: https://github.com/jangrewe/gitlab-ci-android

Appendix **A**

Acronyms

UML Unified Modeling Language

API Application Programming Interface

UI User Interface

CI Continuous Integration

SDK Software Development Kit

FURPS+ Functionality, Usability, Reliability, Performance, Supportability, and Constraints

MVP Model View Presenter

APK Android Application Package

URL Uniform Resource Locator

HTTP Hypertext Transfer Protocol

XML Extensible Markup Language

ART Android Runtime

JVM Java Virtual Machine

Appendix **B**

Contents of enclosed flash drive

```
root
├── readme.txt ........................ important information
├── ElateMe_Android_client_Thesis_text.pdf ........ thesis's text
├── ElateMe_Android_client_VPP_project.vpp . Visual Paradigm project
│   with UML diagrams
├── thesis_sources ............... source code of the thesis's text
├── src .............. directory with source code of the application
└── ElateMe_2018.apk ........... APK installation file for Android OS
```