CHALMERS
UNIVERSITY OF TECHNOLOGY

A Chalmers University of Technology Bachelor's Thesis

Effective Debt Distribution
DATX02-15-05

Martin Calleberg
Julia Friberg
Linus Hagvall
Elin Ljunggren
Josefin Ondrus
Per Thoresson

Supervisor: Alex Gerdes
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2015

Effective Debt Distribution

Martin Calleberg
Julia Friberg
Linus Hagvall
Elin Ljunggren
Josefin Ondrus
Per Thoresson

Examiner: Arne Linde

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000
Göteborg, Sweden June 2015

Abstract

In this report the development of an Android application that structures debts between users is studied. These debts are optimised to find the easiest way to make all users free from debts. A number of relevant optimisation algorithms have been thoroughly investigated. We have searched for algorithms which optimise with the objective to find as few and small transactions as possible. The chosen algorithm searches for cycles of users where debts can be simplified.

The application is designed with usability in focus. To increase the application's usability the design of the application follows Material Design, which is a set of design standards for Android applications established by Google. All data is saved on a server which the client can connect to through a REST API.

Sammanfattning

I denna rapport studeras utvecklingen av en Androidapplikation som strukturerar skulder mellan användare. Dessa skulder optimeras för att hitta det enklaste sättet för alla användare att bli skuldfria. Ett antal relevanta algoritmer har blivit noggrant undersökta. Vi har letat efter algoritmer som optimerar med målet att hitta så få och små transaktioner som möjligt. Den valda algoritmen letar efter cykler av användare där skulderna kan förenklas.

Applikationen är designad med användarvänlighet i fokus. En följd av detta är att utseendet följer Material Design, som är ett set designstandarder för Andoridapplikationer som upprättats av Google. All data sparas på en server som klienten kan ansluta sig till via ett REST API.

Acknowledgements

We would like to thank our supervisor Alex Gerdes for all his help during this project. We would also like to express our gratitude to our mentors Ger Garrigan and Björn Thalén at Jeppesen.

This Bachelor's Thesis was written during the spring semester of 2015 at Chalmers University of Technology.

Vocabulary

Android - Mobile operating system, developed by Google.

API (Application Programming Interface) - Interface representing the possible operations.

Domain model - Conceptual model of the system.

Eclipse Java EE IDE for Web Developers - Tools for creating Java EE and web applications

Git - Version control system.

I/O (Input/Output) - Communication between two systems or system and human.

IDE (Integrated Development Environment) - Used to edit, compile and run code.

iOS - Mobile operating system, developed by Apple.

Java EE (Java Platform Enterprise Edition) - Oracle's enterprise Java computing platform

JUnit - JUnit is used to write and run repeatable tests where output can be asserted to make sure it equals the expected output.

Mock-up - Sketch representing a prototype of a UI.

PayPal - Online payment system.

Resource - Term used when having a resource based API, used when referring to pieces of data.

RESTful web service - A web service designed according to RESTful design principles.

SQLite - Database engine embedded in the program using it.

Swish - Mobile app to send money to other people using the app.

Tomcat - Open source web server. Provides an environment for running Java code.

UI (User Interface) - What the user sees and interacts with.

Use cases - A list of steps that are executed by a user of a system to accomplish a goal, also shows steps done by the system.

XML (Extensive Markup Language) - Markup language used to encode data in a way that is readable by both humans and computers.

Table of contents

1 Introduction **1**
- 1.1 Purpose . . . 1
- 1.2 Scope . . . 2
- 1.3 Challenges . . . 3
- 1.4 Overview of result . . . 3

2 Technical background **5**
- 2.1 Developing for Android . . . 5
- 2.2 Material Design . . . 7
- 2.3 Java . . . 7
- 2.4 Apache Tomcat server . . . 7
- 2.5 Representational State Transfer - REST . . . 7
- 2.6 HyperSQL database . . . 8
- 2.7 Optimisation models . . . 8

3 Method **10**
- 3.1 Programing languages and formats . . . 10
- 3.2 Development tools . . . 10
- 3.3 External libraries . . . 11
- 3.4 Work process . . . 11

4 Requirements and modelling **14**
- 4.1 Description of the domain problem . . . 14
- 4.2 Components and structure . . . 15
- 4.3 Documentation process . . . 16
- 4.4 Optimisation algorithms . . . 16

5 Implementation **22**
- 5.1 Front end . . . 22
- 5.2 Back end . . . 25
- 5.3 Optimisation using cycle transformation . . . 27

6 Result and discussion **30**
- 6.1 Front end . . . 30
- 6.2 Back end . . . 40
- 6.3 Optimisation . . . 41
- 6.4 Removed features . . . 44

6.5 Possible improvements 45
6.6 External guidance 46
6.7 Impact on society 47

7 Conclusion **48**

Bibliography **48**

Appendices **I**

Appendix A Gantt chart **II**

Appendix B Requirements **III**

Appendix C Speed test of algorithm **VI**

1
Introduction

A reoccurring situation is when one or more persons temporarily pay for others. It may be a matter of a bus ticket, the bill at the restaurant or an after-work beer. The problem is that these small debts are often forgotten or mixed up. This can lead to misunderstandings and conflicts as well as contribute to a reduced cost overview. Furthermore, there may be unnecessary costs in terms of time when trying to calculate the correct amount as well as expenses for each transfer made. For students this is probably a larger problem than for people in the working life, since student life often involves more cash management and a weaker economy overall. The idea for this project was mainly based on experiences gained from life as a student.

To make an easily accessible system for helping people in everyday situations like the ones previously mentioned, creating an application for mobile devices could be a good idea. Statistics shows that in the beginning of 2013 about 60% of Sweden's population claim to have used a smartphone to connect to the internet. Among a younger demographic, 16-44 years old, the number was 83-86% [1]. Based on these statistics, it seems that the smartphone is now a part of people's life.

Storing debt information and comparing data within a network of friends could make it possible to minimise the number of transactions resulting in more effective repayments which in turn could reduce unnecessary costs. There exist several applications, for example Splitwise [2] and Pay Back [3], managing some of this functionality but these applications only handle debts between two users or within a specified group of users. In this project we will create an application that minimises transactions between all users and not just in predefined groups.

1.1 Purpose

The purpose of this project is to make it easier for people to manage their debts by developing a mobile application which keeps track of the users' debts and optimises the debt repayment. The application should minimise the number of transactions and at the same time make sure that the amount of each transaction is kept to a minimum. We will investigate which requirements are reasonable to impose on the application and whether such an application can be implemented.

In order to attract users, the application should be easier and faster to use than to calculate the debts with pen and paper. The user interface (UI) should be intuitive yet efficient, with the intention to appeal to a target group as wide as possible.

1.2 Scope

The application keeps track of and optimises different users' debts to each other as described in the previous section. However, the application is not able to handle any transactions or verify that the transactions are made. The application relies on the fact that users enter correct data and treats all entered data as valid.

For the application to be as useful as possible it is required that a user's friends also use the application. This means that the application is of limited help until it is used by a lot of people. This is a problem which needs to be addressed before a possible release of the application. However, the marketing problem is not dealt with and is outside the scope.

The application is created for Android, following Android's guidelines for the UI [4]. Other platforms were not considered during this project. However, most of the logic runs on the back end, which means that adding clients for other platforms is relatively straightforward.

For any application that contains personal accounts, especially one that involves financial statements, proper security is required. The application needs personal accounts to create debts between different users, and will therefore need some sort of login. Although a secure login and encrypted messages are necessary for a release of the application, this is outside the scope of this project.

To make it easier for the user to pay back debts, integration with third party applications could be a potential feature. These could be Swish, PayPal or a similar service. Even though this might be a desirable feature, it is not in the scope.

1.3 Challenges

During the development of this application there were a few problems which needed to be addressed. The most important problems are discussed in this section.

Since the application minimises the number of payments an optimisation problem needs to be solved. The first step will be to define a mathematical model. Next, a suitable algorithm needs to be found.

When deciding which algorithm to use the time complexity has to be taken into consideration since a long computation time has a negative impact on the user experience. There could exist algorithms that give an optimal answer but take too long time to execute. An algorithm that executes fast but does not give an optimal solution could therefore be more suitable for our application. It is necessary to investigate whether an exact solution is needed and how important the speed is for the user experience.

Another problem that needs to be addressed is the need for a scalable back end in order to handle a potential increase of users over time. It should be easy to implement clients for different platforms, such as iOS and Android, without making any significant changes to the back end.

When developing the user interface for the front end client there are a number of factors to take into consideration. The UI has to be as intuitive as possible to not intimidate new users. It also needs to reflect the application's goals, to be fast and easy to use. It is necessary to investigate how the UI should look and function to reach these goals.

To make a user trust the application, the UI must provide a feeling of security. Less information makes the application easier to use, but also makes it feel less secure [5]. A decision about how much information a user needs to see about the debts has to be made.

1.4 Overview of result

We have developed a mobile application for the Android platform in which users can enter debts. These debts are automatically optimised such that the number of transactions and the amount to be paid are minimal. Once logged in, the user gets an overview of their current optimised repayments which tells the user how to pay back the debts as efficient as possible. This screen also displays recent events such as debts added, removed or paid. From this screen the user can easily add a new debt trough a button which takes the user to the add debt screen. Here the user specifies information about the debt and the friends involved.

When a debt is added the optimisation process is started. The optimisation algorithm uses cycle transformation to calculate a more efficient way to pay back the debts currently in the network of users. This algorithm was found to be the most suitable for our project after investigating three different algorithms. The solution functions well even when a large group of people uses the application, which can be seen by the different tests carried out on the back end.

Several other features were added apart from just adding debts between users. These are further discussed in *6 Result and discussion*.

2
Technical background

Developing a system following a client-server architecture requires the use of different software components. This chapter will provide an introduction to the different techniques used as well as some information about the mathematical models that were investigated during the project.

2.1 Developing for Android

Android is the most used mobile operating system in the world. It is developed by Google and is used on hundreds of millions of smartphones in more than 190 countries [6]. Android is not just used by most phones, it also has access to the most apps. In July 2013 Google Play Store, where apps for Android can be purchased and downloaded, exceeded all competitors in terms of the number of apps available [7]. In February 2015 Google Play Store contained 1,4 billion apps and the number keeps growing [8].

When developing for Android there are a number of components which are used. Below the main components are introduced.

2.1.1 Activities

An activity is a main component when implementing an Android application. An activity creates a window which can be filled with content to display to the user. The window often, but not necessarily, fills the whole screen. An application consists of several activities that are linked to each other, typically one activity is used for one action.

An application usually has a main activity which is started when the application is started. The main activity starts other activities according to what actions should be performed. When a new activity is started, the previous activity is stopped. To avoid activities from being created several times, activities are saved to a stack. This enables a previous activity to be restarted after a new activity is finished [9].

2.1.2 Fragments

Fragments handle specific parts of the user interface and are part of activities. One activity can contain several fragments. Fragments are often used to display the same thing in several activities. They can handle their own input and can be removed and added to the current activity as necessary.
Fragments cannot exist without an activity and are directly impacted by what happens to the activity. If the activity is stopped, so are the fragments embedded in that activity. Fragments are added to a stack similarly to activities [10].

2.1.3 Layouts

What is shown to the user can be handled both through layout files in Extensive Markup Language (XML) and programmatically. It is recommended to declare all UI elements in an XML file to easier separate code from views and to easily change the user interface without changing the code and recompiling. Since an application is used on multiple devices with different sizes, different XML files can be implemented and used according to the device the app is currently running on. This is also an argument for using XML files. It can be beneficial to combine the two techniques and programmatically change what is shown when necessary [11].

2.1.4 Services

A service is a component that does not have a user interface. It is used for long-running operations in the background and keeps running even when the application is not used. This is useful for operations like file I/O, network transactions and playing music. Services are also used for notifications [12].

2.1.5 Broadcast receiver

Broadcasts are messages that can be sent to any application. To receive such a message the application needs a broadcast receiver which can handle broadcast messages and decide what actions should be taken [13].

2.2 Material Design

Android 5.0 was released in the summer of 2014. With it came Material Design, Google's new visual language. The purpose of this visual language is to help developers create consistent apps which have the same basic look but that can be extended to do whatever the app is supposed to do [4]. This language is best described with Google's own words that forms their two goals with Material Design:

- "Create a visual language that synthesises classic principles of good design with the innovation and possibility of technology and science." [4]
- "Develop a single underlying system that allows for a unified experience across platforms and device sizes. Mobile precepts are fundamental, but touch, voice, mouse, and keyboard are all first-class input methods." [4]

2.3 Java

Java is a programing language first released in 1995 by Sun Microsystems [14]. It is a concurrent, class-based and object-oriented language and closely resembles the programing language C [15].

One of the biggest advantages with Java is that it is platform independent and lets you write one solution that, regardless of the computer architecture, can run on any Java Virtual Machine [16]. This is possible since the Java code is compiled into bytecode, which is a representation of instructions that can be efficiently interpreted and thereby minimise the operating system dependency for the code. Another advantage of Java is that it features a high performance garbage collector which releases memory when objects are no longer used, thus taking care of some of the responsibility otherwise put on the developer [15].

2.4 Apache Tomcat server

Tomcat can be used as a server for various applications and is an open source web container released in 1999 by Sun Microsystems. It is currently managed by the Apache Software Foundation. [17] Tomcat implements several web communication specifications in Java. Amongst other specifications these are JavaServer Pages, Java Servlet and Java API for WebSocket [18].

2.5 Representational State Transfer - REST

REST is a set of design principles for developing web APIs, it was created in 2000 and functioned as a guide during the development of HTTP 1.1. Despite the connection with HTTP, REST can use other communication protocols as long as the implementation stays RESTful [19].

A REST API should have a resource oriented architecture as opposed to a service oriented architecture [20]. When REST is implemented with HTTP, resources are combined with HTTP methods to address them. A resource can be added, updated, retrieved and deleted as represented by the most commonly used HTTP methods `POST`, `PUT`, `GET` and `DELETE` [21], there are a couple more methods but they are not used as often, especially not with REST.

2.6 HyperSQL database

HyperSQL, or HSQLDB, is a relational database written in Java [22]. Because of the fact that it is written in Java it can be run embedded on a Java server. Therefore, the need for an adapter is removed which would otherwise be necessary if using two different languages.

The database is multithreaded and thus supports concurrent transactions. This is suitable for a server as each new request to it will generally get its own process, or thread as it is called in Java. By using a multithreaded database each of these threads can talk to the database at the same time to reduce response times seen from the client's point of view.

2.7 Optimisation models

To solve the optimisation problem, three types of models were investigated further. This section contains a brief explanation of each one of them in their original form. More information about the reworked problem and how to solve it can be found in *4.4 Optimisation algorithms*.

2.7.1 Bin packing problem

The bin packing problem (BPP) is a well known combinatorial optimisation problem used to model and solve different types of packing problems [23]. The basic heuristic of the BPP is relatively simple to understand. A set of objects should be packed into bins. Initially, all bins have the same fixed capacity c and every object has a volume v, where the volume of one single object can not exceed c. The objective is to minimise the number of bins needed to contain all objects in the set [24].

2.7.2 Integer linear programing

Linear programing (LP) is a type of mathematical optimisation and a part of convex optimisation theory. The objective of a LP is always to optimise a linear function in subject to a set of constraints through minimising or maximising. These constraints form sets of inequalities limiting the function to obtain any solution in the feasible set [25]. For the program to be solved as a LP the objective function as well as all the constraints must be expressed as linear functions.

An Integer LP (ILP) has the same procedure and goal as a LP but with one extra constraint limiting one or more variables to be integers instead [26] [27]. In those cases decision-making is involved, a special form of ILP is used called Binary LP. In Binary LP (BLP) the integers are even more restricted and can only obtain the values one or zero [28].

2.7.3 Cycle transformation

Cycle transformation is a method to settle multiple debts used by Tom Verhoeff at Eindhoven University of Technology. The problem can be modelled as a directed graph where the nodes represent persons and the edges represent debts. If there are more than one edge between two nodes, they should be summarised and represented by a single edge. Optimisation can be done by looking for cycles within the graph and subtract the weight of one edge from all edges in the cycle [29].

3
Method

During the project several methods and tools were used. These are presented in the following chapter and include the programing languages, development environment, external libraries and work process.

3.1 Programing languages and formats

During the development of the system a couple of different languages were used. As Android runs Java [30] and uses XML [31] for the visuals these were the obvious languages of choice for the front end. Since all team members were familiar with Java, this also became the language used by the back end.

On both the back end and the front end databases are used. These both use Structured Query Language, or SQL, as they are both relational databases. Because of previous experience with HyperSQL, this was used in the back end. In Android it is however necessary to use SQLite which is why this was used in the front end [32].

When sending data between the front end and the back end JavaScript Object Notation (JSON) is used. JSON provides a key-value mapping which can easily be parsed by machines but still have the advantage of being easily readable by humans [33].

3.2 Development tools

The implementation of the back end and the front end was done in different IDEs, or Integrated Development Environments, which are applications typically used when developing software. Eclipse Java EE IDE for Web Developers, which supports running a Tomcat server from within the IDE, was used for the back end implementation. For the front end Android Studio was used as it currently is the official IDE for Android development [34].

To allow all team members to collaborate and work in parallel with the code a versioning control system was used. The system used was Git as all members of the project were familiar with it. In addition, an application called SourceTree was used which is a UI based Git client.

During the design of the UI a mock-up tool called Justinmind was used. Justinmind is web based and allows for the creation of fully interactive prototypes which can be designed to follow Android's Material Design, amongst other styles.

3.3 External libraries

A couple of different external libraries were used in the development of the system. JAX-RS is a library for creating RESTful web services and is part of Java EE 6 along with higher versions [35]. By using JAX-RS with the external library Jersey large parts can be abstracted away to simplify the implementation [36]. Amongst things abstracted away are the low-level details of the client-server communication. Furthermore, Jersey also provides additional features to allow for a more customisable API.

For the front end Retrofit was used to simplify the communication to the server. Retrofit turns a REST API into a simple Java interface using annotations [37]. It also translates between Java objects and JSON, which is helpful when communicating with the back end.

As the system allows users to login using Google as an alternative to the normal accounts, OAuth 2.0 using Google APIs Client Library for Java was used. To authenticate a user the front end has to obtain an access token from the Google servers. When the back end receives the access token from the front end it has to ask the Google servers if the token is valid [38].

To send notifications from the back end to the users' devices Google Cloud Messaging (GCM) was used. By using GCM the back end only needs to pass the message to the GCM servers, which then in turn handles the queuing and sending of notification messages to the users [39].

To ensure the correctness of the exposed methods in the back end the testing framework JUnit was used. JUnit allows for writing repeatable tests on Java code [40].

3.4 Work process

This section contains descriptions of which stages the project went through as well as different working methods that were used.

3.4.1 Planning and modelling

In the beginning of the project a time plan was created to make sure there was a good balance between the different stages of the project. We created a gantt chart and specified four milestones that can be seen in *Appendix A Gantt chart*.

The first milestone was to have written requirements and specified how the different features should work. This was done using a domain model, use cases and further documentation which could help with making the implementation easier to carry out. However, the documentation will be further discussed in chapter *4 Requirements and modelling*. The second milestone concerned the optimisation. At the finish of the milestone the optimisation had to be modelled and ready for implementation.

The implementation of the application was divided into two sub-milestones; 3a: The front end should be implemented and tested and 3b: The back end should be implemented and tested. Since the progress of these were dependent on each other, they were made parallel processes. The last milestone consisted of moving the back end to an online server and have an application ready for potential release. Other activities, such as writing the report, were planned in addition to the four milestones.

3.4.2 Implementation

As the system consists of both a client application and a server based back end it became clear that these were the two major parts which could be worked on in parallel. Therefore, the team split up between these two parts, effectively creating two smaller projects within the main project. In both of these groups the project was in turn divided into smaller parts which often could be worked on in parallel. However, each member of the project has some knowledge about the overall structure of the complete system and not only the parts they implemented themselves.

3.4.3 Scrum

To help structure the development process and the division of workload, Scrum [41] was used. Therefore, there was a backlog with features which needed to be implemented and a weekly meeting where the previous features were reviewed and the next set of features to be implemented were decided. There were also a couple of short meetings each week where members could give a short update on how far they had gotten with their tasks.

3.4.4 Gitflow

When using Git there are a number of different ways to work. We chose to use a workflow called Gitflow. When using Gitflow the project has to be structured in a specific way and a specific set of branches has to be used [42]. As can be seen in *figure 3.1* there is a branch called “develop” which is used during the development of the project. When a new feature is to be implemented, a new feature branch is created with a name that describes the feature. These branches are then merged back into “develop” and deleted when the feature is finished.

When the developer branch contains a project suitable for release a new branch is created for that release, as can be seen in *figure 3.1* In the release branch the code is finalised and polished in preparation for the actual release. Some examples of what this implies is making the code more readable by writing comments and refactoring variable names and fixing any bugs. After doing this the release branch is merged into the master branch and then deleted. If any bugs are found in the master branch a hotfix branch can be created and used to fix the bug and then be merged back into the master branch.

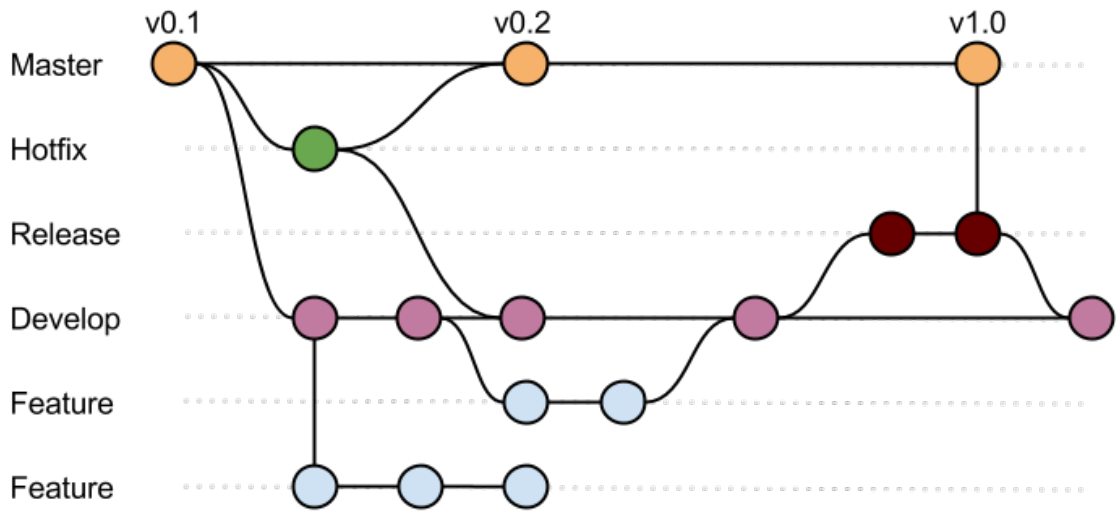

Figure 3.1: Concept of Gitflow branch management between release v0.1 and v1.0. A hotfix can be seen which created the release of version 0.2. Two features can be seen as well, one of which has been finished and merged into the develop branch.

3.4.5 External guidance

Ger Garrigan and Björn Thalén from Jeppesen agreed to mentor the project and provided some of their expertise where it was necessary. The reason why we asked Jeppesen for guidance is because they are an IT company which, amongst other ares, specialises in optimisation [43]. We had a total of four meetings with them where we showed the current progress and they provided feedback.

4
Requirements and modelling

As already said in section *3.4.1 Planning and modelling*, documentation was created prior to implementation. A list of requirements *Appendix B Requirements* was compiled where each requirement was assigned one of five different priorities. Priority one was the highest priority and priority five was the lowest. Requirements with priority five were not planned to be finished but were instead seen as potential future features. The requirements were primarily created from the project members' wishes, but also by reading comments from similar applications to see which features the users thought were missing.

4.1 Description of the domain problem

By using the requirements a description of the domain could be created. Below follows the domain description of the system.

Every person using the system is represented by a user. Each user is identified by a unique ID, in this case their email address. A user will also have a name, a profile picture and an optional phone number which is all public information any other user can see. Furthermore, all users will have a password which in combination with the user ID is used to log in. However, some users will not have a password as their login is handled by a third party service like Google. Each user has a number of friends and new friends are added by friend requests.

Two or more users can be involved in a single debt. Except an amount for each user, a debt also has a description, a timestamp and information about who added it. An optimisation algorithm takes all debts and tries to lower the number of transactions needed while still maintaining all users' current balance of in- and outgoing debts. Secondly, the algorithm tries to lower the size of all transactions without interfering with the number of transactions. This results in optimised debts between friends.

Normally a debt is added to a standard global network, consisting of all the application's users, and optimised with the whole network in consideration. However, a debt can also be added into a closed group. A closed group is a small network which is not a part of the standard network and will be optimised separately. Each closed group has a name and members. If desired, the closed group can be created to have admins which then are the only members that can add new debts. A closed group can also contain simple users, which are users for persons who do not have the application. Simple users are handled by the other members of the group.

When a debt is added it can be facilitated by using a quick selection list. It is a faster way to add several persons when creating a new debt. A quick selection list is created by and belongs to a single user. The list has a name, which is unique amongst the lists the owner has, and a number of friends that will be selected when using the quick selection list.

To notify the user that something has changed in the application, an event is created and a notification is sent. An event is personal for a user and this means that when something involving multiple users occurs multiple events are created, one for each involved user. These events are created when friend requests are sent and accepted, when debts are added or removed, when groups are created and when the user is added to a group.

4.2 Components and structure

The system can be divided into four components; a front end, a back end, a database and an optimisation algorithm. These are interchangeable, meaning they can easily be replaced as long as they support the same functionality. The database and the optimisation algorithm are, however, a part of the back end. Any new database must have the same visible tables as specified by the database schema and a new back end must implement the same API as the current one. The optimisation algorithm only needs to allow new debts to be added whereas nothing depends on the front end's implementation.

The back end runs on a remote server. To be able to communicate over the Internet the back end is implemented as a RESTful web service. As the back end cannot simply hold all data in memory, as this would not scale very well, it uses a database to store the data. When appropriate the back end uses the optimisation algorithm to optimise the current debts, which in turn uses the database to update the necessary tables.

In contrast, the front end runs on the user's phone and only holds some cached data and application settings. It uses the back end's API to fetch data the user wants to view. Requests are sent to the back end when the user changes something which needs to be mirrored in the back end.

As can be seen in *figure 4.1* a typical data flow starts at the front end when the user requests to view something. The client sends a message to the back end which in turn queries the database and then sends a response with the relevant data to the client. The optimisation is performed by a separate process from the back end where the result of the optimisation is added directly to the database. The back end triggers the optimisation whenever a new debt is available. The next time the client requests optimised debts the back end retrieves the updated debts from the database.

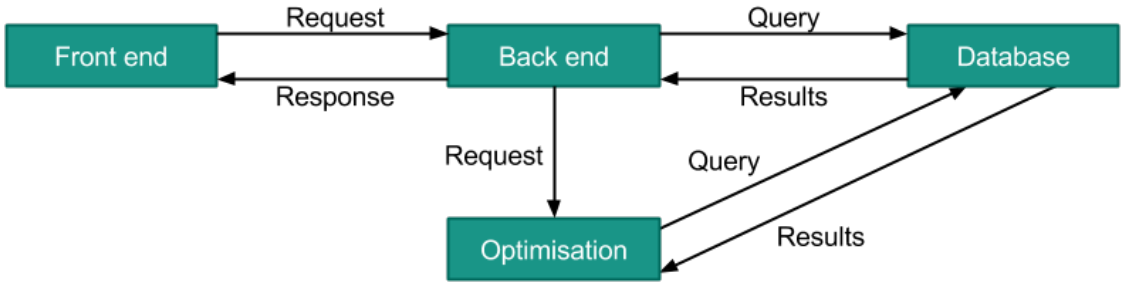

Figure 4.1: The dataflow of the system.

4.3 Documentation process

Using a list of planned features and the description of the domain problem an API document with all required methods was created. The document specifies all input and output parameters and all constants used in the communication between the front end and the back end like status codes and type identifiers. The methods in the API were grouped after the type of resource they managed.

An entity-relationship diagram was created to describe the data stored in the database and how the entities relate to each other. Each type of resource has to have their own table and can contain references to other types or resources.

4.4 Optimisation algorithms

In order to know both how to model the problem and which algorithm to use it is required to know the goal of the optimisation and the constraints on the solution. The primary goal of the optimisation is very simple and easily defined: there should be as few transactions as possible. The secondary goal is that the sum of all transactions should be as small as possible. There is also a tertiary goal: the algorithm should not change an optimised debt unless necessary. If there are two equally good solutions the algorithm should use the solution which is most similar to how the network looked before optimisation.

There are some easily defined constraints, for example the total balance of a user's debts should be the same before and after optimisation. Another constraint which is clear from the domain is that a user should only have optimised debts to friends. Also, a user is not allowed to have a larger sum of outgoing debts after an optimisation than before. This is necessary in some specific circumstances to avoid large amounts of money to be channeled through users which were never part of the large debts in the first place. The reasoning will be discussed more in *6.3 Optimisation*.

There are two constraints without clear definitions that depend on outside variables. One of the constraints is that the algorithm should be fast, a user should not experience a noticeable delay between adding a debt and receiving the updated optimised debts.

The second constraint which does not have a clear definition is about the correctness of the optimisation. The algorithm does not have to find the absolute best solution. The requirement is that the solution should be good enough that the users should not be able to see a better solution themselves.

To find an algorithm which is able to reach the goal, taking the given constraints into account, several algorithms were explored. Below the algorithms are described and it is investigated if they are suitable or not.

4.4.1 Bin packing problem

When adapting the BPP to the mathematical model constructed to solve this specific problem, some fundamental changes to the structure of the BPP are necessary. The initial state of the problem was reformulated in a way suitable for the domain.

The debts to be paid correspond to the volumes of the objects. Mathematically, the value of every object was defined by one balance with a positive value. Together, all the balances with negative values represented the bins. This resulted in the first significant change of the BPP. Instead of a discrete finite set of bins, there now exists a fixed number of bins not all with the same capacity. In the original definition explained in section *2.7.1 Bin packing problem*, one of the basic constraints in the model is restricting every single object to not exceed the bin capacity. This constraint is removed since nothing in the domain of this problem indicates this behaviour.

The most significant change in the reformulation of the BPP is the objective of the model. It is still a combinatorial minimisation problem but instead of placing the objects in a way where the number of bins used is minimised, the goal is to place the objects in a way where we fill all the bins by dividing as few objects as possible. If there is a solution without any division this is the optimal solution, otherwise the solution with the fewest divisions is optimal.

No proper constraint have been found to express the properties of friendships without leaving the scope of the BPP. This is a problem due to the necessity of this part of the model.

Optimisation can be achieved through a number of different types of algorithms that have been developed to solve the BPP. The Best Fit (BF) algorithm puts the item in the bin where there is least amount of room left after the item have been placed. If no bin can contain the object, a new bin is opened. In contrast, the First Fit (FF) algorithm takes the first object and tries to place it in the first bin available. If the object does not fit, the algorithm checks the next bin in the list and keeps going until a placement can be made. If no bin can contain the object, a new bin is opened.

When using the model previously described in this section the accessible data is a set of bins and a set of objects corresponding to the balances of the users. The first step is to sort both sets in a decreasing order. From there it should be easy to find all objects which have a bin of the exact same size with the help of the Best Fit algorithm and place them together. This will result in the first set of optimised debts.

To accomplish the goal of avoiding splitting objects the next step is to find all pair of objects with a combined size of a bin, resulting in two optimised debts. Among the objects not yet placed in a bin, one could search for an arbitrary number of objects with the combined volume which can be matched to a not yet filled bin. However, the complexity will increase exponentially when searching for larger combinations.

All remaining objects not yet placed in bins are moved to a second phase of calculations. Similar to the First Fit algorithm the first object in the list is placed in the bin with the most capacity left. If the volume of the object exceeds the capacity of the bin, the object is split in two. This results in a new object placed using the same strategy as in the previous paragraph. If no progress is made with the adopted Best Fit algorithm, the new object is sorted into the list of remaining objects and the process is repeated until no objects remain and all bins are filled.

When solving the problem with this method, a feasible solution is obtained. However, when searching for an optimised solution, this method results in an unreasonable high time complexity. Furthermore, as mentioned before, it does not seem possible to model the concept of friends without leaving the scope of the BPP. Thus it was decided not to proceed with implementing this algorithm.

4.4.2 Integer linear programing

Based on the problem domain, friendships are modelled by an undirected graph $G = (V, E)$. Let every person using the application be represented by a node $v \in V$. Let a friendship between two users be represented by an edge $e \in E$. Since the graph is undirected, $e_{i,j} = e_{j,i}$ where $e \in \{0, 1\}$ and

$$e_{i,j} = \begin{cases} 1 & \text{if } v_i \text{ and } v_j \text{ are friends} \\ 0 & \text{otherwise} \end{cases}$$

Let every user have a balance $b \in B$ in such a way that $\forall v \in V, \exists b : b \in \mathbb{R}$, where the value of b_i is the sum of all outgoing and incoming debts for user v_i. For every new outgoing debt a user adds, the balance decreases. For every new incoming debt a user get, the balance increases. From the set of balances B, we specify two subsets $B^+ \subset B$ and $B^- \subset B$, where $B^+ = \{b > 0\}$ and $B^- = \{b < 0\}$. This is done to separate the users in debt from the users with credit.

To model the amount of money transferred, a variable t is introduced where t_{ij} is the amount transferred from person i to person j. $t \geq 0$, $t \in \mathbb{R}$. For the BLP to match the domain, the following constraints are formulated:

$$\sum_i b_i = 0 \tag{4.1}$$

$$\forall i : \sum_j t_{i,j} = b_i^- \tag{4.2}$$

$$\forall j : \sum_i t_{i,j} = b_i^+ \tag{4.3}$$

$$\forall i, j : (1 - e_{i,j}) t_{i,j} = 0 \tag{4.4}$$

Equation 4.1 ensures that the sum of all debts is equal to the sum of all credits. 4.2 makes sure that we do not send more money from one person than this person has in debt and 4.3 that we send exactly as much money to one person as this person has in credit. Equation 4.4 ensures that no transfers can be made between two persons that are not friends and results in $e_{i,j} = 0 \Rightarrow t_{i,j} = 0$.

Last but not least, we define the objective function:

$$min \sum_{i,j} func(t_{i,j})$$

which will minimise the sum of $func(t_{i,j})$, used to identify the transfers. The value of $func$ is determined by:

$$func(t_{i,j}) = \begin{cases} 1 & \text{if } t_{i,j} \geq 0 \\ 0 & \text{if } t_{i,j} = 0 \end{cases}$$

which will increase by one for every transfer made and remain unchanged if no transfer is made between person i and j.

There exist many different algorithms and methods to solve a ILP. Instead of implementing a complex algorithm from scratch to solve this problem, some time was spent on finding and testing external libraries which could provide APIs to solve this form of linear programs. The API investigated further was the free *LP_Solve* API [44].

LP_solve uses two different methods in two steps to solve ILPs [44]. The first step in finding an optimal solution is to search for extreme points in the set of feasible solutions [45]. In the part of the LP containing real numbers, LP_Solve uses an algorithm called the Simplex algorithm [46] to do this. The next step is to restrict the solution to contain only integers. This is done through a method called Branch and Bound which search for the optimal solution by discarding possible solutions after comparing upper and lower bounds [47].

The solutions obtained from this library are optimal. However, this library can not calculate a result fast enough when the number of users increase. For every debt added the algorithm needs to recalculate. When recalculating, Simplex take every value in the entire network into account to find a solution, resulting in a long computation time. It is unsuitable to use an algorithm that takes too long time, even though it calculates an exact result. Therefore we chose to not proceed with this algorithm and instead search for an algorithm that does not recalculate every value when not necessary.

4.4.3 Cycle transformation

A third suggestion for optimisation algorithm is a reworked version of cycle transformation explained in *2.7.3 Cycle transformation*. The main structure with a single graph containing nodes where every user is represented by exactly one node remain the same. There is however a difference in how the edges are handled because of the fact that friendships are necessary in the model.

Every edge represents exactly one friendship between two users. If there is an optimised debt between two friends the weight of the edge shows the size of the optimised debt and the direction shows who should pay. This means that when a new debt is

added to a friendship the weight is added to the already existing edge rather than creating a new edge. If there currently does not exist an optimised debt between two friends the weight of the edge will be zero and the direction of the edge is irrelevant.

This method is the most promising and nothing indicates that it has too high time complexity. Since all constraints are fulfilled and the optimised solution is meeting our requirements we decided to implement this algorithm which will be explained in *5.3 Optimisation using cycle transformation*.

5
Implementation

This project was mainly focused on implementation since all members were interested in seeing a version of the application potentially ready for release. In this chapter, design and implementation for both the front end and back end are handled, along with implementation of the optimisation.

5.1 Front end

The basic structure of the application is explained here, along with the implementation process, how the communication with the back end works and interesting implementation techniques used in the front end.

5.1.1 Implementation process

The design process for the UI involved decision making regarding features, research of user interface guidelines, creation and evaluation of a prototype and implementation of the application. The process was iterative and the UI was informally tested by the group and their friends throughout development to make it as user friendly as possible. The primary tests were done by group members working with the back end.

The features with higher importance needed to be accessed easily and were therefore placed where they were most accessible by the user. All views in the application were decided during this process by placing the features where they were most suitable. One feature at a time was implemented and tested in parallel with the implementation of the back end.

The research phase included research of what the guidelines for Android applications were. All guidelines could be found at Android Developer Guide. It was decided that the design for the application should follow Material Design for Android 5.0 Lollipop.

The creation and evaluation of a prototype was done by discussing and visualising every view. The visualisation was done using the desktop application Justinmind. Justinmind is an application with the possibilities to create and interact with a visual prototype of the UI for both mobile devices and websites. This tool was used to create most of the views and the interaction between them to get an overview of how the application should look and feel.

A theme for the application was created as a first step to simplify the implementation and to make sure the colours, fonts and sizes were the same for the whole application. It also made it easier if changes to any of these had to be done as any changes would automatically be made for the entire application.

5.1.2 Client-server communication

For the communication between the back end and front end we use REST. The back end holds a REST API which the front end connects to using a framework called Retrofit. Using Retrofit the developer just needs create an interface which will represent the REST API. After setting up the Retrofit interface a call to the server is almost as easy as calling any other function in the project.

There is, however, one thing that needs to be addressed when making network calls on Android. Android does not allow network calls on the main thread. If heavy work is done on the main thread, which handles the UI, the UI will become slow and unresponsive. This means that all the communication between the front end and back end has to be made in separate threads. Retrofit handles this by using either callback or asynctask, both of which are used in the implementation of our system. A callback is an object that is passed into a method of the Retrofit interface and returned with success or failure, depending on the response from the back end. Asynctask is Androids way to simplify the start of a new thread.

5.1.3 Structure

When the application is started, an activity is initialised which checks if a user is already logged in or not. This activity then starts either the main activity or the login activity depending on whether or not the user is logged in.

When logged in, the application consists of a main activity where different fragments are displayed depending on how the user interacts with the application. All activities and fragments are connected to a layout XML file which contains all elements visible to the user. What the user chooses in the navigation drawer, the side menu, decides which fragments should be shown. The views not directly reachable from this menu are started as an activity. Examples of such views are views for adding a debt, creating a group and creating a quick selection.

A number of different model classes are used for storing information about different objects. In the model several response classes are stored as well. These represent the responses received from the server when calling on methods through the REST API, more about these responses can be found in chapter *5.2.6 Status codes*.

Apart from the previously mentioned structure there is a package for utils, where classes for authentication, internal storage and autocompletion for text fields can be found. The internal storage contains cached data about the logged in user and and provides easy access to cached friends of that user. All cached data makes the application faster, since they are used in several places in the application and thus render it unnecessary to call, and wait for, the server each time the data is needed.

5.1.4 News feed

The events concerning a user are shown as a list in a news feed. All of these events are sent from the back end as a list where each event has a specific type. Before presenting the events to the user a check is made to see which kind of event it is and based on the result the correct information is displayed to the user.

Because of the fact that events have different types they need to be parsed differently. An event about a debt contains an object representing the debt, while a friend request contains a user which needs to be parsed as such. The parsing is done through a custom adapter which handles all events separately to parse them correctly.

5.1.5 Notifications

Notifications are sent through Google's server and received by a broadcast receiver. The receiver then receives a wakeup event and keeps the device awake while the work is passed on to a service.

The service works asynchronously with the message and creates different notifications depending on the message type. The most advanced notification handled by the service is when a friend request is received. When shown to the user as a notification, the user can choose to immediately accept the request.

5.1.6 Search suggestions

To implement search suggestions Android's search widget was used. To use this standard search widget the suggestions needed to be in a SQLite database inside the application. This means that when the application starts the data about the user's friends is fetched from the back end and stored in the internal database. The internal database is updated whenever the user makes a change to their friends. The Android search widget uses a content provider which acts as a link between the search view and the database [48].

5.2 Back end

In this section the implementation of the back end will be described. Some of the most interesting aspects of the back end will be discussed in more detail.

5.2.1 Implementation process

As previously described in *4 Requirements and modelling*, there was a lot of focus on documentation before the implementation began. Using the API document created during this process, a framework could be constructed which only returned the standard HTTP status code 501, standing for not implemented. Using the framework the required features for each sprint were implemented iteratively until every planned method had been implemented or removed from the documentation due to new circumstances.

A schema of the database was created from the ER-diagram of the database. The schema described the diagram in textual form and was translated into SQL that could be compiled into a database.

5.2.2 Client-server communication

The communication between the back end and front end uses REST. The translation between the URL sent with the request and the path to the Java method is handled by Jersey. The method needs to be annotated in the code for it to be found by Jersey as well as obtain the correct parameters when called on, as can be seen in figure *figure 5.1*

```
35 @Path("/user")
36 public class UserHandler {
   ..................

52⊖    @Path("/create/{email}/{password}/{name}/{phoneNumber}")
53     @POST
54     @Produces(MediaType.APPLICATION_JSON)
55     public Response createUser(
56             @PathParam("email") String email,
57             @PathParam("password") String password,
58             @PathParam("name") String name,
59             @PathParam("phoneNumber") String phoneNumber) throws JSONException {
```

Figure 5.1: Shows the declaration of a class and a method with the appropriate annotations.

Each connection to the back end receives its own process, or thread. As the back end is multithreaded it is important to create thread safe methods that cannot break the system if interleaved by other methods. The advantage of using a new thread for each connection is that a heavy request with a long execution time can be run in parallel with smaller requests instead of simply halting the system until it is done.

5.2.3 Structure

Each group of methods handling the same type of resource has their own handler. Therefore, the methods handling events and quick selections have their own handlers accessible through the subpaths *event*, *notification* and *quickSelection*. Groups containing a large number of methods were divided into subgroups, even though they handled the same type of resource. The methods handling users were divided into the two handlers *friend* and *user* and the methods handling debts and groups were divided into three different handlers; *debtCredit*, *groups* and *groupPayment*.

As already described in *5.2.2 Client-server communication*, each request is run in a separate thread. Beside these threads and the main thread creating them, there are two additional threads. The first is the optimisation thread which performs the optimisation on the new debts added since it last ran. The other thread is a service which periodically sends out updates to users about the number of debts added and the total sum of these.

5.2.4 Database

As the data stored in the back end needs to be persistent if the server would crash for some reason or be turned off, a database is used. The database used is a HyperSQL database which runs embedded in the back end application to minimise query times. The database automatically handles the creation of ID-numbers for those resources with no other logical unique identifier. For example two debts with the same amount and description between the same two users are not unique, whereas a user always has a unique email which can be used as their identifier.

It is important to handle the database correctly when working with a multithreaded system. To avoid uncommitted dependencies, where a thread reads data between another thread's commits, each thread must have their own connection to the database. This was done with a pool of a hundred connections. If all connections are occupied when a new request from the front end is received, that thread has to wait until a connection is available.

5.2.5 Authentication

Even if secure connections are outside of the scope the system needs a way to check which user sent which request. This is accomplished by an authentication token that is being sent from the front end to the back end when sending a request. The token is constructed differently depending on whether the account is handled by our system or if it is handled by a third party service like Google. In the back end each request is passed through a filter which checks this token and parses it to validate the username and password. The username is then added to the request so the addressed method can access this information.

5.2.6 Status codes

If an error occurs in the back end, the client needs to be notified. When errors appear in Java they are typically handled by throwing exceptions that the user of the method can catch. However, in the server-client architecture there is no need to send the whole exception as an object as much of the data it holds will not be of any use for the client. Instead of sending an exception a status code is returned together with the actual response from the method. These types of responses work like envelopes as they encapsulate a response and hold extra data for that response.

5.3 Optimisation using cycle transformation

The optimisation algorithm which is used in the application is built on the same principle as cycle transformation [29], namely to find and optimise cycles. By changing the weight of every edge within a cycle by a specific amount, see the model in *4.4.3 Cycle transformation*, the number of transactions and the sum of all transactions can be lowered while all constraints still hold true.

When a new debt is added to the optimisation algorithm, the amount is added to the weight between the nodes representing the involved users. If the edge between users A and B have the weight zero and a debt is added from A to B the edge will get the same weight as the amount and point towards B. When a second debt is added the weight will be modified by the amount of the second debt. If the result is negative, the weight is multiplied by minus one and the direction of the edge is changed. If the new debt involves more than two users the debt will be split up into multiple debts including only two users for the optimisation.

If the weight equals zero after a new debt has been added, the algorithm will not be applied as this is equal to a debt being paid. In a network of optimised debts a payment will not open up to a better optimisation. In all cases where the weight is not equal to zero after a debt is added the algorithm will be applied with respect to the two users involved.

5.3.1 Finding cycles

The first step of the optimisation algorithm is to find all relevant cycles. The goal is to find all lists of users which start and ends on the users involved in the added debt and where all users in the cycle are friends with both adjacent users. With the model we are using, this translates to finding all cycles in an undirected equivalent of the graph where both involved users are adjacent to each other.

The process of finding all cycles starts with one of the involved users as origin. The path to the first involved user is very simple and only contains the node representing the first user. All discovered paths for all discovered nodes are saved in a temporary data structure P and all nodes which have paths that can be continued on is saved in a separate list U. At the start P only contain the first node and only one path for that node. U also only contains the first node.

P - All discovered nodes and all discovered paths to the same nodes.
U - All nodes which needs to be checked for path building.

The algorithm will then continue building on every path in P leading to the node currently being built on in U. When U is empty there are no more cycles to be found. Nodes will be added and removed from U during the algorithm.

c - The node in U currently being used for path building.
n - The current node connected to c being used for path building.

The building process consists of taking every node n connected to the current node c. Then all paths leading to c which exists in P are copied and n is added at the end of all copied paths. If n already exists in a path, that path is not copied since a path is not allowed to contain duplicates. If this is the first time n is discovered, n is added to P. Then all copied paths are also added to P. Paths which already exists in P are ignored. If at least one path is added to n and U does not contain n, n will be added to U and as a result n will later be the central node of the building process.

If n is the second user involved in the debt, all paths leading to n is also cycles. The optimisation algorithm will try to optimise every cycle which is found, as described in section *5.3.2 Optimising cycles*. If the cycle can be optimised, the algorithm will restart with the newly optimised debts as the current iteration of the algorithm ends. If the cycle can not be optimised the algorithm will continue and keep looking for more cycles. After every node connected to c have been iterated over, c is removed from U.

If no cycle is found that can be optimised, the algorithm will keep looking for more cycles until stopped. In order to prevent the algorithm from going on forever, a max length of a cycle has been set. Paths will be ignored if they reach the max length, currently set to five, without creating a cycle.

An example of a debt network can be seen in *figure 5.2*. A new debt has been added between A and C. The algorithm will start looking for cycles with A as origin. This means that A is added to both U and P. In the the first iteration of the path building process all friends of A will be added to both P and U, and A will be removed from U. The process will continue looking for cycles through D and E but will not find any. When continuing with B's friends it will not find any cycles through G and H, but will find one cycle with three edges, A-B-C-A.

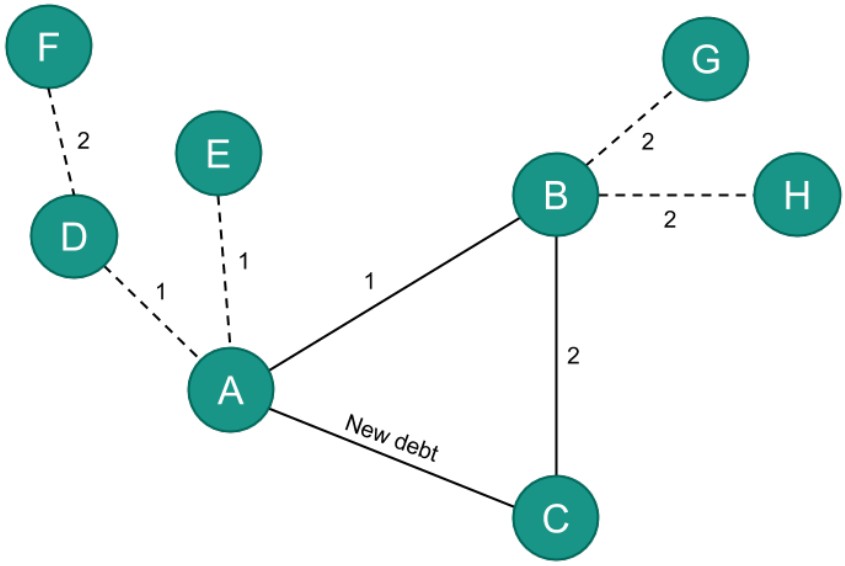

Figure 5.2: The numbers on the edges indicates the path building step. The cycle A-B-C-A has been found after two steps.

5.3.2 Optimising cycles

Every cycle found will be checked for a better optimisation. If a cycle can be optimised it is always possible to lower the number of debts by removing one of two values. The value is either the smallest weight of an edge in a certain direction or the smallest weight of an edge in the opposite direction. One of these two values can be removed from each edge in the cycle and lead to a reduced number of transactions. The cycle is being tried for both values, to find the most suitable optimisation.

Except for checking if the number of transactions decreases, the total sum of all outgoing edges from every node is checked. After the optimisation this sum should be equal to, or less than, the sum before the cycle optimisation. If the cycle optimisation would result in a larger sum, the optimisation will not be done. If the optimisation is performed, there are two statements which can be true. Either the total weight of the cycle is decreased or the number of transactions is decreased. In most cases where an optimisation can be performed, both of these statements will be true afterward.

When a cycle is optimised the current iteration of the algorithm is stopped. The algorithm is restarted for every pair of nodes in the cycle which have had the weight of their edge changed to something else than zero.

6
Result and discussion

The project has resulted in a working system with a back end which can optimise the debts added by the front end application. This section will describe the result of the project and discuss the different parts.

6.1 Front end

In this section the resulting application is presented including images showing what the final application looks like on the client. We also explain and discuss why different choices were made regarding the UI as well as some difficulties with the current design.

In *figure 6.1* the splash screen for the application is shown. The application's name is Debtimize and the logo can be seen in the figure. The colour scheme used in the application consists of cyan as the primary colour and deep orange as the complementary colour.

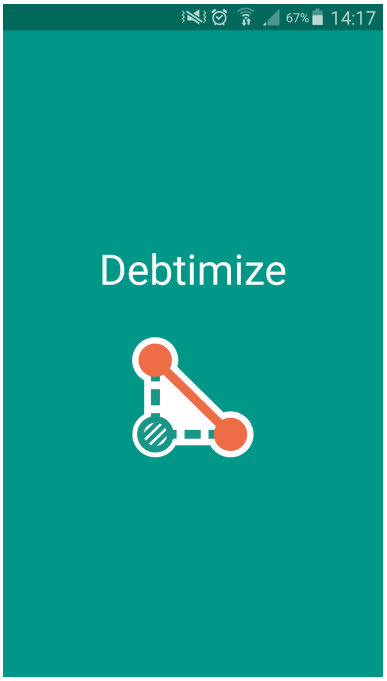

Figure 6.1: The splash screen shown when the application is started.

6.1.1 Log in and create account

The login screen, as can be seen in *figure 6.2* , is designed to easily be understood and does not have any disturbing or unnecessary elements. Users can log in as a normal user or use their Google account. Furthermore, if they have logged in before they will automatically be logged in until they explicitly select to log out within the application. If the user desires to create an account, this can also be done from the login screen.

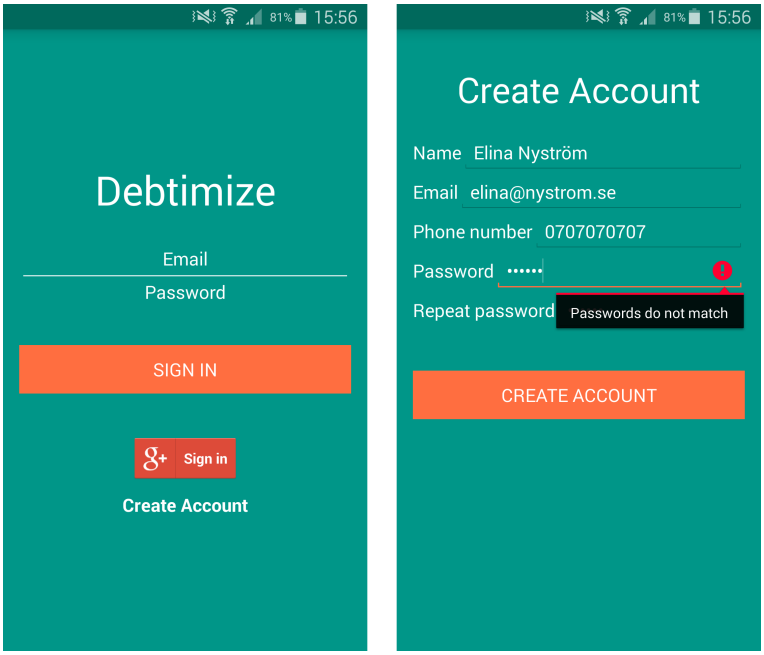

Figure 6.2: The view on the left is presented to the users on startup. The user can choose how to log in or to create a new account. If choosing to create a new account, the view on the right is presented.

6.1.2 Home screen

On the home screen the user can see the balance at the top of the screen, as can be seen in *figure 6.3*. The balance is green when the user should receive money and red when the user owes money. Since the balance is the most important information for the user to see, it is designed to be the centre of attention on the home screen. Under the balance the user can see how many debts they should pay or receive for their balance to be zero. The debts can be marked as paid or a reminder can be sent to the user who owes money. The details of an optimised debt is an important feature and can therefore be easily accessed from the home view. The most important feature of the application is to add debts, which is why a button for adding new debts is placed on the home screen according to Material Design.

The news feed shows recent events so that the user can quickly get an overview of what has happened since they last used the application. Some of the events are strictly informative and others are possible to interact with. If for example the user has received a friend request, it can be accepted through the event.

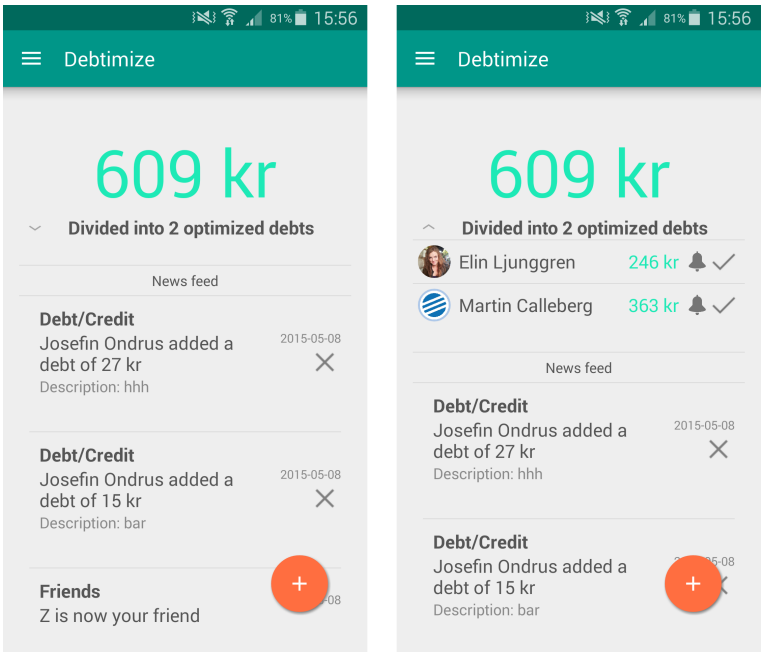

Figure 6.3: The home screen of the application. On the right the list of optimised debts have been expanded by the user.

The placement of the button for adding a debt, which is standard in Material Design, makes the event at the bottom of the news feed partly hidden. This makes it impossible to interact with the button to the right in that event. A solution that was discussed was to add swiping functionality to events which would remove the need for a button in each event. This was not implemented due to lack of time and other features being prioritised higher.

6.1.3 Navigation menu

To navigate in the application, a navigation drawer has been implemented as can be seen in *figure 6.4*. At the top of the menu user can see information about their account and go to a view where this information can be changed. Below the different views the user can navigate to are listed and at the bottom the user can log out of the application.

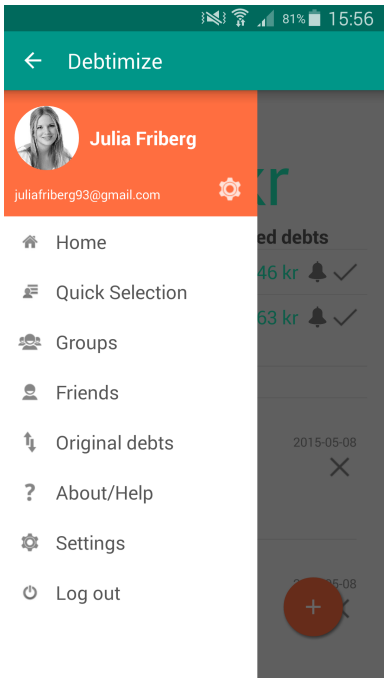

Figure 6.4: The navigation drawer, or menu, in the application.

6.1.4 Add debt

The view for adding a debt is easily accessed from almost any part of the application, probably most commonly from the home screen. When adding a debt there are a lot of different input data needed from the user. This view, seen to the left in *figure 6.5*, include input in the form of text fields, drop down menus/spinners, lists and radio buttons. To make it as user friendly as possible the components are placed in a logical order, the user most likely prefers to enter the amount of the debt first, and then continuing with the description and so on. The date of the debt is defaulted to today's date, but may be changed by choosing a new date with the help of a date-picker. The date-picker is an android element which is displayed as a calendar for the user to choose from, as can be seen in the second view in *figure 6.5*.

The user is able to choose if the debt should be added into the Network or in a group in which the user is a member. The friends involved in the debt are added through search suggestions and when added to the list the friend is also added to a spinner from where the user can choose who paid.

The last choice the user has to make is if the debt should be split equally between all involved users or not. If the logged in user chooses "Do not split", the debt will not be shared with the user who paid. If "Unequally" is chosen the dialog shown on the right in *figure 6.5* will appear. In this dialog the user is able to enter the percentage that should be paid by each user.

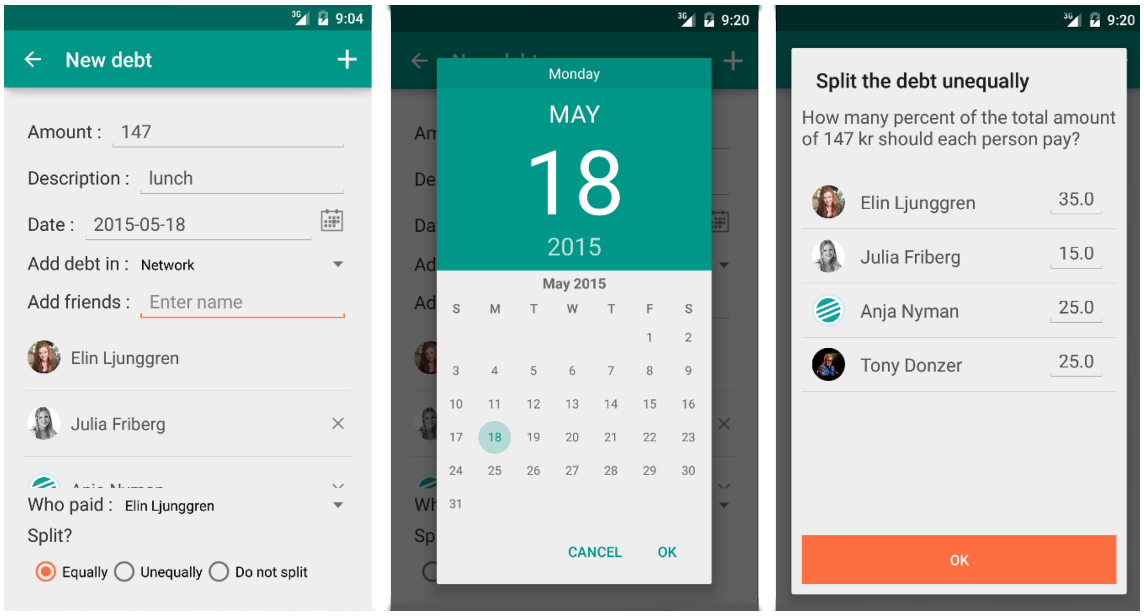

Figure 6.5: To the left is the view shown when adding a debt, the middle view is the datepicker and the view to the right is the dialog for unequal debts.

The add debt view was the view which took the longest time to implement. This was mainly because the definition of a debt was changed several times, making it necessary to change the layout of the view and adding more elements to gather all information needed. The functionality for adding a debt to several users was implemented very late in the project and made a big impact on this view. Unfortunately there was no time to make the view easier to use since all energy was put into implementing the new functionality. Preferably the inputs for date of debt, who paid and how to split the debt could be advanced settings. These settings should then be hidden until the user chooses to see them.

6.1.5 Original debts

Original debts are shown in tabs as can be seen in *figure 6.6*. The first tab shows all debts, the second all incoming debts and the third all outgoing debts. For each debt the amount is shown in red, if the debt is outgoing, or green, if the debt is incoming. Additional information about each debt is also displayed. The user can remove a debt using the cross icon or add new debts through the plus icon at the top.

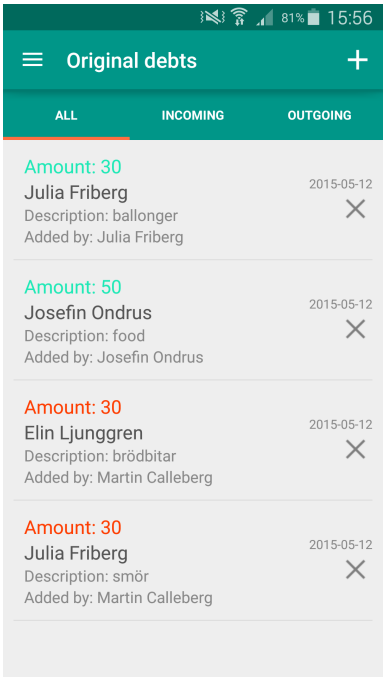

Figure 6.6: View presenting all of the user's original debts. Using the tabs the user can filter the debts to only show outgoing or incoming.

The functionality for seeing all original debts was added since a part of the purpose for the project was to make it easier for people to manage their debts. This is also why we choose to divide debts between incoming and outgoing. Another reason for adding the functionality was to make the user trust the optimisation by providing insight into where the optimised debts originate from.

6.1.6 Friends

The friends view presents all the user's friends in alphabetical order. Using the search field the user can find other users. To make it as easy as possible for the user, the functionality of custom search suggestions was added. The search suggestions present friends which match the search phrase. Other users can be found by pressing the search button. By tapping on one of the users more detailed information about that user is presented, as can be seen to the right in *figure 6.7*. This is actually a view requested by users of similar applications. In the detailed view the user can send friend requests to that user and, when the two users are friends, add new debts, see the shared debts and remove the friendship.

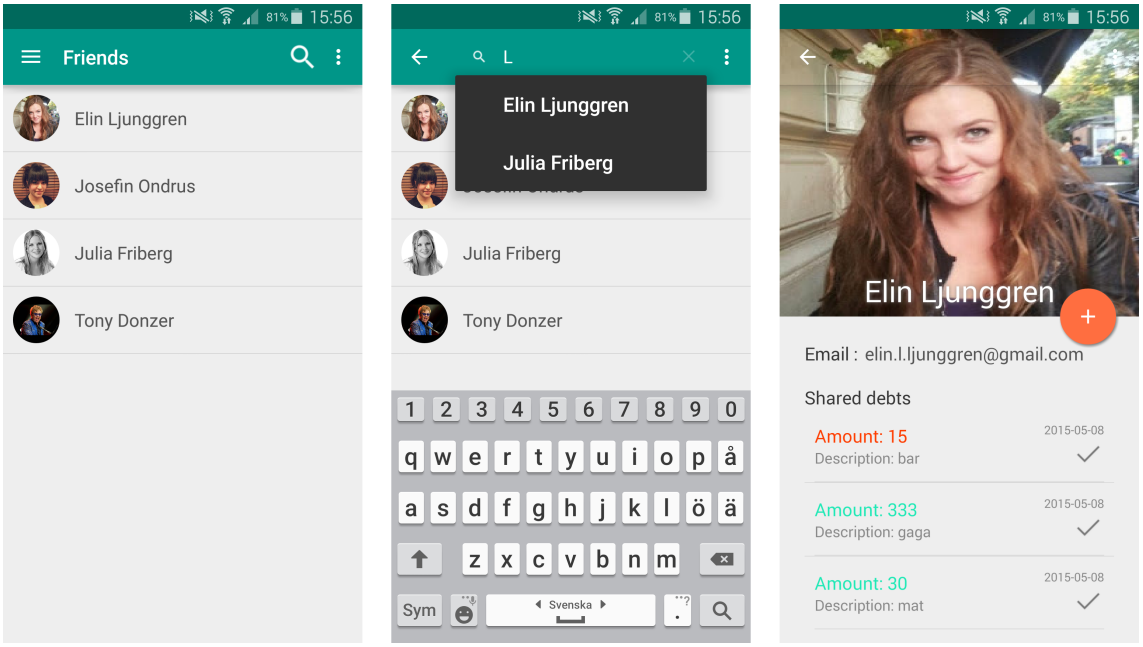

Figure 6.7: To the left the user's friends are listed, in the middle the user has started to search for users and to the right is a detailed view about a single user.

The way to find users who are not your friends is not as intuitive as it should be. It may confuse the user that the suggestions only show friends and not other users of the application. The reason for this implementation was that it could be experienced as overwhelming and unnecessary to get suggestions with all users containing the search phrase. This problem could be solved by adding a filter for showing the most relevant users. However, this was not prioritised since the time was limited.

6.1.7 Quick selection

Quick selection functionality helps the user choose several of their friends to add to a debt at once. Quick selections can be seen and added from the quick selection view, as can be seen on the left in *figure 6.8*. By tapping one of the items a list of the users in that quick selection is presented, the right view in *figure 6.8*. Changes to the quick selection can be done through a pen icon at the top and new debts can easily be added by clicking the button at the bottom.

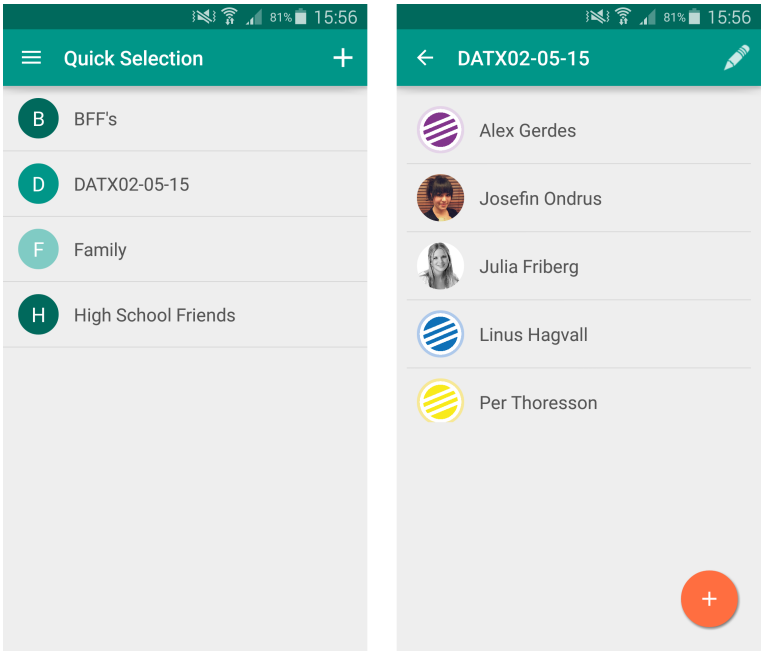

Figure 6.8: The home screen of the application. On the right the list of optimised debts have been expanded by the user.

The reason why this functionality exists is to lower the amount of excise. A quick selection can be created to ease the process of adding a debt to several friends that are frequently involved in the same debts.

6.1.8 Groups

Groups are listed the same way quick selections are. When viewing a single group, an overview is shown with the group's settings, its members and their balances and the optimised debts. This can be seen to the right in *figure 6.9*. When creating or editing an admin controlled group the user acting as administrator can choose if debts should be visible or not to the group members. This can be seen to the left in *figure 6.9*. If debts are chosen to not be visible, a user can only see their own debts. In groups without administrators the group members can see all debts. A debt can be added by clicking the button for add debt. In groups controlled by administrators, administrators are the only ones who can add new debts.

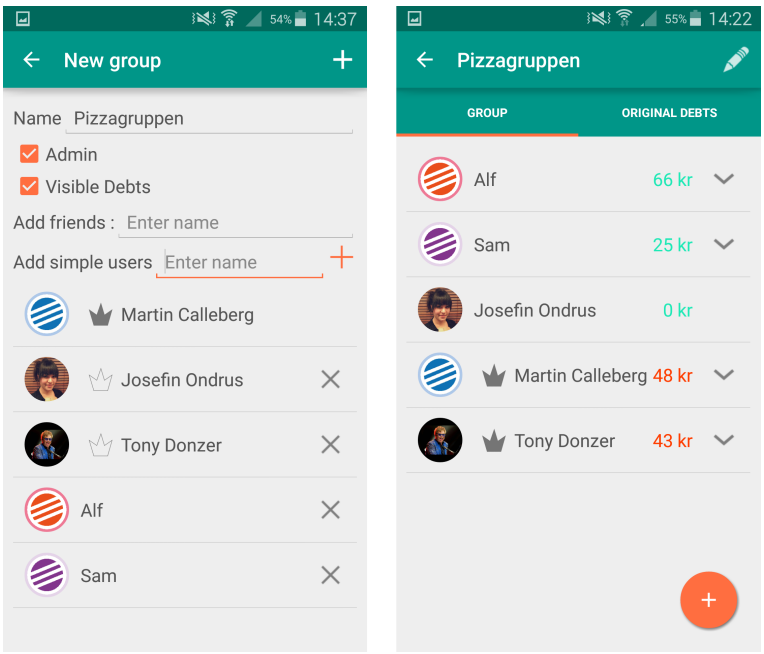

Figure 6.9: To the left a new group is being created and to the right the same group has been added together with a couple of debts resulting in the members having different balances.

We chose to separate original debts in the network and original debts in groups to lower the risk of confusion. This resulted in a tab in the group view showing all original debts deriving from that specific group.

Closed groups is a way for users to separate debts added within the group. For example if a group of people go on a trip they might want to isolate all debts during the trip. A closed group can also be useful within different associations where users might be interested in keeping the debts separated. Admin groups are useful when all members trust one user but not everyone trusts everybody else, for example in an association where everyone trusts the leader. There could also exist some users who are not comfortable with having debts in the network where they can be transferred more freely and instead only use closed groups.

6.1.9 User settings

In this view, reached from the top of the navigation drawer, the user's information is shown and can be changed. This can be seen in *figure 6.10* At the top, the profile picture for the user is seen and can be changed if the user clicks it. The user can then choose an image from the images saved on the phone. The profile picture is followed by the user information for the account. However, the email cannot be changed because we do not support this functionality yet. Furthermore, if the account is handled by a third party like Google the user cannot change their password from within the application. Moreover, when the user is debt free the account can be deactivated by pressing the deactivate account-button and is reactivated again when the user tries to log in with that account. When an account has been deactivated debts cannot be added to this user and the user is not shown anywhere in the application.

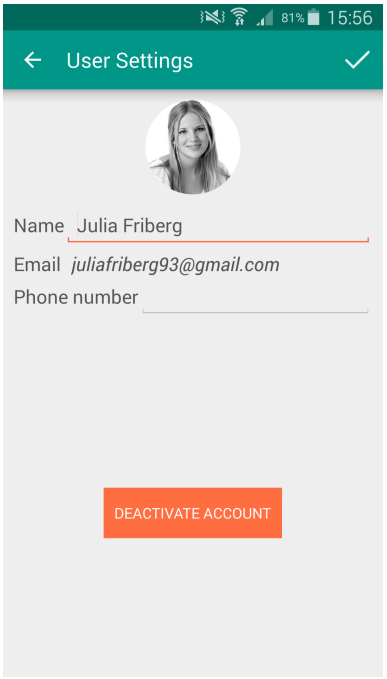

Figure 6.10: View where the user can change their profile settings.

6.1.10 Settings

The settings for the application are divided into two categories; notifications and general. This can be seen in *figure 6.11*. Under the notification category different types of notifications can be turned on or off. Under "General" there only exists one setting so far, choosing language. The application languages to choose from are English and Swedish. The standard language depends on the device and is Swedish for devices in Swedish and English for all other devices.

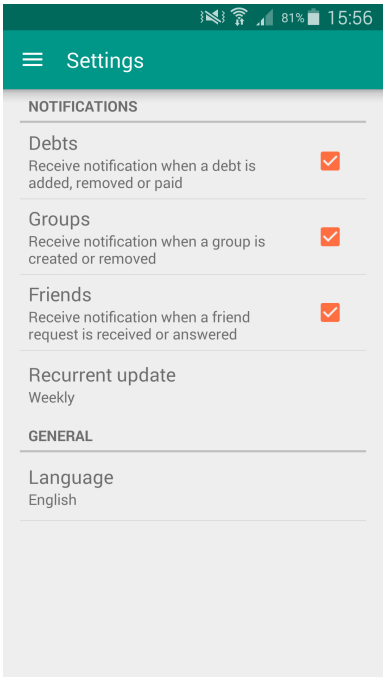

Figure 6.11: Settings view where the user can change their notification settings and application language.

6.2 Back end

In this section some of the more interesting parts of the back end is presented. We also have some small discussion about these parts.

6.2.1 Testing

As previously said in chapter *3.3 External libraries* all methods on the back end have JUnit tests to check their correctness. All methods must be able to handle invalid data and have some way of notifying the front end if something went wrong. This is done through HTTP response codes together with custom status codes defined in the back end's API document. These custom status codes give a more precise description of the error than the HTTP codes usually do. Each of these custom status codes are tested as a part of testing the methods correctness.

Having tests for all exposed methods on the back end has been of great help. Bugs in the code could more easily be tracked down to whether they were a front or back end issue. When changing a feature the tests showed which methods were affected and in what way so that any new issue originating from the feature could be fixed.

6.2.2 REST

REST is currently the most popular style for open APIs [20]. However, the implementation varies quite a lot between the most popular APIs and is often missing one or more of the core principles of REST. This makes it hard for new developers to find the actual best practices as the practices most commonly found on the web are based on the principles used by these faulty APIs.

Because of this, some of the principles of RESTful design were not followed when creating the API for the application. One example is that we did not provide formal descriptions with our methods which is one of the core principles of REST [49]. A formal description is supposed to be parsable by machines so that in- and output from the API can be automatically handled. However, of the 45 most popular APIs only 11% actually provide it [21] and thus this was rarely included in REST tutorials. The lack of formal description has not resulted in any significant problems as the front end and back end teams have been working closely together.

There are some principles which are more commonly used and were included in more tutorials and guides. For example a REST API should always follow the protocol standard of the communication protocol it uses [21]. In our case, along with most other REST APIs, that is HTTP. The correct HTTP verbs should be used and proper status codes should be returned. This principle was taken into consideration when implementing the API. However, after reading the specification of the HTTP verbs more thoroughly it has become clear that `PATCH` is probably the verb which should have been used instead of `PUT` on several methods. `PATCH` should be used when updating a part of a resource and `PUT` when replacing a resource [50]. Most of our update methods use `PUT` but only update a resource partially. This has, however, not lead to any problems for us.

6.3 Optimisation

Most requirements on the optimisation algorithm, as mentioned in *4.4 Optimisation algorithms*, are quite straight forward. Although, the requirement which says that a user is not allowed to have more outgoing debts after an optimisation than before is not as obvious. This requirement exists to avoid a situation where a user has to wait for an incoming debt before being able to pay an outgoing debt. In some specific circumstances debts can be channeled through a user. An exaggerated but clear example is a user with only one optimised debt of 1 kr could after optimisation have one incoming debt of 1 000 kr and one outgoing debt of 1 001 kr. The balance is correct but it is a situation the user would disapprove of.

The development process of the algorithm originated from an idea where an ILP-solver should be used to solve a small portion of the network. Then the algorithm would look for cycles around the edge. When looking into how the cycles would be found and optimised it became clear that an algorithm based on only cycles would fit better into the application. Later on it was discovered that the algorithm had the same core principles as cycle transformation.[29]

6.3.1 Complexity

This variant of cycle transformation has the high worst case complexity $O(n^4)$, where n is the total number of users. Optimising an unoptimised network with a large amount of users and debts would take a long time. The reason why the algorithm is fast, despite having a high complexity, is because it takes advantage of the fact that the network of debts is optimised before a new debt is added. This leads to that the algorithm does not need to reoptimise the whole network every time a debt is added. The algorithm only has to consider the parts of the network that are in close proximity to the added debt. This means that the time it takes to add a debt does not increase with the total number of users. Hence, the complexity is of less importance for this specific usage of the algorithm and the average time for adding a debt will remain constant.

The fact that the algorithm does not have to reoptimise the whole network also helps with reaching the third goal of the optimisation algorithm, which can be seen in *4.4 Optimisation algorithms*. An algorithm which optimises the whole network of original debts can generate a completely different network of optimised debts each time a new debt is added. This can make a user's optimised debts change more frequently than necessary even if the user has not added any debts, which can be confusing. This is avoided since the algorithm only optimises a subset of the network.

6.3.2 Finding cycles

Cycles are found by recursively creating longer and longer paths, starting at a user involved in an added debt. When the other user involved in the added debt is found the cycle is completed. By using this approach shorter cycles, which more commonly can be optimised, will generally be found before longer cycles. This way, less time is being spent on finding long cycles with a lower possibility of being able to be optimised.

A possibility to speed up the process of finding cycles could be to look for cycles from both of the involved users and try to meet in the middle. This way, cycles of the same length could be found by combining two paths of half the length. As creating paths in a normally designed network take exponentially longer time with a higher max length, the creation of all paths would be faster using this alternative approach. There are, however, some very specific networks where higher max length does not necessarily increase the time for creating paths. Furthermore, instead of

spending as much time on creating paths, it would take more time to fit paths together to cycles. Due to time constraints in combination with the fact that the current algorithm met our demands this idea was not explored.

6.3.3 Optimising cycles

If the weight of one of the edges is removed from every edge in a cycle that edge would be removed. The trick is to choose a weight which will also lower the total weight and keep the constraints of the optimisation, in this case mainly the constraint that a user should not have more outgoing debts after an optimisation than before.

In the implementation description explained in *5.3.2 Optimising cycles*, the cycle is tested for optimisation with the smallest weight in each direction. Since the number of edges is to be minimised, the value to remove should correspond to the exact weight of at least one edge. To remove a value corresponding to the smallest weight in one direction, no edges in the cycle will change direction and the risk of breaking the constraint mentioned in the paragraph above decreases. After testing this in several situations, nothing indicates that this method will generate a less optimal solution after the algorithm is finished.

6.3.4 Testing of the optimisation algorithm

The algorithm is tested with several examples where the optimal solution has been calculated manually beforehand. The tests consist of different types of networks where each one of them has different constellations of users and debts. Every test is designed to test a specific difficulty. For example if the algorithm can find and optimise over cycles where there are pairs of users currently without debts between them.

The algorithm has also been tested with large amounts of users and debts in order to test speed and avoid serious errors. An example of a serious error would be if the balance of a user would change after the optimisation. When testing with large networks the optimal solution is unknown and hence it is not tested if the algorithm finds the most optimal solution in these cases. One thing we can see when testing this large number of users or debts is the number of original debts and optimised debts. The result of comparing the original debts with the optimised debts is in line with what to expect from a well optimised network from what we can see from the results of the small network tests. For ensuring the ability to find the best optimisation we rely on the tests of small networks where we can see that the algorithm actually has found the most optimal solution.

The test for large networks generate a pseudorandom network of friends where a user has increased chance of being friends with a friend of a friend than with a random person. This results in a more realistic network where there exists small groups of friends as well as some people who know each other but do not know each others friends. The algorithm has been tested with up to 2 000 users and 20 000 debts. No

serious errors were detected and the optimisation time for each added debt remained the same when the number of users varied, see *Appendix C Speed test of algorithm*. This supports the thesis of the algorithm scaling well with an increased number of users.

6.4 Removed features

During the development of the application some features which were originally in the requirements were not implemented for various reasons. These are discussed in the following section.

6.4.1 Max amount

The initial requirements specified that a user should be able to decide a max amount of how much a friend should be able to owe the user, both before and after optimisation. This feature was created to allow a user to place different level of trust in different friends. This meant both specifying the total amount of debt a specific friend should be able to add towards the user as well as how much of the user's incoming debt could come from the friend.

During implementation of this feature it became clear that it is hard to find a definition of how to calculate the amount a friend owes a user before optimisation. Applying max amount after optimisation is ineffective as demonstrated in the following example. As can be seen in *figure 6.12*, C owes B 10 kr and A has set a max amount of 10 kr to B. Then if B adds a new debt to A, the optimised solution would be that that C should pay A 10 kr instead. This means B can add a new debt of 10 kr to A again, meaning B could effectively add twice the allowed amount.

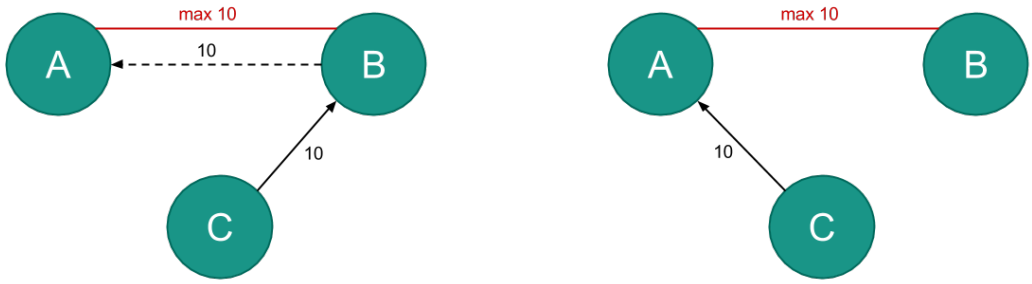

Figure 6.12: A, B and C are friends, the solid black arrow is an existing debt, the dashed arrow is a new debt and the solid red line is the max amount.

6.4.2 Pay original debts

The ability to mark a debt as paid was also a feature which was removed. The feature was supposed to make it easier for users but in some cases it had the opposite effect. As can be seen in *figure 6.13*, A, B and C owe 10 kr in a circle, all three users would have the optimised debt 0 kr. If a user choose to pay an original debt in this

case it would effectively create a new debt and not remove a debt as a user might expect when paying a debt. This problem in combination with the fact that the purpose of the application is to simplify so that the user does not have to manage the original debts and only has to focus on the optimised debts lead to the removal of this feature.

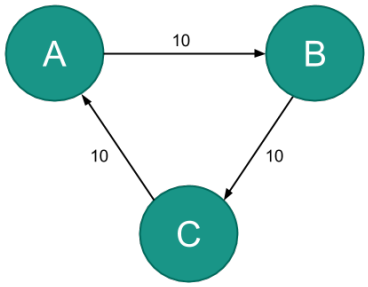

Figure 6.13: A, B and C are friends. The arrows indicate debts.

6.5 Possible improvements

Even though the system fulfills all the planned requirements there are still a lot of improvements and new features that could be added. The application already supports login through Google as a third party alternative but more alternatives like Facebook could be added as well. Matching the friends from a third party service to our application's users could also be done to make it easier for new users to find their friends. Another way which would make it easier for users to handle their friends would be the ability to invite friends to the application by email, or maybe even through text message from the user's phone. A reason for not including more third party alternatives was the amount of time and problems that connecting to a third party entails.

Currently the system does not actually handle any money but is instead a tool to keep track of debts. To make it easier for the users to manage their debts it could be useful to integrate a service which actually can perform payments. Swish and PayPal are such services which potentially could be used.

If a future system would be handling transactions with real money it would require secure communication. Currently the communication between the back end and front end is very insecure. Someone with enough knowledge would easily be able to identify themselves as another user. If the system would handle real money this would be unacceptable and would probably lead to the app being targeted by attackers. The users must feel that they can trust the application so establishing secure connections to the back end would be required before releasing the application in its current state.

For a new user it might be hard to understand all features without some trial-and-error. It would be a good thing to add a guide the first time the user starts the

application that guides the user through the features.

To help users get an overview of their debts some sort of statistics could be added. These statistics could potentially display the number of debts added each month and the total sum of these. Perhaps they could be more detailed and show these for each of the user's friends.

Currently, when adding a debt in the application, there is no way to add a debt where several persons were paying. Even though we did not include this feature in the requirements, this was something we discussed during the implementation and would like to include in a potential update. Furthermore, the view for adding debts in itself is not as intuitive as we would have wanted it to be. Since there is so much information and so many inputs required when adding a debt, it has been hard to fit it all into one page in a good way. Still, we are happy with the result, but there is room for improvements.

On the front end side of the application there are still some bugs that are not resolved yet. If we were to proceed with the implementation and finally release the application, it would be necessary to do a lot more usage testing. Not only to catch the errors, but also to get a better understanding of how user friendly the application is. Testing the front end is rather different comparing to testing the back end and would require a lot of time, which is why we decided to not prioritise usage testing and instead use our knowledge in usability.

6.6 External guidance

As mentioned in *3.4.5 External guidance*, meetings were held with representatives from Jeppesen. The feedback gained from these meetings resulted in a few actions being taken. The most noteworthy being changes to three requirements and an investigation of the bin packing problem, which they saw as a possible solution.

The representatives were open to questions and discussions and were always willing to help. However, we did not ask many questions concerning how to solve our current problems because we wanted to solve them ourselves. The meetings were more about getting feedback on our work from an external source than to get actual help with problems. During the project they were very encouraging and were always positive to our progress which made the work more motivating.

Generally the Jeppesen representatives did not provide suggestions for creating optimisation algorithms, instead they gave feedback on the algorithms we presented. The representatives did not participate in the development of the algorithm we currently use but they approved of the idea when it was presented to them. They also read and commented on the written explanations of the algorithms in this report which have been very helpful.

6.7 Impact on society

It is not realistic to expect this application to make a significant difference for the society even if the use would be widespread. It could however have some impact when it comes to economic or social aspects.

We think that most people have a clear enough view of their personal economy that this application will be nothing more than a convenience for them. But since convenience is a thing that many people strive for we also believe that there are some people who would benefit from the application and gain a better overview and understanding of their personal economy.

The impact on the social aspect is most likely on a similar level. Most people who lend money to their friends have friendships which are mature enough to avoid conflicts caused by borrowed money. The same way as when it comes to private economy there are some people who sometimes handle a situation involving money poorly. In situations where the size, or existence of, a debt is disagreed upon some people would benefit from the application and thereby avoid conflicts.

7
Conclusion

The purpose of this project was to investigate whether an application that keeps track of debts and suggests optimised repayments can be created. The purpose also stated that the application should be intuitive and easy to use. Based on the result we consider this accomplished. Even though a number of minor bugs still exist in the application, all important features work as intended.

We believe the application is user friendly with a clean design. Even though it might be easier to write down a single debt with pen and paper, the advantages of this application appear when there are multiple debts between multiple users. The application creates an easy overview of a user's current debt situation and tells the user how to make their repayments more efficient.

The application follows Material Design and as a result should be easy to understand for new users. The position and layout of menus and buttons could be recognised from other popular applications such as Gmail and Google Maps.

After investigating different algorithms we now have an application that can optimise debt repayments in such a way that both the number of transactions and the total sum of the transactions are lowered. This is functional in both groups and in the contact network as desired. In both cases several types of unwanted behaviour have been eliminated through certain constraints, such as optimised debts changing recipient more frequently than necessary which could confuse the user. The optimisation as well as the whole application is scaleable and it should not be a problem to run the system with a large amount of users.

Bibliography

[1] Statistics Sweden, "Privatpersoners användning av datorer och internet 2013," pp. 9–60. [Online]. Available: http://www.scb.se/Statistik/_Publikationer/LE0108_2013A01_BR_IT01BR1401.pdf [Feb 8, 2015]

[2] Splitwise, Inc, "Split expenses with friends." 2015. [Online]. Available: https://www.splitwise.com/ [May 18, 2015]

[3] John Simon Co, "Pay back - iou manager," 2015. [Online]. Available: https://play.google.com/store/apps/details?id=com.johnsimon.payback [May 18, 2015]

[4] Google Inc, *Material Design*, 2015. [Online]. Available: https://www.google.com/design/spec/material-design/introduction.html# [April 28, 2015]

[5] Networld Media Group, "Balancing mobile payments security vs. ease of use," 2013. [Online]. Available: http://www.mobilepaymentstoday.com/blogs/balancing-mobile-payments-security-vs-ease-of-use/ [May 18, 2015]

[6] Google Inc, *Android, the world's most popular mobile platform*, 2015, creative Commons Attribution 2.5. [Online]. Available: http://developer.android.com/about/index.html [April 27, 2015]

[7] V. H, "Android's google play beats app store with over 1 million apps, now officially largest." *Phonearena.com*, July 24 2013. [Online]. Available: http://www.phonearena.com/news/Androids-Google-Play-beats-App-Store-with-over-1-million-apps-now-officially-largest_id45680 [April 28, 2015]

[8] Statista, "Number of available applications in the google play store from december 2009 to february 2015," 2015. [Online]. Available: http://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/ [April 28, 2015]

[9] Google Inc, *Activities*, 2015, creative Commons Attribution 2.5. [Online]. Available: http://developer.android.com/guide/components/activities.html [April 27, 2015]

[10] ——, *Fragments*, 2015, creative Commons Attribution 2.5. [Online]. Available: http://developer.android.com/guide/components/fragments.html [April 27, 2015]

[11] ——, *Layouts*, 2015, creative Commons Attribution 2.5. [Online]. Available: http://developer.android.com/guide/topics/ui/declaring-layout.html [April 27, 2015]

[12] ——, *Services*, 2015, creative Commons Attribution 2.5. [Online]. Available: http://developer.android.com/guide/components/services.html [April 27, 2015]

[13] ——, *Intents and Intent Filters*, 2015, creative Commons Attribution 2.5. [Online]. Available: http://developer.android.com/guide/components/intents-filters.html [April 27, 2015]

[14] Oracle America, Inc., "What is java technology and why do i need it?" 2015. [Online]. Available: https://www.java.com/en/download/faq/whatis_java.xml [May 4, 2015]

[15] J. Gosling *et al.*, "The java® language specification, java se 8 edition," pp. 1, 373, 2015. [Online]. Available: https://www.java.com/en/download/faq/whatis_java.xml [May 4, 2015]

[16] Oracle America, Inc., *Chapter 6. The Java Virtual Machine Instruction Set*, 2015. [Online]. Available: https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-6.html [May 4, 2015]

[17] The Apache Software Foundation, "The tomcat story," 2015. [Online]. Available: http://tomcat.apache.org/heritage.html [May 5, 2015]

[18] ——, "Documentation index," 2015. [Online]. Available: http://tomcat.apache.org/tomcat-8.0-doc/index.html [May 5, 2015]

[19] R. T. Fielding and R. N. Taylor, "Principled design of the modernweb architecture," Information and Computer Science, University of California, Irvine, Tech. Rep., 2000. [Online]. Available: https://www.ics.uci.edu/~fielding/pubs/webarch_icse2000.pdf [May 7, 2015]

[20] D. Renzel, P. Schlebusch, and R. Klamma, "Today's top "restful" services and why they are not restful," Advanced Community Information Systems (ACIS), RWTH Aachen University, Informatik 5, Germany, Tech. Rep., 2012. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-35063-4_26 [May 7, 2015]

[21] F. Bülthoff and M. Maleshkova, "Restful or restless – current state of today's top web apis," in *The Semantic Web: ESWC 2014 Satellite Events*, ser. Lecture Notes in Computer Science, V. Presutti *et al.*, Eds., vol. 8798. Springer International Publishing, 2014. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-11955-7_6 [May 7, 2015]

[22] The hsql Development Group, "Features summary," 2015. [Online]. Available: http://hsqldb.org/web/hsqlFeatures.html [May 5, 2015]

[23] M. Hofri, *Bin Packing Heuristics*, ser. Texts and Monographs in Computer Science. Springer New York, 1987. [Online]. Available: http://dx.doi.org/10.1007/978-1-4612-4800-2_5

[24] K. Sim and E. Hart, "Generating single and multiple cooperative heuristics for the one dimensional bin packing problem using a single node genetic programming island model," in *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '13. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2463372.2463555 [May 8, 2015]

[25] S. Vajda, *LINEAR PROGRAMMING: Algorithms and applications*. Springer Netherlands, 1981.

[26] J. Noyes and E. W. Weisstein, "Linear programming," 2015. [Online]. Available: http://mathworld.wolfram.com/LinearProgramming.html [May 10, 2015]

[27] A. Schrijver, *Theory of Linear and Integer Programming*, ser. Wiely - interscience series in discrete mathematics and optimization. John Wiley and Sons, Chichester, 1986. [Online]. Available: https://promathmedia.files.wordpress.com/2013/10/alexander_schrijver_theory_of_linear_and_integerbookfi-org.pdf [May 8, 2015]

[28] M. Cadoli, "The expressive power of binary linear programming," Dipartimento di Informatica e Sistemistica, Universitá di Roma "La Sapienza", Italy, Tech. Rep., 2001. [Online]. Available: http://dx.doi.org/10.1007/3-540-45578-7_41 [May 8, 2015]

[29] T. Verhoeff, "Settling multiple debts efficiently: An invitation to computing science," Faculty of Mathematics and Computing Science Eindhoven University of Technology, Tech. Rep., 2003. [Online]. Available: http://www.mathmeth.com/tom/files/settling-debts.pdf [Apr 27, 2015]

[30] Google Inc, *Application Fundamentals*, 2015, creative Commons Attribution 2.5. [Online]. Available: http://developer.android.com/guide/components/fundamentals.html [May 9, 2015]

[31] ——, *UI Overview*, 2015, creative Commons Attribution 2.5. [Online]. Available: http://developer.android.com/guide/topics/ui/overview.html [May 9, 2015]

[32] ——, *Saving Data in SQL Databases*, 2015, creative Commons Attribution 2.5. [Online]. Available: http://developer.android.com/training/basics/data-storage/databases.html [May 29, 2015]

[33] Internet Engineering Task Force, "Introducing json," 2015. [Online]. Available: http://json.org/ [Apr 25, 2015]

[34] Google Inc, *Android Studio Overview*, 2015, creative Commons Attribution 2.5. [Online]. Available: http://developer.android.com/tools/studio/index.html [May 8, 2015]

[35] Oracle Corporation, *JavaTM Platform, Enterprise Edition 6 API Specification*, 2011. [Online]. Available: http://docs.oracle.com/javaee/6/api/ [May 4, 2015]

[36] ——, "Restful web services in java," 2015. [Online]. Available: https://jersey.java.net/ [Apr 15, 2015]

[37] Square Inc, *Introduction*, 2015. [Online]. Available: http://square.github.io/retrofit/ [Mar 14, 2015]

[38] Google Inc, *Using OAuth 2.0 to Access Google APIs*, 2015. [Online]. Available: https://developers.google.com/identity/protocols/OAuth2 [May 7, 2015]

[39] ——, *Google Cloud Messaging for Android*, 2015, creative Commons Attribution 2.5. [Online]. Available: https://developer.android.com/google/gcm/index.html [May 6, 2015]

[40] The Apache Software Foundation, "About," 2014. [Online]. Available: http://junit.org/ [May 9, 2015]

[41] R. Weidner, "An introduction to scrum," *Scrum Alliance*.

[42] Atlassian, "Comparing workflows," 2015. [Online]. Available: http://developer.android.com/tools/studio/index.html [Mar 14, 2015]

[43] Jeppesen Systems AB. (2015) What we do. [Online]. Available: http://ww1.jeppesen.com/company/about/what-we-do.jsp [May 12, 2015]

[44] M. Berkelaar, *Introduction to lp_solve 5.5.2.0*, 2015. [Online]. Available: http://lpsolve.sourceforge.net/5.5/ [Apr 19, 2015]

[45] S. Gass and M. Fu, Eds., *Simplex Method (Algorithm)*. Springer US, 2013, pp. 1394–1395. [Online]. Available: http://dx.doi.org/10.1007/978-1-4419-1153-7_200768 [May 4, 2015]

[46] M. Bhatti, *Linear Programming*. Springer New York, 2000, pp. 315–436. [Online]. Available: http://dx.doi.org/10.1007/978-1-4612-0501-2_6 [Apr 28, 2015]

[47] M. S. Bazaraa and O. Kirca, "A branch-and-bound-based heuristic for solving the quadratic assignment problem," *Naval Research Logistics Quarterly*, vol. 30, no. 2, pp. 287–304, 1983. [Online]. Available: http://dx.doi.org/10.1002/nav.3800300210 [May 7, 2015]

[48] Google Inc, *Content Providers*, 2015, creative Commons Attribution 2.5. [Online]. Available: http://developer.android.com/guide/topics/providers/content-providers.html [April 27, 2015]

[49] R. T. Fielding, “Rest apis must be hypertext-driven,” 2008. [Online]. Available: http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven [May 8, 2015]

[50] ——, *Hypertext Transfer Protocol – HTTP/1.1*, Internet Engineering Task Force. [Online]. Available: https://www.ietf.org/rfc/rfc2068.txt [May 6, 2015]

A

Gantt chart

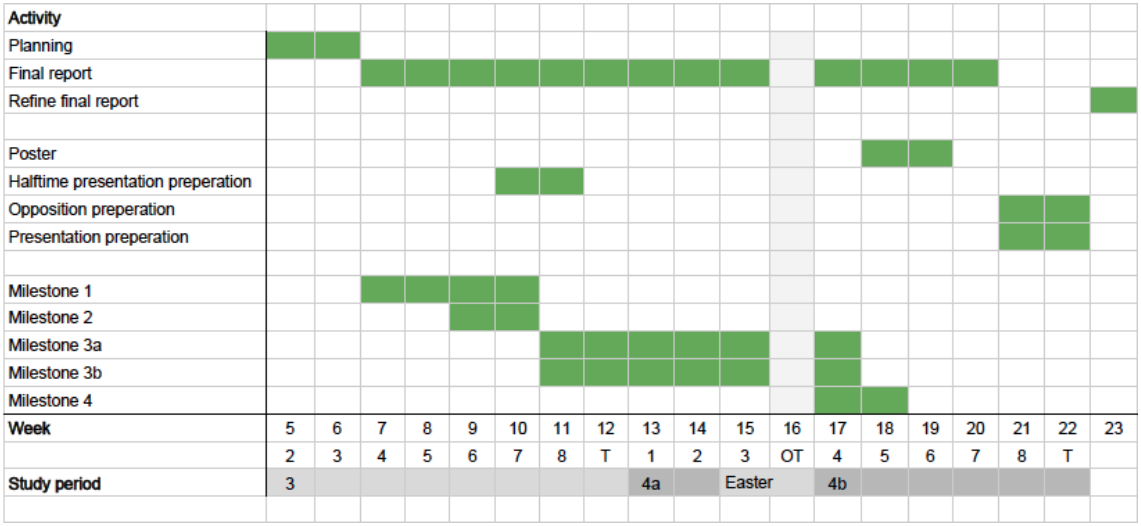

Figure A.1: Gantt chart

B

Requirements

All requirements are sorted by prio. However, they are not ordered under each priority.

1. Prio 1
   - 1.1. The user should be able to create an account.
   - 1.2. The user should be able to log in.
   - 1.3. The user should be able to log out.
   - 1.4. The user should be able to add debts/credits to a single user.
   - 1.5. The user should be able to mark an (optimized) debt as paid.
     - 1.5.1. A debt can not be partly paid.
   - 1.6. The user should be able to remove debts/credits where he/she is one of the parties involved.
   - 1.7. The user should be able to add and remove friends.
     - 1.7.1. Adding a friend is done by a request which requires a positive answer before two users become friends.
     - 1.7.2. Removing a friend also removes yourself from the friend's friends.
     - 1.7.3. A friend can only be removed if the optimized balance towards that friend is zero.
   - 1.8. The user should be able to view the information about a friend, including name, profile picture and email.
   - 1.9. The user should be able to see his/her optimized debt/credit.
   - 1.10. The system should be able to optimize the network of debts/credits mainly to as few and secondary to as small transactions as possible.
     - 1.10.1. After optimization, a debt/credit can only exist if the two parties are friends.
   - 1.11. The system should send notifications to users when a new debt or credit where the user is one of the parties involved.
2. Prio 2:
   - 2.1. The user should be able to see a feed of all events connected to the user.
     - 2.1.1. friend request received and other use accepting my friend request,

2.1.2. user added/removed to a closed group,
2.1.3. debt/credit added/paid,/removed

2.2. The user should be able to create/delete/change/view a quick selection.

2.3. The user should be able to add a debt to multiple users, either by manually choosing the users involved or use a premade list of users. It should be possible to specify who the debtor is.

2.4. The user should be able to find friends by searching by:

2.4.1. the friend's email address,
2.4.2. the friend's display name,
2.4.3. the friend's phone number.

2.5. The user should be able to view the notification settings.

2.6. The user should be able to change the notification settings.

2.7. The user should be able to send push notification to another user.

2.8. The user should be able to view and change his/her user information.

3. Prio 3:

3.1. The system should send notifications each month with a short summary of the recent debts and credits since last notification.

3.2. The application should have a help / about view with information how to use it.

3.3. The user should be able to create / link an account using his/her Google account.

3.4. The user should be able to create closed groups in which debts can be added to and where optimization is only applied inside the group.

3.4.1. A closed group can be admin-controlled.
3.4.2. A closed group can contain temporary users and ordinary users.
3.4.3. When a user gets invited to a closed group the user will automatically join with the option to decline if the user who invites is a friend. Otherwise the invite needs to be accepted before the user joins the group.
3.4.4. Each user in a closed group can add/delete debts/credits in it, unless it is admin-controlled, then only the admin can add debts/credits.
3.4.5. The creator of the group or admin in an admin group can decide if all debts should be visible for all or just the involved users.
3.4.6. A debt in a closed group can be moved to the network. This can only be done if the involved users are friends. This can be compared to removing an optimized pay in a closed group and adding a debt for the same amount out of the group.

3.5. The user should be able to remove a closed group when all debts in it have been paid or moved to the network.

4. Prio 4:
    4.1. When adding a debt to multiple users, the user should be able to specify in percentage or amount of how much of the debt each user owes.
    4.2. The user should be able to remove his/her account.
    4.3. The user should be able to create / link an account using his/her Facebook account.
    4.4. The user should be able to add / change his/her profile picture.
    4.5. The user should be able to view statistics over his/her debts and credit.
        4.5.1. How much debt.
        4.5.2. How much credit.
        4.5.3. When the user pay back.
        4.5.4. When the user's friends pay back.
    4.6. The user should be able to change language of the application.
        4.6.1. Swedish and English
5. Prio 5 - Outside scope
    5.1. The user should be able to use Swish to pay the debts.
    5.2. The user should be able to use PayPal to pay the debts.
    5.3. The system should be secure to use by:
    5.4. Encrypting data between user and server
    5.5. The user should be able to invite friends who do not yet use the application.

C

Speed test of algorithm

Debts are added to a pseudorandom network of friends where a user has increased chance of being friends with a friend of a friend than with a random person. This results in a more realistic network where there exists small groups of friends as well as some people who know each other but do not know each others friends.

The number of debts added is within five percent of six times the number of users. The tests ran on a 2.3GHz processor as a single process.

Number of users	100	200	300	400	500	1 000	2 000
Average time per debt (ms)	4	4	4	4	4	4	4